USENIX

conference proceedings

Proceedings of NSDI '10: 7th USENIX Symposium on Networked Systems Design and Implementation

San Jose, CA, USA April 28–30, 2010

NSDI '10: 7th USENIX Symposium on Networked Systems Design and Implementation

San Jose, CA, USA

April 28–30, 2010

Sponsored by

USENIX

in cooperation with ACM SIGCOMM and ACM SIGOPS

© 2010 by The USENIX Association
All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

ISBN 978-931971-73-7

USENIX Association

Proceedings of NSDI ’10: 7th USENIX Symposium on Networked Systems Design and Implementation

April 28–30, 2010
San Jose, CA, USA

Conference Organizers

Program Co-Chairs
Miguel Castro, *Microsoft Research Cambridge*
Alex C. Snoeren, *University of California, San Diego*

Program Committee
Lorenzo Alvisi, *University of Texas at Austin*
Hari Balakrishnan, *Massachusetts Institute of Technology*
John Byers, *Boston University*
Patrick Crowley, *Washington University in St. Louis*
Jeff Dean, *Google, Inc.*
Nick Feamster, *Georgia Institute of Technology*
Michael J. Freedman, *Princeton University*
Charles Killian, *Purdue University*
Dejan Kostić, *École Polytechnique Fédérale de Lausanne*
Philip Levis, *Stanford University*
Petros Maniatis, *Intel Research Berkeley*
Michael Mitzenmacher, *Harvard University*
Sue Moon, *KAIST*
Venkat Padmanabhan, *Microsoft Research India*
Sylvia Ratnasamy, *Intel Research Berkeley*
Jennifer Rexford, *Princeton University*
Rodrigo Rodrigues, *Max Planck Institute for Software Systems*
Timothy Roscoe, *ETH Zurich*
Antony Rowstron, *Microsoft Research Cambridge*
Stefan Savage, *University of California, San Diego*
Srinivasan Seshan, *Carnegie Mellon University*
David Wetherall, *University of Washington*
Ellen Zegura, *Georgia Institute of Technology*

Poster Session Chair
Charles Killian, *Purdue University*

Steeering Committee
Thomas Anderson, *University of Washington*
Brian Noble, *University of Michigan*
Jennifer Rexford, *Princeton University*
Mike Schroeder, *Microsoft Research*
Chandu Thekkath, *Microsoft Research*
Amin Vahdat, *University of California, San Diego*
Ellie Young, *USENIX Association*

The USENIX Association Staff

External Reviewers

Kevin Almeroth
Jill Boyce
Mung Chiang
George Danezis
Sonia Fahmy
Pedro Fonseca
Nate Foster
Sharon Goldberg
Ben Greenstein
Andreas Haeberlen
Daniel Halperin
Wenjun Hu
Eric Keller
Katrina LaCurts
Nick McKeown
Alan Mislove
Ansley Post
Asfandyar Qureshi
Bozidar Radunovic
Anmol Sheth
Atul Singh
Martin Suchara
Mukarram bin Tariq
Luis Veiga
Mythili Vutukuru
Alexander Wieder
Minlan Yu

NSDI '10: 7th USENIX Symposium on Networked Systems Design and Implementation
April 28–30, 2010
San Jose, CA, USA

Message from the Program Co-Chairs. vii

Wednesday, April 28

Cloud Services

Centrifuge: Integrated Lease Management and Partitioning for Cloud Services 1
Atul Adya, Google; John Dunagan and Alec Wolman, Microsoft Research

Volley: Automated Data Placement for Geo-Distributed Cloud Services 17
Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, and Alec Wolman, Microsoft Research; Harbinder Bhogan, University of Toronto

Optimizing Cost and Performance in Online Service Provider Networks 33
Zheng Zhang, Purdue University; Ming Zhang and Albert Greenberg, Microsoft Research; Y. Charlie Hu, Purdue University; Ratul Mahajan, Microsoft Research; Blaine Christian, Microsoft Corporation

Wireless 1

Exploring Link Correlation for Efficient Flooding in Wireless Sensor Networks 49
Ting Zhu, Ziguo Zhong, Tian He, and Zhi-Li Zhang, University of Minnesota, Twin Cities

Supporting Demanding Wireless Applications with Frequency-agile Radios 65
Lei Yang, University of California, Santa Barbara; Wei Hou, Tsinghua University; Lili Cao, Ben Y. Zhao, and Haitao Zheng, University of California, Santa Barbara

Peer-to-Peer

Contracts: Practical Contribution Incentives for P2P Live Streaming 81
Michael Piatek and Arvind Krishnamurthy, University of Washington; Arun Venkataramani, University of Massachusetts; Richard Yang, Yale University; David Zhang, PPLive; Alexander Jaffe, University of Washington

Experiences with CoralCDN: A Five-Year Operational View 95
Michael J. Freedman, Princeton University

Whānau: A Sybil-proof Distributed Hash Table 111
Chris Lesniewski-Laas and M. Frans Kaashoek, MIT CSAIL

Web Services 1

Crom: Faster Web Browsing Using Speculative Execution 127
James Mickens, Jeremy Elson, Jon Howell, and Jay Lorch, Microsoft Research

WebProphet: Automating Performance Prediction for Web Services 143
Zhichun Li, Northwestern University; Ming Zhang, Microsoft Research; Zhaosheng Zhu, Data Domain Inc.; Yan Chen, Northwestern University; Albert Greenberg and Yi-Min Wang, Microsoft Research

Mugshot: Deterministic Capture and Replay for JavaScript Applications 159
James Mickens, Jeremy Elson, and Jon Howell, Microsoft Research

Thursday, April 29

Wireless 2

AccuRate: Constellation Based Rate Estimation in Wireless Networks 175
Souvik Sen, Naveen Santhapuri, and Romit Roy Choudhury, Duke University; Srihari Nelakuditi, University of South Carolina

Scalable WiFi Media Delivery through Adaptive Broadcasts 191
Sayandeep Sen, Neel Kamal Madabhushi, and Suman Banerjee, University of Wisconsin—Madison

Maranello: Practical Partial Packet Recovery for 802.11 205
Bo Han and Aaron Schulman, University of Maryland; Francesco Gringoli, University of Brescia; Neil Spring and Bobby Bhattacharjee, University of Maryland; Lorenzo Nava, University of Brescia; Lusheng Ji, Seungjoon Lee, and Robert Miller, AT&T Labs—Research

Routing

Reverse traceroute 219
Ethan Katz-Bassett, University of Washington; Harsha V. Madhyastha, University of California, San Diego; Vijay Kumar Adhikari, University of Minnesota; Colin Scott, Justine Sherry, Peter van Wesep, Thomas Anderson, and Arvind Krishnamurthy, University of Washington

Seamless BGP Migration with Router Grafting 235
Eric Keller and Jennifer Rexford, Princeton University; Jacobus van der Merwe, AT&T Labs—Research

Datacenter Networking

ElasticTree: Saving Energy in Data Center Networks 249
Brandon Heller, Stanford University; Srini Seetharaman, Deutsche Telekom R&D Lab; Priya Mahadevan, Hewlett-Packard Labs; Yiannis Yiakoumis, Stanford University; Puneet Sharma and Sujata Banerjee, Hewlett-Packard Labs; Nick McKeown, Stanford University

SPAIN: COTS Data-Center Ethernet for Multipathing over Arbitrary Topologies 265
Jayaram Mudigonda and Praveen Yalagandula, HP Labs; Mohammad Al-Fares, University of California, San Diego; Jeffrey C. Mogul, HP Labs

Hedera: Dynamic Flow Scheduling for Data Center Networks. 281
Mohammad Al-Fares and Sivasankar Radhakrishnan, University of California, San Diego; Barath Raghavan, Williams College; Nelson Huang and Amin Vahdat, University of California, San Diego

Improving MapReduce

Airavat: Security and Privacy for MapReduce 297
Indrajit Roy, Srinath T.V. Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel, The University of Texas at Austin

MapReduce Online 313
Tyson Condie, Neil Conway, Peter Alvaro, and Joseph M. Hellerstein, University of California, Berkeley; Khaled Elmeleegy and Russell Sears, Yahoo! Research

Web Services 2

The Architecture and Implementation of an Extensible Web Crawler 329
Jonathan M. Hsieh, Steven D. Gribble, and Henry M. Levy, University of Washington

Prophecy: Using History for High-Throughput Fault Tolerance 345
Siddhartha Sen, Wyatt Lloyd, and Michael J. Freedman, Princeton University

Friday, April 30

Malware

Carousel: Scalable Logging for Intrusion Prevention Systems 361
Vinh The Lam, University of California, San Diego; Michael Mitzenmacher, Harvard University; George Varghese, University of California, San Diego

SplitScreen: Enabling Efficient, Distributed Malware Detection 377
Sang Kil Cha, Iulian Moraru, Jiyong Jang, John Truelove, David Brumley, and David G. Andersen, Carnegie Mellon University

Behavioral Clustering of HTTP-Based Malware and Signature Generation Using Malicious Network Traces . . .391
Roberto Perdisci, Georgia Institute of Technology and Damballa, Inc.; Wenke Lee and Nick Feamster, Georgia Institute of Technology

Network Performance

Glasnost: Enabling End Users to Detect Traffic Differentiation 405
Marcel Dischinger and Massimiliano Marcon, MPI-SWS; Saikat Guha, MPI-SWS and Microsoft Research; Krishna P. Gummadi, MPI-SWS; Ratul Mahajan and Stefan Saroiu, Microsoft Research

EndRE: An End-System Redundancy Elimination Service for Enterprises 419
Bhavish Aggarwal, Microsoft Research India; Aditya Akella and Ashok Anand, University of Wisconsin—Madison; Athula Balachandran, Carnegie Mellon University; Pushkar Chitnis, Microsoft Research India; Chitra Muthukrishnan, University of Wisconsin—Madison; Ramachandran Ramjee, Microsoft Research India; George Varghese, University of California, San Diego

Cheap and Large CAMs for High Performance Data-Intensive Networked Systems. 433
Ashok Anand, Chitra Muthukrishnan, Steven Kappes, and Aditya Akella, University of Wisconsin—Madison; Suman Nath, Microsoft Research

Message from the Program Co-Chairs

NSDI '10 carries on the conference's tradition of presenting the best work in the area of networked systems. We have a strong program, with papers spanning a broad range of topics in systems and networking.

We received 175 paper submissions, tying the 2008 record. We rejected three submissions without review, because they violated the formatting requirements in the Call for Papers by more than 10%. Each of the remaining submissions was reviewed by at least three program committee members. We selected 85 submissions for a second round of reviewing. These submissions each received at least two additional reviews by program committee members. In a small number of cases, we used external reviewers to complement the expertise of the program committee. All 25 members of the program committee attended the meeting at MIT on December 14 in person. Discussions at the program committee meeting led to a final program with 29 papers. Because of the special role conferences play in our field, all papers were shepherded by a program committee member and, where supported by the shepherd, received extra pages to address reviewers' comments.

This year NSDI has moved to entirely electronic proceedings. In addition to the environmental implications, the elimination of incremental per-page costs allowed us to be extremely generous with additional pages; over half of the authors took advantage of the opportunity to include up to two pages of supplementary content. While we believe the quality of a number of papers was significantly improved by the extended length, we encourage the community to consider the ramifications of a substantially longer final version in the general case. We welcome your feedback on this and other ways that NSDI can potentially leverage its new, all-electronic format.

We are grateful to everyone whose hard work made this conference possible. Most of all, we are indebted to all of the authors who submitted their work to this conference. We thank the program committee for their dedication and hard work in reviewing papers and participating in the extensive discussions at the PC meeting, as well as in shepherding the final versions. We thank our external reviewers for lending their expertise on short notice. We extend special thanks to Hari Balakrishnan and Sheila Marian for hosting the program committee meeting at MIT. We are grateful to the conference sponsors for their support and to the USENIX staff for handling the conference logistics, marketing, and proceedings publication; it is a pleasure to work with them. Eddie Kohler and Geoff Voelker continue to provide invaluable service to the community by providing and supporting their HotCRP reviewing system and Banal format checker. Finally, we thank the NSDI '10 attendees and future readers of these papers: in the end, it is your interest in this work that makes all of these efforts worthwhile.

Miguel Castro, ***Microsoft Research Cambridge***
Alex C. Snoeren, ***University of California, San Diego***

Centrifuge: Integrated Lease Management and Partitioning for Cloud Services

Atul Adya[†], John Dunagan[*], Alec Wolman[*]
[†]Google, [*]Microsoft Research

Abstract: *Making cloud services responsive is critical to providing a compelling user experience. Many large-scale sites, including LinkedIn, Digg and Facebook, address this need by deploying pools of servers that operate purely on in-memory state. Unfortunately, current technologies for partitioning requests across these in-memory server pools, such as network load balancers, lead to a frustrating programming model where requests for the same state may arrive at different servers. Leases are a well-known technique that can provide a better programming model by assigning each piece of state to a single server. However, in-memory server pools host an extremely large number of items, and granting a lease per item requires fine-grained leasing that is not supported in prior datacenter lease managers.*

This paper presents Centrifuge, a datacenter lease manager that solves this problem by integrating partitioning and lease management. Centrifuge consists of a set of libraries linked in by the in-memory servers and a replicated state machine that assigns responsibility for data items (including leases) to these servers. Centrifuge has been implemented and deployed in production as part of Microsoft's Live Mesh, a large-scale commercial cloud service in continuous operation since April 2008. When cloud services within Mesh were built using Centrifuge, they required fewer lines of code and did not need to introduce their own subtle protocols for distributed consistency. As cloud services become ever more complicated, this kind of reduction in complexity is an increasingly urgent need.

1 Introduction

Responsiveness is critical to delivering compelling cloud services. Many large-scale sites, including LinkedIn, Digg and Facebook, address the simultaneous needs of scale and low-latency by partitioning their user data across pools of servers that operate purely on in-memory state [36, 34, 35, 21, 23]. Processing most operations directly out of memory yields low latency responses. These sites achieve reliability by using some separate service (such as a replicated database) to reload the data into the server pool in the event of a failure.

Unfortunately, current technologies for building in-memory server pools lead to a frustrating programming model. Many sites use load balancers to distribute requests across such servers pools, but load balancers force the programmer to handle difficult corner cases: requests for the same state may arrive at different servers, leading to multiple potentially inconsistent versions. For example, in a cloud-based video conferencing service, the data items being partitioned might be metadata for individual video conferences, such as the address of that conference's rendezvous server. Inconsistencies can lead to users selecting different rendezvous points, and thus being unable to connect even when they are both online. The need to deal with these inconsistencies drastically increases the burden on service programmers. In our video conferencing example, the developer could reduce the chance that two nodes would fail to rendezvous by implementing quorum reads and writes on the in-memory servers. In other cases, programmers are faced with supporting application-specific reconciliation, a problem that is known to be difficult [39, 33, 17].

This paper describes Centrifuge, a system for building in-memory server pools that eliminates most of the distributed systems complexity, allowing service programmers to focus on the logic particular to their service. Centrifuge does this by implementing both lease management and partitioning using a replicated state machine. Leases are a well-known technique for ensuring that only one server at a time is responsible for a given piece of state [3]. Partitioning refers to assigning each piece of state to an in-memory server; requests are then sent to the appropriate server. To support partitioning, the replicated state machine implements a membership service and dynamic load management. Partitioning for in-memory servers additionally requires a mechanism to deal with state loss. Centrifuge addresses this need with explicit API support for recovery: it notifies the service indicating which state has been lost and needs to be recovered, e.g., because a machine crashed and lost its lease. Centrifuge does not recover the state itself so that appli-

[†]adya@google.com. Work done while at Microsoft.

e.g., recovering from a variety of datacenter storage systems or even relying on clients to re-publish state into the system. Relying on client republishing is the approach taken by the Live Mesh services [22], and we describe this in more detail in Section 4. This combination of functionality allows Centrifuge to replace most datacenter load balancers and simultaneously provide a simpler programming model.

Providing both lease management and partitioning is valuable to the application developer, but it leads to a scalability challenge in implementing Centrifuge. Each in-memory server may hold hundreds of millions of items, and there may be hundreds of such servers. Naively supporting fine-grained leases (one for each item) allows flexible load management, but it could require a large number of servers dedicated solely to lease traffic. In contrast, integrating leasing and partitioning allows Centrifuge to provide the benefits of fine-grained leases without their associated scalability costs.

In particular, integrating leasing and partitioning allows Centrifuge to incorporate the following techniques: leases on variable-length ranges, manager-directed leasing, and conveying the partitioning assignment through leases. Variable-length ranges specify contiguous portions of a flat namespace that are assigned as a single lease. Internally, Centrifuge's partitioning algorithm uses consistent hashing to determine the variable-length ranges. Manager-directed leasing avoids the problem of lease fragmentation. It allows the manager to change the length of the ranges being leased so that load can be shed at fine granularities, while simultaneously keeping the number of leases small. This is in contrast to the traditional model where clients request leases from a manager, which can potentially degenerate into requiring one lease for each item. Manager-directed leasing also leads to changes in the leasing API: instead of clients requesting individual leases, they simply ask which leases they have been assigned. Finally, because the lease and partitioning assignments are both being performed by the manager, there is no need for separate protocols for these two tasks: the lease protocol implicitly conveys the results of the partitioning algorithm.

Centrifuge has been implemented and deployed in production as part of Microsoft's Live Mesh, a large-scale commercial cloud service in continuous operation since April 2008 [22]. As of March 2009, it is in active use by five Live Mesh component services spanning hundreds of servers. As we describe in Section 4, Centrifuge successfully hid most of the distributed systems complexity from the developers of these component services: the services were built with fewer lines of code and without needing to introduce their own subtle protocols for distributed consistency or application-specific reconciliation. As cloud services become ever more complicated, reducing this kind of complexity is an increasingly urgent need.

To summarize, this paper's main contributions are:

- we demonstrate that integrating leasing and partitioning can provide the benefits of fine-grained leases to in-memory server pools without their associated scalability costs;
- we show that real-world cloud services written by other developers are simplified by using Centrifuge; and
- we provide performance results from Centrifuge in production as well as a testbed evaluation.

The remainder of this paper is organized as follows: In Section 2, we describe the design and implementation of Centrifuge. In Section 3, we explain the Centrifuge API through an example application. In Section 4, we describe how three real-world cloud services were simplified using Centrifuge. In Section 5, we report on the behavior of Centrifuge in production and we evaluate Centrifuge on a testbed. In Section 6, we describe related work. In Section 7, we conclude.

2 Design and Implementation

The design of Centrifuge is motivated by the needs of in-memory server pools. Centrifuge is designed to support these servers executing arbitrary application logic on any particular in-memory data item they hold, and Centrifuge helps route requests to the server assigned a lease for any given piece of data, enabling the computation and data to be co-located. The nature of the in-memory data covered by the lease is service-specific, e.g., it could be the rendezvous information for the currently connected participants in a video-conferencing session, or a queue of messages waiting for a user who is currently offline. Furthermore, Centrifuge does not store this data on behalf of the application running on the in-memory server (i.e., Centrifuge is not a distributed cache). Instead, the application manages the relationship between the Centrifuge lease and its own in-memory data. Centrifuge has no knowledge of the application's data; it only knows about the lease.

A common design pattern in industry for a datacenter in-memory server pool is to spread hundreds of millions of objects across hundreds of servers [36, 34, 35, 21, 23]. Additionally, these applications are designed so that even the most heavily loaded object requires much less than one machine's worth of processing power. As a result, each object can be fully handled by one machine holding an exclusive lease, thereby eliminating the usefulness of read-only leases. Because of this, Centrifuge only grants exclusive leases, simplifying its API and internal design

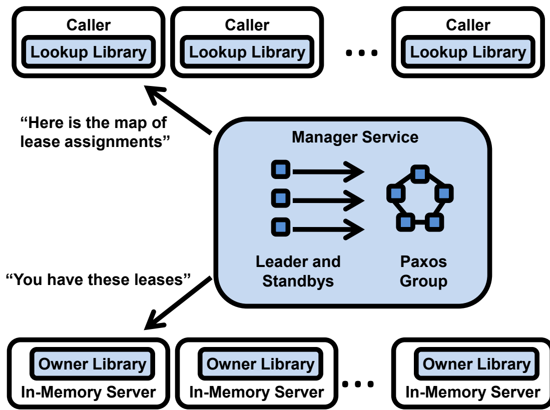


Figure 1: *Servers using Centrifuge link in libraries that talk to a Centrifuge Manager service.*

without compromising its usefulness in this application domain.

Centrifuge's architecture is shown in Figure 1. Servers that want to send requests link in a *Lookup* library, while servers that want to receive leases and process requests link in an *Owner* library. In our video conferencing example, web server frontends would link in the Lookup library and forward requests for a particular conference's rendezvous information to the appropriate in-memory server linking in the Owner library. Both the libraries communicate with a logically centralized *Manager* service that is implemented using a replicated state machine.

At a high level, the job of the Manager service is to partition a flat namespace of "keys" among all the servers linking in Owner libraries. The Manager service does this by mapping the key space into variable-length ranges using consistent hashing [6] with 64 virtual nodes per Owner library. The Manager then conveys to each Owner library its subset of the map (i.e., its partitioning assignment) using a lease protocol. We refer to this technique as manager-directed leasing: the Centrifuge manager controls how the key space is partitioned and assigns leases directly on the variable-length ranges associated with these partitions. As a result, the manager avoids the scalability problems traditionally associated with fine-grained leasing.

When a new Owner library contacts the Manager service, the Manager service recalls the needed leases from other Owner libraries and grants them to the new Owner library. Centrifuge also reassigns leases for adaptive load management (described in more detail in Section 2.4). Finally, Lookup libraries contact the Manager service to learn the entire map, enabling them to route a request to any Owner.

We briefly explain the usage of Centrifuge by walking through its use in Live Mesh's Publish-subscribe service,

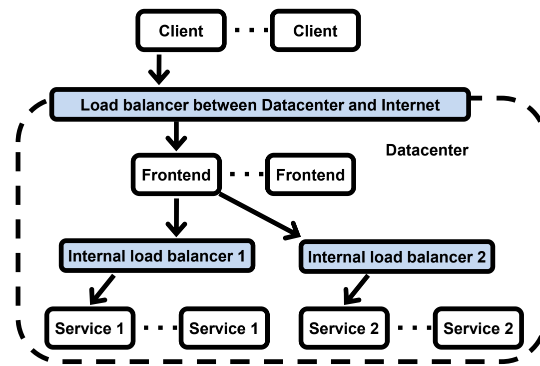


Figure 2: *Datacenter applications are often divided into multiple component services, and servicing clients requests frequently requires communicating with multiple such services. Centrifuge is designed to replace only the internal network load balancers used by the component services.*

described in more detail in Section 4. Servers that wish to publish events to topics link in the Lookup library; they lookup the server where a given topic is hosted using the hash of the topic name as the lookup key. The servers hosting these topics link in the Owner library; they receive leases on the topics based on the hash of the topic name. When a server has an event to publish, it makes a call to its Lookup library, gets the address of the appropriate server hosting the topic, and then sends the publish message to this server. When this server receives the message, it checks with its Owner library that it holds the lease on this particular topic, and then forwards the event to all subscribed parties.

Centrifuge is designed for services that route requests which both originate and terminate within the datacenter. This is depicted in Figure 2. Datacenter applications often include many such internal services: for example, LinkedIn reports having divided their datacenter application into a client-facing frontend and multiple internal services, such as news, profiles, groups and communications [36]. In Section 4, we describe how the Live Mesh application similarly contains multiple internal services that use Centrifuge. If requests originate outside the datacenter (e.g., from web browsers), using Centrifuge requires an additional routing step: requests first traverse a traditional network load balancer to arrive at frontends (e.g., web servers) that link in the Lookup library, and they are then forwarded to in-memory servers that link in the Owner library.

2.1 Manager Service

To describe the Manager service, we first present the high availability design. We then present the logic for lease management, partitioning, and adaptive load management.

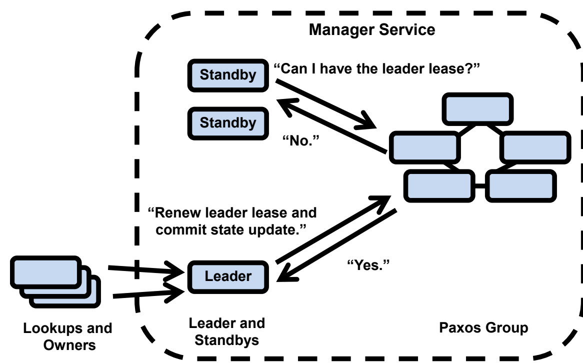


Figure 3: *In the Manager service, one set of servers run a Paxos group that provides a state store and a leader election protocol, and another set of servers act as either leader or standby. Only the current leader executes the logic for partitioning, lease management, and communication with Lookups and Owners.*

2.1.1 Leader Election and High Availability

The Manager service's high availability design is depicted in Figure 3. At a high level, the Manager service consists of two sets of servers: one set of servers provides a Paxos group, and the other set of servers act as either leader or standby. In detail, the Paxos group is used to elect a current leader from the set of standby servers, and to provide a highly-available store used by the leader and standbys. The current leader executes the logic around granting leases to Owner libraries, partitioning, and the protocols used to communicate with the Owner and Lookup libraries (or simply, the Owners and Lookups). Every time the leader receives a request that requires it to update its internal state, it commits the state change to the Paxos group before responding to the caller. To deal with the case that the leader becomes unresponsive, all the standby servers periodically ask the Paxos group to become the leader; if a new standby becomes the leader, it reads in all the state from the Paxos group, and then resumes processing where the previous leader left off.

In this high availability design, Paxos is only used to implement a leader election protocol and a highly available state store. Most of the complicated program logic runs in the leader and can be non-deterministic. Thus, this split minimizes the well-known difficulties of writing deterministic code within a Paxos group [37, 27]. A similar division of responsibility was also used in the Chubby datacenter lease manager [3, 4].

At a logical level, all the Owners and Lookups can simply send all requests to every leader and standby; they will only ever get a response from the leader. For efficiency, the Owners and Lookups only send requests to the server they believe to be the leader unless that server becomes unresponsive. If the leader has become unresponsive, the Owners and Lookups start broadcasting their messages to all the leader or standby servers until one replies, and they then switch back to sending their requests to the one leader. Owners and Lookups learn of the Manager nodes through an external configuration file which can be updated when new Manager nodes are placed into service.

The configuration that we use in deployment is three standby servers and five servers running Paxos. This allows the service to continue operating in the event of any two machine failures: the Paxos group requires three of its five servers to be operational in order to form a majority, while any one standby can become the leader and take over communication with all the Owners and Lookups. For simplicity, we will hereafter refer to the current leader in the Manager service as just the Manager.

2.1.2 Partitioning, Leasing and Load Management

To implement partitioning and lease management, the Manager maintains a set of namespaces, one per pool of in-memory servers it is managing. Each namespace contains a table of all the consistent hashing ranges currently leased to each Owner, and every leased range is associated with a lease generation number. When a new Owner contacts the Manager, the Manager computes the new desired assignment of ranges to Owners, recalls leases on the ranges that are now destined for the new Owner, and grants new 60 second leases on these ranges to the new Owner as they become available (we show in Section 5.1.1 that 60 second leases are a good fit for our deployment environment). The removal of an Owner is similar. To support an incremental protocol for conveying changes to the assignment of leases, the Manager also maintains a change log for the lease table. This change log is periodically truncated to remove all entries older than 5 minutes. We describe the communication protocols between the Manager and the Lookup library and between the Manager and the Owner library in Sections 2.2 and 2.3 respectively.

The Centrifuge implementation also includes two features that are not yet present in the version running in production: state migration and adaptive load management. State migration refers to appropriately notifying nodes when a lease is transferred so that they can migrate the state along with the lease. Though load management is found in some network load balancers, we are not aware of any that support leasing or state migration. To support adaptive load management, Owners report their incoming request rate as their load. The adaptive load management algorithm uses these load measurements to add or subtract virtual nodes from any Owner that is more than 10% above or below the mean load

while maintaining a constant number of virtual nodes overall. For example, if one Owner is more than 10% above the mean load, a virtual node is subtracted from it and added to the least-loaded Owner, even if that least-loaded Owner is not 10% below the mean load. The particular load management algorithm is pluggable, allowing other policies to be implemented if Centrifuge requires them in the future.

2.2 Lookup Library

Each Lookup maintains a complete (though potentially stale) copy of the lease table: for every range, it knows both its lease generation number and the Owner node holding the lease. Due to the use of consistent hashing, this only requires about 200KB in the current Centrifuge deployment: 100 owners × 64 virtual nodes × 32B per range. This is a tiny amount for the servers linking in the Lookup libraries, and the small size is one reason the Lookup library caches the complete table rather than trying to only cache names that are frequently looked up.

Lookups use the lease table for two purposes. First, when the server linking in the Lookup library asks where to send a request on a given piece of state, the Lookup library reads the (potentially stale) answer out of its local copy of the lease table. Second, when the lease generation number on a range changes, the Lookup library signals a loss notification unless there is a flag set stating that the state was cleanly migrated to another Owner. At a high level, loss notifications allow servers linking in the Lookup library to republish data back in to the in-memory server pool; Sections 3 and 4 describe the use of loss notifications in more detail.

2.2.1 Lookup-Manager Protocol

To learn of incremental changes to the Manager's lease table, each Lookup contacts the Manager once every 30 seconds. An example of this is depicted in Figure 4. In this example, the Manager has just recorded a change, noted as LSN (log sequence number) 3, into its change log. This change split the range [1-9] between the Owners B and C, and the lease generation numbers (LGNs for short) have been modified as well. The Lookup contacts the Manager with LSN 2, indicating that it does not know of this change, and the Manager sends the change over. The Lookup then applies these changes to its copy of the lease table. If the Lookup sends over a sufficiently old LSN, and the Manager has truncated its log of lease table changes such that it no longer remembers this old LSN, the Manager replies with a snapshot of the current lease table. The Manager also sends over a snapshot of the entire lease table whenever it is more efficient than sending over the complete change list (in practice, we only observe this behavior when the

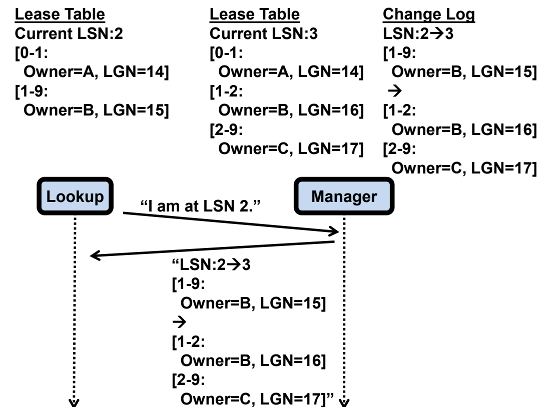


Figure 4: *Example of the protocol between the Lookup and the Manager.*

system is being brought online and many Owners are rapidly joining). Because the total size of the lease table is small, this limits the amount of additional data that the manager needs to send to Lookups when the system experiences rapid changes in Owner membership.

2.3 Owner Library

Each Owner only knows about the ranges that are currently leased to it. Owners send a message requesting and renewing leases every 15 seconds to the Manager. Because Manager leases are for 60 seconds, 3 consecutive lease requests have to be lost before a lease will spuriously expire. A lease request signals Owner liveness and specifies the leases where the Owner wants renewals. The Manager sends back a response containing all the ranges it is renewing and all the new ranges it has decided to grant to the Owner. Grants are distinguished from renewals so that if an Owner restarts, it will not accept an extension on a lease it previously owned. For example, if a just restarted Owner receives a renewal on a lease "X", it refuses the renewal, and the Manager learns that the lease is free. This causes the Manager to issue a new grant on the range, thus triggering a change in the lease generation number. This change in lease generation number ensures that the Manager's log of lease changes reflects any Owner crashes, thus guaranteeing that Lookups will appropriately trigger loss notifications.

Every message from the Manager contains the complete set of ranges where the Owner should now hold a lease. Although we considered an incremental protocol that sent only changes, we found that sending the complete set of ranges made the development and debugging of the lease protocol significantly easier. For example, we did not have to reconstruct a long series of message exchanges from the Manager log file to piece together how the Owner or Manager had gotten into a bad state. Instead, because each message had the complete set of

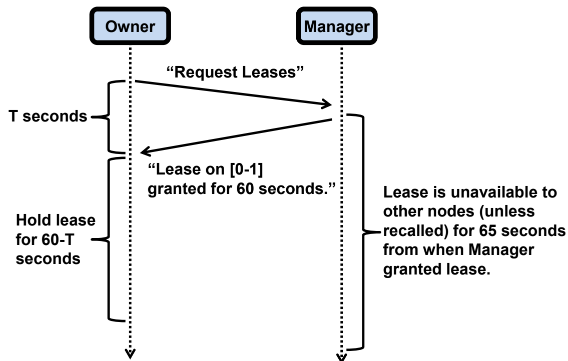


Figure 5: *How the lease protocol between the Owner and the Manager guarantees the safety property that at most one Owner holds the lease at any given point in time assuming only clock rate synchronization.*

leased ranges, we could simply look at the previous message and the current message to see if the implementation was generating the correct messages. Furthermore, because of the use of consistent hashing, all messages were still quite small: there are 64 leased ranges per Owner (one per virtual node) and each range is represented using 32B, which adds up to only 2KB per lease message.

2.3.1 Dealing with Clocks

Even after including the complete set of ranges in every lease message, there were still two subtle issues in the lease protocol. The first subtlety is guaranteeing the lease safety property: each key is owned by at most one Owner at any given point in time. For Centrifuge, we assume clock *rate* synchronization, but not clock synchronization. In particular, we assume that the Manager's clock advances by no more than 65 seconds in the time it takes the Owner's clock to advance by 60 seconds. This assumption allows Centrifuge to use the technique depicted in Figure 5, and previously described by Liskov [20]. The Owner is guaranteed to believe it holds the lease for a subset of the time that the Manager makes the lease unavailable to others because: (1) the Owner starts holding the lease only after receiving a message from the Manager, and (2) the Owner's 60-second timer starts before the Manager's 65-second timer, and 60 seconds on the Owner's clock is assumed to take less time than 65 seconds on the Manager's clock.

2.3.2 Dealing with Message Races

The second subtlety is dealing with message races. We explain the benefits of not having to reason about message races using an example involving lease recalls. Lease recalls improve the ability of the Manager to quickly make leases available to new Owners when they join the system – new leases can be handed out after a single message round trip instead of waiting up to 60

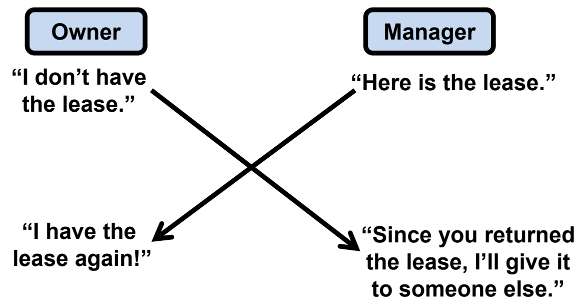


Figure 6: *Without care, message races can lead to violating the lease safety property. Centrifuge prevents this by including sequence numbers in the lease messages between the Manager and the Owner, and using the sequence numbers to filter out such message races.*

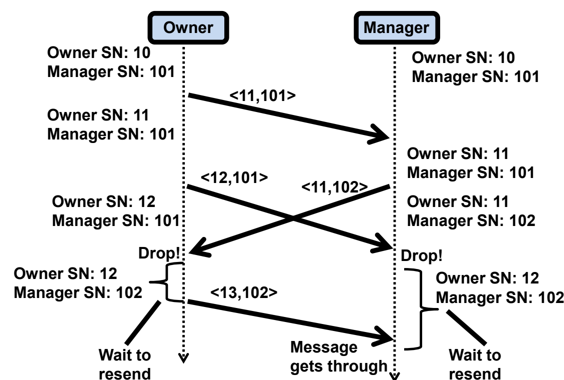


Figure 7: *The Manager and Owner use sequence numbers to filter out messages races. When a message race occurs, they wait for a random backoff, and then resend.*

seconds for the earlier leases to expire. However, lease recalls introduce the problem of lease recall acknowledgments and lease grants passing in mid-flight. This problem is depicted in Figure 6: When an Owner receives a lease recall request, it drops the lease, and then sends a message acknowledging the lease recall to the Manager. If the Manager has since changed its mind and sent out a new lease grant, the Manager needs some way to know that it is not safe to act on the lease recall acknowledgment for the earlier lease grant.

We solved this problem by implementing a protocol that hides message races from the program logic at the Manager and Owners. This protocol adds two sequence numbers to all lease messages, as depicted in Figure 7. The sender of a message includes both its own sequence number and the most recent sequence number it heard from the remote node. When the Manager receives a message from the Owner that does not contain the Manager's most recent sequence number, the Manager knows that the Owner sent this incoming message before the

Owner processed the previous message from the Manager, and the Manager drops the racing message from the Owner. The Owner does likewise. This prevents the kind of message race depicted in Figure 6. After either party drops a racing message, it waits for a random backoff, and then resends its message. Forward progress resumes when one node's message is received on the other side before its counterpart initiates its resend, as depicted in the Figure.

After the Manager receives the new message from the Owner, the Manager's own state most likely changes. The Manager may send a new message to the Owner, but the earlier racing message from the Manager to the Owner is permanently discarded. This is because there is no guarantee that the previous message to the Owner is still valid, e.g., the Manager may no longer want to grant a lease to the Owner. Finally, the protocol also includes a session nonce (not shown). This prevents an Owner from sending a message, crashing and re-establishing a connection with new sequence numbers, and then having the previously sent message be received and interpreted out of context.

2.4 Scalability

Centrifuge is designed to work within a datacenter management paradigm where the incremental unit of capacity is a cluster of a thousand or fewer machines. The current services using Centrifuge are all part of the Live Mesh application, which does follow this paradigm; it can be deployed into some number of clusters, and each individual cluster is presumed to have good internal network connectivity (e.g., no intra-cluster communication crosses a WAN connection). The use of clusters determines our scalability goals for Centrifuge – it must be able to support all the machines within a single cluster. Also, this level of scalability is sufficient to meet Centrifuge's goal of replacing internal network load balancers, which in this management paradigm are never shared across clusters. As we show in Section 5, Centrifuge scales well beyond this point, and thus we did not investigate further optimizing our implementation.

2.5 Loss Notification Rationale

One alternative to our design for loss notifications is to replicate the in-memory state across multiple Owner nodes. The primary benefit of this alternative design is that applications could obtain higher availability during node failures because there would be no need to wait for clients to republish data lost when nodes fail or reboot. However, the downsides would be significant. First, because replication cannot handle widespread or correlated failures, there is no benefit in terms of simplifying applications: the loss notification mechanism is still needed. Second, there is the additional cost of the RAM needed to hold multiple copies of the state at different nodes. Given the infrequency of node failures and reboots in the datacenter, it seems unlikely that the benefits of replication outweigh the significant costs of holding multiple copies in RAM. Finally, the implementation would be significantly more complex both in terms of the manager logic around leasing and partitioning as well as the Owner logic around performing the operations. This would add the requirement that the Owner actions become deterministic because Centrifuge has no notion of what actions the application is performing at the Owner nodes.

```
// Lookup API
URL  Lookup(Key key)
void LossNotificationUpcall(KeyRange[] lost)

// Owner API
bool CheckLeaseNow(Key key, out LeaseNum leaseNum)
bool CheckLeaseContinuous(Key key, LeaseNum leaseNum)
void OwnershipChangeUpcall(KeyRange[] grant,
  KeyRange[] revoke)
```

Figure 8: *The Centrifuge API divided into its Lookup and Owner parts. We omit asynchronous versions of the calls and calls related to dynamic load balancing and state migration. Upcalls are given as arguments to the relevant constructors.*

3 API

A simplified version of the Centrifuge API is shown in Figure 8. The API is divided into the calls exported by the Lookup library and the calls exported by the Owner library. We explain this API using the Publish-subscribe service, shown in Figure 9, as our running example.

3.1 Lookup

The servers that wish to make use of the Publish-subscribe service must link in the Lookup library. When the server in the example wishes to send a message subscribing its URL to a particular topic (in the example, the message is "Subscribe(1, http://A)") the server calls Lookup("1") and gets back the URL for the Publish-subscribe server responsible for this subscription list.

The semantics of Lookup() are that it returns *hints*. If it returns a stale address (e.g., the address of a Publish-subscribe server that is no longer responsible for this subscription list), the staleness will be caught (and the request rejected) at the Publish-subscribe server using the Owner API. Because the Manager rapidly propagates updated versions of the lease map to the Lookup library, callers should retry after a short backoff on such rejected requests. When the system is quiesced (i.e., no servers are joining or leaving the system), all calls to Lookup() return the correct address.

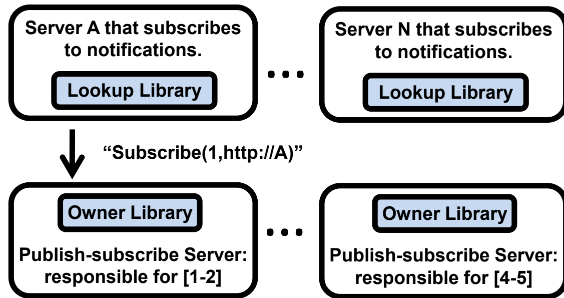


Figure 9: *We explain the Centrifuge API using the Publish-subscribe service's Subscribe() operation.*

If a node crashes, the servers linking in the Lookup library may wish to learn of this crash so they can proactively republish the data that was lost. In the example of Figure 9, Server A can respond to a loss notification by re-sending its earlier subscribe message. As mentioned in Section 2.2, the Manager enables this by assigning new lease generation numbers to all the ranges held by the crashed node (even if they are assigned back to the crashed node after it has recovered). All the Lookup libraries learn of the lease generation number changes from the Manager, and they then signal a LossNotificationUpcall() on the appropriate ranges.

3.2 Owner

The Owner part of the API allows a Publish-subscribe server to perform an operation guarded by a lease. Because the Manager may not have assigned any particular lease to this Owner, the Owner must be prepared to fail this operation if it does not have the lease; there is no API call that forces the Manager to give the lease for a given key to the Owner.

The code for the Subscribe operation at the Publish-subscribe servers is shown in Figure 10. This code uses leases to guarantee that requests for a given subscription list are only being served by a single node at a time, and that this node is not operating on a stale subscription list (i.e., a subscription list from some earlier time that the node owned the lease, but where the node has not held the lease continuously).

The general pattern to using the Owner API is shown at the top of Figure 10. When a request arrives at a Publish-subscribe server, step (1) is to call CheckLeaseNow() to check whether it should serve the request. If this call succeeds, step (2) is to validate any previously stored state by comparing the current lease number with the lease number from when the state was last modified. If the Publish-subscribe server has held the lease continuously, this old lease number will equal the Owner library's current lease number, and the check will succeed. If the Publish-subscribe server has not held the lease con-

```
// General pattern:
// (1) Check that request has arrived at correct node
// (2) Check that existing state is not stale;
//     discard state that turns out to be stale
// (3) Perform arbitrary operation on this state;
//     store lease number with any created state
// (4) Check that lease has been continuously held;
//     if so, return result

bool Subscribe(key, address) {
  // (1) Check that this is the correct node
  ok = CheckLeaseNow(key, out currentLeaseNum);
  if (!ok) return false;
  // (2) Check that existing state is not stale;
  //     discard state that turns out to be stale
  storedLeaseNum = this.leases[key];
  if (currentLeaseNum != storedLeaseNum)
     this.subscriptionLists[key] = EmptyList();
     // in this app, okay to reset to EmptyList()
     // if prior state was stale
  // (3) Perform arbitrary operation on this state;
  //     store lease number with any created state
  // in this case, simply add subscription and
  // store lease number
  this.subscriptionLists[key].Add(address);
  this.leases[key] = currentLeaseNum;
  // (4) Check that lease has been continuously held;
  //     if so, return result
  if (!CheckLeaseContinuous(key, currentLeaseNum))
    return false;
  return true;
}
```

Figure 10: *How the Subscribe() operation uses the Owner API.*

tinuously, the old subscription list may be stale (i.e., it may not reflect all operations executed on the list), and therefore the old subscription list should be discarded.

Step (3) is to perform an arbitrary operation on this state, and then to store any state modifications along with the lease number. In the case of Subscribe(), the operation is adding the address to the list of subscriptions. The newly stored lease number will be checked in future calls to Subscribe().

Step (4) is to return a result to the caller, or more generally, to send a result to some other node. If the lease was lost while the operation was in progress, the caller can simply return false and does not need to proactively clean up the state that is now invalid. Future calls to Subscribe() will either fail at CheckLeaseNow() or will clear the invalid state when they find the stored lease number to be less than the current lease number.

Note that throughout this sequence of steps, there is no point where the Publish-subscribe server requests a lease on a given item. Instead, the Publish-subscribe server simply checks what leases it has been assigned. As mentioned in Section 1, this is a departure from standard lease manager APIs, and the novel Centrifuge API is critical to allowing Centrifuge to provide the benefits of fine-grained leasing, while only granting leases on a small number of ranges.

This example has focused on the use of leases within a single service, but lease numbers can also be used in

communication between services. For example, a server linking in the Owner library can include a lease number in a request to another service. The other service can then guard against stale messages by only processing a request if the included lease number is greater than or equal to any previously seen lease number. This technique has also been described in previous work on lease managers; for example, it is one of the patterns for using Chubby's lease numbers, which are called Chubby sequencers [3].

The last part of the Owner API is the OwnershipChangeUpcall(). This upcall may be used to initialize data structures when some new range of the key space has been granted, or to garbage collect state associated with a range of the key space that has been revoked. Because of thread scheduling and other effects, this upcall may be delivered some short time after the lease change occurs.

4 How Centrifuge Supports the Live Mesh Services

Centrifuge is used by five component services that are part of the Live Mesh application [22]. All these component services were built by other developers. The Live Mesh application provides a number of features, including file sharing and synchronization across devices, notifications of activity on these shared files, a virtual desktop that is hosted in the datacenter and allows manipulating these files through a web browser, and connectivity to remote devices (including NAT traversal). In the remainder of this Section, we describe how three of the Live Mesh services use Centrifuge to enable a particular scenario. We then explain how these services were simplified by leveraging the lease semantics of Centrifuge. Finally, we discuss some common characteristics of the Live Mesh services and how these characteristics influenced the design of Centrifuge.

4.1 Example of Three Live Mesh Services

The particular scenario we focus on is one where a user has two PCs, one at home and one at work, and both are running the Live Mesh client software. At some point the user wants to connect directly from their work PC to their home PC so as to retrieve an important file, but the home PC has just been given a new IP address by the user's ISP. To enable the work PC to find the home PC, the home PC needs to publish its new IP address into the datacenter, and the work PC then needs to learn of the change.

Figure 11 depicts how the component services enable this scenario. First, the home PC sends its new IP address in a "publish new IP" request to a Frontend server. The Frontend server calls Lookup() using the home PC's "DeviceID" as the key and routes this request to the in-memory Device Connectivity server tracking the IP address for this home PC. The Device Connectivity servers use the Owner library with DeviceIDs as the keys, and they store the IP address for a device under each key.

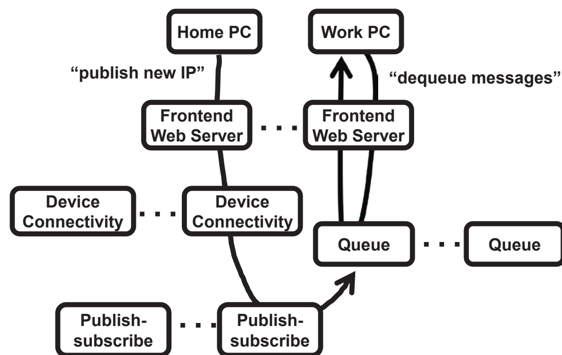


Figure 11: *How three Centrifuge-based services within the Live Mesh application cooperate to enable a work PC to connect to a home PC that has just acquired a new IP address.*

After updating the IP address, the Device Connectivity server sends a message with this new IP address to the in-memory Publish-subscribe server tracking subscriptions to the home PC's device connectivity status; the Device Connectivity server uses the home PC's DeviceID as the key for calling Lookup(). The Publish-subscribe servers also use the Owner library with DeviceIDs as the keys, but under each key, they store a subscription list of all devices that want to receive connectivity status updates from the home PC. Finally, the Publish-subscribe server sends out a message containing the device connectivity update to each subscribed client device.

Because some of the subscribed client devices (such as the user's work PC) may be offline, each message is routed to an in-memory Queue server. If a subscribed client device is online, it maintains a persistent connection to the appropriate Queue server, and the messages are immediately pushed over the persistent connection. If a subscribed client device is offline and later comes online (the case depicted for the work PC in Figure 11), it sends a "dequeue messages" request. This request arrives at a Frontend and is routed to the appropriate Queue server, which responds with the message containing the new IP address. The Publish-subscribe service is the source for all the messages sent through the Queue servers; all these messages describe changes for datacenter objects that the client had previously subscribed to, such as the device connectivity status in this example.

The Queue servers also use the Owner library with DeviceIDs as the keys, and they store a message queue under each key. The Queue servers also link in the Lookup library so that they can receive a LossNotifi-

cationUpcall() if any Device Connectivity or Publish-subscribe server crashes. If it receives a loss notification, the Queue server puts the notification into the queue for any client device that had state stored on the server that crashed, allowing the client device to quickly learn that it needs to re-publish its IP address, poll to re-read other client devices' IP addresses, and/or re-subscribe to learn of future IP address changes.

Because these servers are all co-located within the same datacenter, we assume that there are no persistent intransitive connectivity failures between these machines, and therefore if the Device Connectivity, Publish-subscribe and Queue servers can all talk to the Centrifuge Manager and renew their leases, they will also be able to eventually send messages to each other. If an initial message send fails (perhaps because the Lookup library had slightly stale information), the server should simply retry after a short period of time.

Because of the in-memory nature of these server pools, a crash always results in the crashed server losing its copy of the data. All the services currently using Centrifuge rely on Centrifuge's loss notifications for two purposes: to trigger the client to re-create any lost data within the in-memory server pool and to poll all datacenter objects where a relevant change message may have been lost. The client already knows the small set of datacenter objects it needs to poll because it had previously subscribed to them using the Publish-subscribe service, and the Queue service is only used to deliver change messages from these subscriptions and loss notifications. For these services, relying on the client for state re-creation is sufficient because the state is only useful if the client is online. To pick one example, if a client is offline, its last published IP address is irrelevant. Furthermore, the reliance on clients to recreate the state allows the service to forgo the expense of storing redundant copies of this state on disk within the datacenter (although Centrifuge itself is also compatible with recovery from datacenter storage). Lastly, although the power of leases to simplify distributed storage systems is well-established [40, 13, 25, 3], the next several sections elaborate on how leases can also simplify services that are not tightly integrated with storage (the Live Mesh services).

4.2 Simplifications to Device Connectivity Service

The main simplification from using leases in the Device Connectivity service is that all updates logically occur at a single server. This means that there are never multiple documents on multiple servers containing the IP address about a single device. Because there are never multiple documents, there is no need to write application-specific logic about reconciling the documents, and there is no need to add application-specific metadata simply to aid in document reconciliation (e.g., the time at which the IP address was updated). Although it may be feasible to design a good reconciliation heuristic for device IP addresses, this is yet another tax on the developer. Additionally, the Device Connectivity service is also used to store other kinds of data besides IP addresses, and reconciliation becomes more difficult as the data becomes more complex. The use of leases allows the developer to avoid writing the application-specific reconciliation routines for IP addresses and for all these other kinds of data.

4.3 Simplifications to Queue Service

In the Queue service, the main simplification from using leases is that the service can provide a strong guarantee to all its callers: once a message has been successfully enqueued, either the client will receive it, or the client will know it has lost some messages and must appropriately poll. This allows the Publish-subscribe server to consider itself "finished" with a message once it has been given to the Queue service; the Publish-subscribe server does not have to deal with the possibility that the client device neither received the message nor even learned that it lost a message. Such silent message loss could be particularly frustrating; for example, the home PC could publish its new IP address without the work PC ever learning of the change, leading to a long-lived connectivity failure. The Queue service's guarantee prevents this problem.

To provide this guarantee, the Queue service relies on there being at most one copy of a queue at any given point in time. In contrast, if there were multiple copies of a queue, the Publish-subscribe server might believe it successfully delivered a message, while the client device only ever connected to another copy of the queue to read its messages. While it may be feasible to build a protocol that addresses this issue in alternative ways (perhaps using counters or version vectors), the Centrifuge lease mechanism avoided the need for such an additional protocol.

4.4 Simplifications to Publish-subscribe Service

The simplifications from using leases in the Publish-subscribe service are similar to the simplifications in the Queue service. The Publish-subscribe service uses leases to provide the following strong guarantee to its callers: once a message has been accepted by the Publish-subscribe service, each subscriber will either receive the message or know that they missed some messages. The details of how leases enable this guarantee in the Publish-subscribe service are essentially the same as those described in Section 4.3 for the Queue service.

4.5 Live Mesh Service Characteristics

We now briefly discuss how the common characteristics of the Live Mesh services influenced the design choices we made for Centrifuge.

All of the Live Mesh services that use Centrifuge store a relatively large number of small objects in memory on each server, and the operations performed on these objects are relatively lightweight. As a result, we designed Centrifuge to rely on statistical multiplexing over these many small objects. This allows Centrifuge to operate on variable length ranges of the keyspace, rather than providing a directory service optimized for moving individual items. None of the current Live Mesh services need to support data items where the processing load on an individual object is near the capacity of an entire server. Services with such a workload would find statistical multiplexing much less effective. To enable such workloads, one might consider modifications to Centrifuge such as: a load balancing algorithm with different dynamics; a different partitioning algorithm; and support for replicating objects across multiple owners.

Another aspect of the Centrifuge design motivated by the needs of Live Mesh services is the capability of recovering state from clients, rather than just from storage servers within the datacenter. Most of the state stored at the Owner nodes for the Live Mesh services is cached data where the original copy is known at the client (e.g., the client's IP address for the Device Connectivity service). The need to recover state from clients led to our design for loss notifications, where these notifications are delivered to the Lookup nodes, rather than building recovery functionality directly into the Owner nodes.

Finally, as we will show in Section 5, reboots and machine failures in the Live Mesh production environment are infrequent. This observation, combined with the fact that large quantities of RAM are still reasonably expensive, led to our decision to recover state from the clients rather than replicating state across multiple Owner nodes.

5 Evaluation

As we mentioned in the Introduction, Centrifuge has been deployed in production as part of Microsoft's Live Mesh application since April of 2008 [22]. In Section 5.1 we examine the behavior of Centrifuge in production over an interval of 2.5 months, stretching from early December 2008 to early March 2009. During this time, there were approximately 130 Centrifuge Owners and approximately 1,000 Centrifuge Lookups. Previewing our results, we find that Centrifuge easily scaled to meet the demands of this deployment. In Section 5.2, we use a testbed to examine Centrifuge's ability to scale beyond the current production environment.

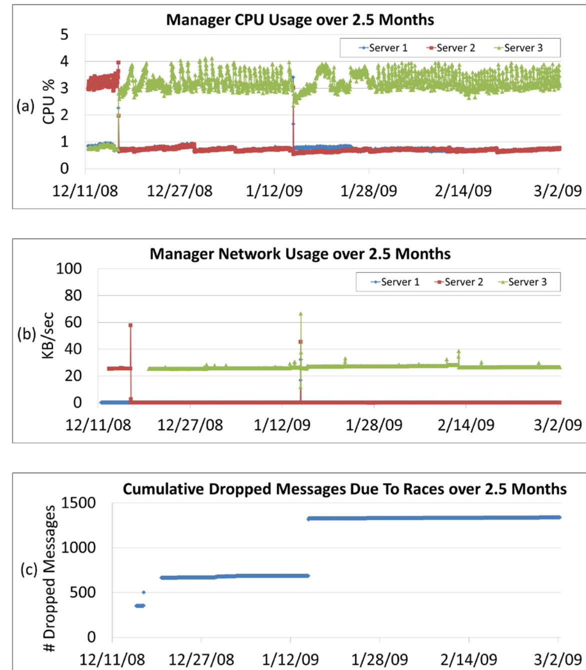


Figure 12: *CPU and network load measurements from the leader and standbys in the Centrifuge Manager service deployed in production over 2.5 months, as well as messages dropped due to races.*

5.1 Production Environment

Because the Centrifuge Manager is the scalability bottleneck in our system, we focus our observation on the behavior of this component. We first examine the steady-state behavior of the Manager over 2.5 months, and then focus on the behavior of the Manager during the hours surrounding the rollout of a security patch.

5.1.1 Steady State Behavior

Figures 12(a) and (b) show measurements from each leader and standby server at the granularity of an hour from 12/11/2008 to 3/2/2009. We observe that both the CPU and network utilization is very low on all these servers, though it is slightly higher on the current leader at any given point in time, and there are bursts of network utilization when a standby takes over as the new leader, as on 12/16/2008 and 1/15/2009. The low steady-state CPU and network utilization we observe provides evidence that our implementation easily meets our current scalability needs.

In both cases where the leader changed, the relevant administrative logs show that a security patch was rolled out, requiring restarts for all servers in the cluster. The patch rollout on 12/16/2008 at 06:30 led to the leader changing from Server 2 to Server 3, while the patch rollout on 1/15/2009 at 21:20 led to the servers rotating the leader role in quick succession, with Server 3 resuming

the leader role at the end of this event. Because the second security patch rollout led to multiple changes in the leader, we examine the dynamics of this rollout in more detail in Section 5.1.2.

Because there were no security patch rollouts between 1/16/2009 and 3/2/2009, we examined this 1.5 month period to see how frequently Owners lost their leases due to crashing, network disconnect, or any other unplanned event. An Owner crash can make the portion of the key space assigned to that Owner unavailable until the Owner's lease has expired, and we use 60-second leases (as mentioned in Section 2.1.2) because we expect unplanned Owner failures to be quite rare. We observed a total of 10 lease losses from the 130 Owners over the entire 1.5 months. This corresponds to each individual Owner having a mean time to unplanned lease loss (i.e., not due to action by the system administrator) of 19.5 months. This validates our expectation that unplanned Owner failures are quite rare. Finally, the number of Owners returned to 130 in less than 10 minutes following 7 of the lease losses, and in about an hour for the other 3 lease losses; this shows that Owner recovery or replacement by the cluster management system [16] is reasonably rapid.

We also examine one aspect of the Manager-Owner lease protocol in detail over this same 2.5 month interval. As described in Section 2.3.2, our lease protocol incorporates a simple mechanism for preventing message races from compromising the lease safety invariant during lease recalls: detecting such races and dropping the messages. We do not expect message races to be common, but if they were, repeated message drops might lead to one of the Owners losing its lease. This motivates us to examine the number of messages dropped due to race detection, as shown in Figure 12(c). We first observe that message races do occur in bursts when a new standby takes the leader role. However, during the 1.5 months from 1/16/2009 to 3/2/2009 where Server 3 continuously held the leader role, only 12 messages were dropped. This observation validates our expectation that message races are very rare in steady state.

5.1.2 Rollout of Security Patch

Figure 13 examines a 2.5 hour window at a 30 second granularity where all 3 standby servers rotated through the leader role. Servers 1 and 3 were restarted at approximately 21:20. Server 2 then took over as the leader, leading to slightly higher CPU utilization and significantly higher network utilization at this server. Network utilization peaks at almost 500 KB/sec, significantly more than the steady state of around 10 KB/sec. At approximately 21:45, Server 2 was similarly restarted, leading to Server 1 taking over as the leader. Finally, at approximately 22:20, Server 3 took the leader role from Server

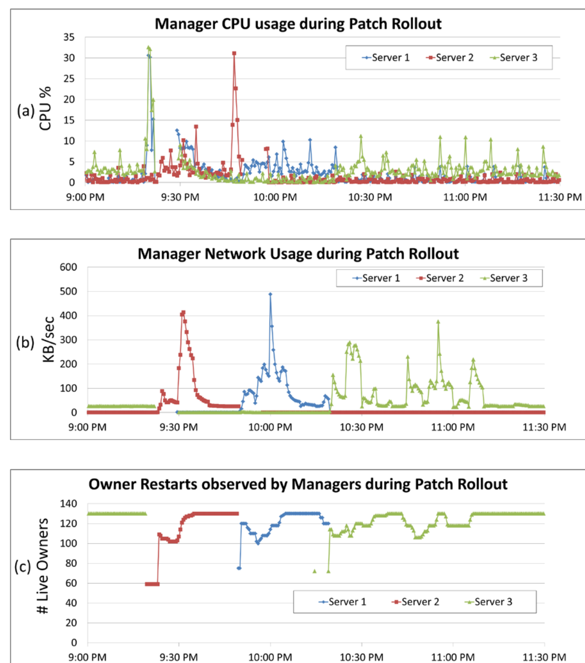


Figure 13: *Measurements from the evening of 1/15/2009 for the leader and standby servers in the Manager service deployed in production, capturing CPU, network load, and Owner restarts.*

2. We do not know why this last change in leader role occurred, as Server 2 was not restarted at this point. In all cases, restarts were preceded by a spike in CPU utilization, likely due to applying the patch.

Figure 13(c) shows the number of live Owners seen by each leader. The number of live Owners dips at approximately the same time as the change in leader, reflecting how the patch is applied to one group of servers, and then applied to another group of servers after a modest interval.

Figure 13(b) also shows that network utilization continued to experience bursts well after a new standby had taken the leader role. From additionally examining the Manager logs, we find that 76% of this network traffic is due to the Lookup libraries, likely because after restarting they need to get the entire lease table from the leader.

Figure 13 show that even during a period of abnormally high churn in both Owners, Lookups and Managers, the observed load at the leader and standbys in the Manager service is small. This offers further evidence that the Centrifuge implementation meets its current scalability demands.

Finally, we investigated one of the Owners to double check that the stability we observe in the Manager was also reflected in the Owner API success rate. We arbitrarily chose the approximately 5-day time period 1/8/2009 22:00 to 1/13/2009 17:00, a period when the Manager service did not observe any churn. During this time pe-

Role	# Servers	Instances/ Server	Total Instances
Manager Paxos group	5	1	5
Manager leader/standby	3	1	3
Device Connectivity	8	25	200
Frontend	24	84	2,016

Table 1: *Centrifuge testbed configuration.*

riod, this particular Owner experienced 0 failed calls to CheckLeaseNow() and CheckLeaseContinuous() out of over 53 million invocations. This is consistent with the intuition that the stability observed at the Manager results in calls to the Owner API always succeeding.

5.2 Testbed

In this subsection, we evaluate Centrifuge's ability to scale beyond the current production deployment. In particular, we use more Lookups and Owners than are present in the production setting, and we evaluate the Manager load when these Lookups and Owners are restarted more rapidly than in a production patch rollout. Previewing our results, we find that the Manager easily scales to this larger number of Owners and Lookups and this more rapid rate of restarts.

Our testbed consists of 40 servers, each running a 2.26 GHz Core2 Duo processor with 4GB RAM and the Windows Server 2008 SP2 operating system. Table 1 shows our testbed configuration. The approximately 10:1 ratio between Device Connectivity servers (Owners) and Frontends (Lookups) was chosen based on the ratio deployed in production. We made minor modifications to the performance counter implementation on the Device Connectivity Servers and Frontends in order to run this many instances on each server.

To examine the ability of Centrifuge to support a more rapid patch rollout rate across this larger number of Owners and Lookups, we conducted two separate experiments. In the first experiment, we restarted all the Owners over the course of 32 minutes, and in the second experiment, we restarted all the Lookups over the same interval. Compared to the production patch rollout of Section 5.1.2, this is restarting approximately twice as many nodes (2,200) in half as much time (1 hour). In both experiments, we measure CPU and network usage at the leader in the Manager service. Figures 14 shows the results: even when the Owners were restarting, the leader CPU averaged only light utilization, and network usage only went up to 5 MB/sec. Based on this, we conclude that the Centrifuge implementation supports the current deployment by a comfortable margin.

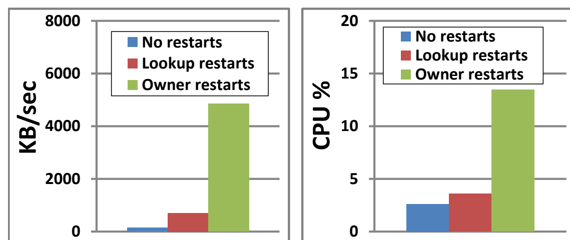


Figure 14: *The CPU and network load on the Manager leader under rapid restarts for a large number of Owners and Lookups.*

6 Related Work

Centrifuge integrates lease management and partitioning into a system that makes it easier to build certain datacenter services. Accordingly, we divide our discussion of related work into lease managers, partitioning systems, and other software infrastructure for building datacenter services.

6.1 Lease Managers

The technique of using leases to provide consistency dates back over two decades to Gray and Cheriton's work on the V file system [13]. In this section we survey the three leasing systems most closely related to Centrifuge: Frangipani's lease manager [40] because of its approach to scalability; Chubby [3] because of its use to support datacenter applications; and Volume Leases [42, 41] because of how it grants leases on many objects at a time.

Frangipani implements a scalable lease management service by running multiple lease managers, and having each of these lease managers handle a different subset of the total set of leases. Centrifuge scales using a very different technique: exposing a novel API so that lease recipients receive all or none of the leases within a range. Centrifuge's design allows a pool of in-memory servers needing a large number of leases to be supported by a single lease manager. However, the techniques in Centrifuge and Frangipani are composable: one could imagine applying Frangipani's technique to further scale Centrifuge by creating multiple Centrifuge managers and having each of them handle a different subset of the lease namespace.

Like Centrifuge, Chubby implements a lease manager that funnels all clients through a single machine and relies on a Paxos layer for high availability. Chubby is designed to be used primarily for leader election in a datacenter, and in practice it typically maintains around a thousand locks to support tens of thousands of clients [3]. In contrast, Centrifuge directly provides both partitioning and leases on ranges from a partitioned namespace, enabling Centrifuge to replace internal network load balancers.

Supporting Centrifuge's functionality using Chubby would require modifying Chubby to incorporate elements from Centrifuge, including the partitioning functionality. Alternatively, one could imagine extracting the partitioning functionality out of an existing system such as BigTable [5], modifying it to support Centrifuge's partitioning algorithm, and then deploying this additional service alongside Chubby.

Volume Leases [42, 41] is a protocol for granting leases on groups of objects, such as all the files and directories in a file system volume or all the web pages served by a single server. Centrifuge differs from Volume Leases in at least two major ways. First, Centrifuge can dynamically create new object groupings by sending out new ranges, while none of the work on Volume Leases investigated dynamically creating volumes. In Centrifuge, dynamically splitting and merging ranges underlies the majority of the logic around partitioning (the policy for modifying ranges) and leasing (the mechanism for conveying the modified ranges). Secondly, the work on Volume Leases did not include high-availability for the lease manager; Volume Leases are designed to be the cache coherency protocol for a system that might or might not incorporate high availability, not a stand alone lease manager.

6.2 Partitioning Systems

The three most closely related pieces of prior research in partitioning are the partitioning subsystem of BigTable [5], network load balancers [9, 28], and other software partitioning systems such as DHTs [32, 38, 31, 30] and the LARD system [29]. We already compared Centrifuge to the partitioning subsystem of BigTable as part of our comparison to Chubby.

The main contrast between Centrifuge and network load balancers is that network load balancers do not include lease management. Attempting to add lease management (and the requisite high availability) would constitute a major addition to these systems, possibly mirroring the work done to build Centrifuge.

Centrifuge implements partitioning using client libraries, an approach previously taken by many DHTs and the LARD system. Compared to this prior work, the contribution of Centrifuge is demonstrating that combining such partitioning with lease management simplifies the development of datacenter services running on in-memory server pools. Though there has been a great deal of work on DHTs, we are not aware of any DHT-based system that explores this integration, most likely because DHTs often focus on scaling to billions of peers and providing leases on the DHT key space is viewed as incompatible with this scalability goal. In contrast, Centrifuge does incorporate a lease manager to provide a better programming model for in-memory server pools, and Centrifuge only aims to scale to hundreds of such servers.

6.3 Other Infrastructure for Datacenter Services

There is great interest within both industry and the research community in providing better infrastructure for datacenter services [2, 12, 15, 11]. Much of this interest has been directed at datacenter storage: BigTable [5], Dynamo [8], Sinfonia [1] and DDS [14] form a representative sample. Centrifuge is designed to support in-memory server pools. These server pools may want to leverage such a datacenter storage system to support reloading data in the event of a crash, but they also benefit from partitioning and leasing within the in-memory server pool itself.

Distributed caching has been widely studied, often in the context of remote file systems [7, 10, 19, 18, 24, 6], and it is commonly used today in datacenter applications (e.g., memcached [26]). Centrifuge differs from such systems in that Centrifuge is not a cache; Centrifuge is infrastructure that makes it easier to build other services that run on pools of in-memory servers. Because Centrifuge is a lower-level component than a distributed cache, it can serve a wider class of applications. For example, the Publish-subscribe Service described in Section 4 devotes significant logic to state management (e.g., deciding what state to expire, what state to lock, evaluating access control rules, etc.). This logic is service-specific, and it is not something that the service developer would want to abdicate to a generic distributed cache. In contrast, the Publish-subscribe Service can leverage Centrifuge because it only provides the lower-level services of leasing and partitioning.

7 Conclusion

Datacenter services are of enormous commercial importance. Centrifuge provides a better programming model for in-memory server pools, and other developers have validated this by using Centrifuge to build multiple component services within Microsoft's Live Mesh, a large-scale commercial cloud service. As datacenter services continue to increase in complexity, such improvements in programmability are increasingly vital.

Acknowledgements

We greatly appreciate the contributions to the Centrifuge design from Bruce Copeland, Abolade Gbadegesin, Alex Mallet, Tom Kleinpeter, and Mike Zintel. We particularly wish to thank Jeremy Dewey for his work implementing leader election and state reliability in Paxos, and Greg Prier for his work improving the quality of Centrifuge.

Thanks to Dennis Crain for helping us with the MSR shared computing cluster. Finally, we thank Marcos Aguilera, Dahlia Malkhi, Dave Maltz, Rodrigo Rodrigues (our shepherd), Stefan Saroiu, Alex Snoeren, and the anonymous reviewers for their insightful feedback on earlier drafts of this paper.

References

[1] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. In *SOSP '07: Proceedings of the Twenty-first ACM Symposium on Operating Systems Principles*, 2007.

[2] Amazon Web Services. http://aws.amazon.com.

[3] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI '06: Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006.

[4] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos Made Live: An Engineering Perspective. In *PODC '07: Proceedings of the Twenty-sixth ACM Symposium on Principles of Distributed Computing*, 2007.

[5] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI '06: Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006.

[6] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *SOSP '01: Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, 2001.

[7] Michael Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *OSDI '94: Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, 1994.

[8] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *SOSP '07: Proceedings of the Twenty-first ACM Symposium on Operating Systems Principles*, 2007.

[9] F5. http://www.f5.com.

[10] Michael J. Feeley, William E. Morgan, Frederic H. Pighin, Anna R. Karlin, Henry M. Levy, and Chandramohan A. Thekkath. Implementing Global Memory Management in a Workstation Cluster. In *SOSP '95: Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, 1995.

[11] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-Based Scalable Network Services. In *SOSP '97: Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, 1997.

[12] Google App Engine. http://appengine.google.com.

[13] Cary G. Gray and David R. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *SOSP '89: Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, 1989.

[14] Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *OSDI '00: Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, 2000.

[15] Steven D. Gribble, Matt Welsh, Rob von Behren, Eric A. Brewer, David Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R. H. Katz, Z. M. Mao, S. Ross, and B. Zhao. The Ninja Architecture for Robust Internet-Scale Systems and Services. *Computer Networks*, 35(4):473–497, 2001.

[16] Michael Isard. Autopilot: Automatic Data Center Management. *SIGOPS Operating Systems Review*, 41(2):60–67, 2007.

[17] Anne-Marie Kermarrec, Antony Rowstron, Marc Shapiro, and Peter Druschel. The IceCube approach to the reconciliation of divergent replicas. In *PODC '01: Proceedings of the Twentieth ACM Symposium on Principles of Distributed Computing*, 2001.

[18] Benjamin C. Ling and Armando Fox. The Case for a Session State Storage Layer. In *HOTOS IX: Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, 2003.

[19] Benjamin C. Ling, Emre Kiciman, and Armando Fox. Session State: Beyond Soft State. In *NSDI '04: Proceedings of the First Symposium on Networked Systems Design and Implementation*, 2004.

[20] Barbara Liskov. Practical Uses of Synchronized Clocks in Distributed Systems. *Distributed Computing*, 6(4):211–219, 1993.

[21] Live Meeting. http://office.microsoft.com/livemeeting.

[22] Live Mesh. http://www.mesh.com.

[23] Live Messenger. http://messenger.live.com.

[24] Qiong Luo, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, Honguk Woo, Bruce G. Lindsay, and Jeffrey F. Naughton. Middle-tier Database Caching for e-Business. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, 2002.

[25] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: Abstractions as the Foundation for Storage Infrastructure. In *OSDI'04: Proceedings of the 6th Symposium on Opearting Systems Design and Implementation*, 2004.

[26] Memcached. http://www.danga.com/memcached.

[27] Jeff Napper, Lorenzo Alvisi, and Harrick Vin. A Fault-Tolerant Java Virtual Machine. In *DSN 2003: Proceedings of the International Conference on Dependable Systems and Networks*, 2003.

[28] NetScaler. http://www.citrix.com/netscaler.

[29] Vivek S. Pai, Mohit Aron, Gaurov Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich Nahum. Locality-aware Request Distribution in Cluster-based Network Servers. In *ASPLOS-VIII: Proceedings of the Eighth International Conference on Architectural*

Support for Programming Languages and Operating Systems, 1998.

[30] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A Scalable Content-Addressable Network. In *SIGCOMM '01: Proceedings of ACM SIGCOMM*, 2001.

[31] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. In *USENIX '04: Proceedings of the USENIX Annual Technical Conference*, 2004.

[32] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms*, 2001.

[33] Yasushi Saito and Marc Shapiro. Optimistic Replication. *ACM Computing Surveys*, 37(1):42–81, 2005.

[34] Scaling Digg. http://highscalability.com/scaling-digg-and-other-web-applications.

[35] Scaling Facebook. http://www.facebook.com/note.php?note_id=39391378919.

[36] Scaling LinkedIn. http://hurvitz.org/blog/2008/06/linkedin-architecture.

[37] Joseph G. Slember and Priya Narasimhan. Static Analysis Meets Distributed Fault-Tolerance: Enabling State-Machine Replication with Nondeterminism. In *HOTDEP '06: Proceedings of the 2nd Workshop on Hot Topics in System Dependability*, 2006.

[38] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *SIGCOMM '01: Proceedings of ACM SIGCOMM*, 2001.

[39] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *SOSP '95: Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, 1995.

[40] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A Scalable Distributed File System. In *SOSP '97: Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, 1997.

[41] Jian Yin, Lorenzo Alvisi, Michael Dahlin, and Calvin Lin. Using Leases to Support Server-Driven Consistency in Large-Scale Systems. In *ICDCS '98: Proceedings of the 18th International Conference on Distributed Computing Systems*, 1998.

[42] Jian Yin, Lorenzo Alvisi, Michael Dahlin, and Calvin Lin. Volume Leases for Consistency in Large-scale Systems. *IEEE Transactions on Knowledge and Data Engineering Special Issue on Web Technologies*, 11(4):563–576, 1999.

Volley: Automated Data Placement for Geo-Distributed Cloud Services

Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman
Microsoft Research, {sagarwal, jdunagan, navendu, ssaroiu, alecw}@microsoft.com

Harbinder Bhogan
University of Toronto, hbhogan@cs.toronto.edu

Abstract: *As cloud services grow to span more and more globally distributed datacenters, there is an increasingly urgent need for automated mechanisms to place application data across these datacenters. This placement must deal with business constraints such as WAN bandwidth costs and datacenter capacity limits, while also minimizing user-perceived latency. The task of placement is further complicated by the issues of shared data, data inter-dependencies, application changes and user mobility. We document these challenges by analyzing month-long traces from Microsoft's Live Messenger and Live Mesh, two large-scale commercial cloud services.*

We present Volley, a system that addresses these challenges. Cloud services make use of Volley by submitting logs of datacenter requests. Volley analyzes the logs using an iterative optimization algorithm based on data access patterns and client locations, and outputs migration recommendations back to the cloud service.

To scale to the data volumes of cloud service logs, Volley is designed to work in SCOPE [5], a scalable MapReduce-style platform; this allows Volley to perform over 400 machine-hours worth of computation in less than a day. We evaluate Volley on the month-long Live Mesh trace, and we find that, compared to a state-of-the-art heuristic that places data closest to the primary IP address that accesses it, Volley simultaneously reduces datacenter capacity skew by over 2×, reduces inter-datacenter traffic by over 1.8× and reduces 75th percentile user-latency by over 30%.

1 Introduction

Cloud services continue to grow rapidly, with ever more functionality and ever more users around the globe. Because of this growth, major cloud service providers now use tens of geographically dispersed datacenters, and they continue to build more [10]. A major unmet challenge in leveraging these datacenters is automatically placing user data and other dynamic application data, so that a single cloud application can serve each of its users from the best datacenter for that user.

At first glance, the problem may sound simple: determine the user's location, and migrate user data to the closest datacenter. However, this simple heuristic ignores two major sources of cost to datacenter operators: WAN bandwidth between datacenters, and over-provisioning datacenter capacity to tolerate highly skewed datacenter utilization. In this paper, we show that a more sophisticated approach can both dramatically reduce these costs and still further reduce user latency. The more sophisticated approach is motivated by the following trends in modern cloud services:

Shared Data: Communication and collaboration are increasingly important to modern applications. This trend is evident in new business productivity software, such as Google Docs [16] and Microsoft Office Online [32], as well as social networking applications such as Facebook [12], LinkedIn [26], and Twitter [43]. These applications have in common that many reads and writes are made to shared data, such as a user's Facebook wall, and the user experience is degraded if updates to shared data are not quickly reflected to other clients. These reads and writes are made by groups of users who need to collaborate but who may be scattered worldwide, making it challenging to place and migrate the data for good performance.

Data Inter-dependencies: The task of placing shared data is made significantly harder by inter-dependencies between data. For example, updating the wall for a Facebook user may trigger updating the data items that hold the RSS feeds of multiple other Facebook users. These connections between data items form a communication graph that represents increasingly rich applications. However, the connections fundamentally transform the problem's mathematics: in addition to connections between clients and their data, there are connections in the communication graph in-between data items. This motivates algorithms that can operate on these more general graph structures.

Application Changes: Cloud service providers want to release new versions of their applications with ever greater frequency [35]. These new application features

can significantly change the patterns of data sharing and data inter-dependencies, as when Facebook released its instant messaging feature.

Reaching Datacenter Capacity Limits: The rush in industry to build additional datacenters is motivated in part by reaching the capacity constraints of individual datacenters as new users are added [10]. This in turn requires automatic mechanisms to rapidly migrate application data to new datacenters to take advantage of their capacity.

User Mobility: Users travel more than ever today [15]. To provide the same rapid response regardless of a user's location, cloud services should quickly migrate data when the migration cost is sufficiently inexpensive.

In this paper we present Volley, a system for automatic data placement across geo-distributed datacenters. Volley incorporates an iterative optimization algorithm based on weighted spherical means that handles the complexities of shared data and data inter-dependencies, and Volley can be re-run with sufficient speed that it handles application changes, reaching datacenter capacity limits and user mobility. Datacenter applications make use of Volley by submitting request logs (similar to Pinpoint [7] or X-Trace [14]) to a distributed storage system. These request logs include the client IP addresses, GUIDs identifying the data items accessed by the client requests, and the structure of the request "call tree", such as a client request updating Facebook wall 1, which triggers requests to data items 2 and 3 handling Facebook user RSS feeds.

Volley continuously analyzes these request logs to determine how application data should be migrated between datacenters. To scale to these data sets, Volley is designed to work in SCOPE [5], a system similar to MapReduce [11]. By leveraging SCOPE, Volley performs more than 400 machines hours worth of computation in less then a day. When migration is found to be worthwhile, Volley triggers application-specific data migration mechanisms. While prior work has studied placing static content across CDNs, Volley is the first research system to address placement of user data and other dynamic application data across geographically distributed datacenters.

Datacenter service administrators make use of Volley by specifying three inputs. First, administrators define the datacenter locations and a cost and capacity model (e.g., the cost of bandwidth between datacenters and the maximum amount of data per datacenter). Second, they choose the desired trade-off between upfront migration cost and ongoing better performance, where ongoing performance includes both minimizing user-perceived latency and reducing the costs of inter-datacenter communication. Third, they specify data replication levels and other constraints (e.g., three replicas in three different datacenters all located within Europe). This allows administrators to use Volley while respecting other external factors, such as contractual agreements and legislation.

In the rest of this paper, we first quantify the prevalence of trends such as user mobility in modern cloud services by analyzing month-long traces from Live Mesh and Live Messenger, two large-scale commercial datacenter services. We then present the design and implementation of the Volley system for computing data placement across geo-distributed datacenters. Next, we evaluate Volley analytically using the month-long Live Mesh trace, and we evaluate Volley on a live testbed consisting of 20 VMs located in 12 commercial datacenters distributed around the world. Previewing our results, we find that compared to a state-of-the-art heuristic, Volley can reduce skew in datacenter load by over $2\times$, decrease inter-datacenter traffic by over $1.8\times$, and reduce 75th percentile latency by over 30%. Finally, we survey related work and conclude.

2 Analysis of Commercial Cloud-Service Traces

We begin by analyzing workload traces collected by two large datacenter applications, Live Mesh [28] and Live Messenger [29]. Live Mesh provides a number of communication and collaboration features, such as file sharing and synchronization, as well as remote access to devices running the Live Mesh client. Live Messenger is an instant messaging application. In our presentation, we also use Facebook as a source for examples due to its ubiquity.

The Live Mesh and Live Messenger traces were collected during June 2009, and they cover all users and devices that accessed these services over this entire month. The Live Mesh trace contains a log entry for every modification to hard state (such as changes to a file in the Live Mesh synchronization service) and user-visible soft state (such as device connectivity information stored on a pool of in-memory servers [1]). The Live Messenger trace contains all login and logoff events, all IM conversations and the participants in each conversation, and the total number of messages in each conversation. The Live Messenger trace does not specify the sender or the size of individual messages, and so for simplicity, we model each participant in an IM conversation as having an equal likelihood of sending each message, and we divide the total message bytes in this conversation equally among all messages. A prior measurement study describes many aspects of user behavior in the Live Messenger system [24]. In both traces, clients are identified by application-level unique identifiers.

To estimate client location, we use a standard commercial geo-location database [34] as in prior work [36].

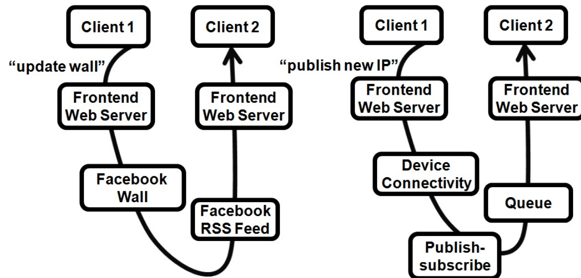


Figure 1. *Simplified data inter-dependencies in Facebook (left) and Live Mesh (right). In Facebook, an "updated wall" request arrives at a Facebook wall data item, and this data item sends the request to an RSS feed data item, which then sends it to the other client. In Live Mesh, a "publish new IP" request arrives at a Device Connectivity data item, which forwards it to a Publish-subscribe data item. From there, it is sent to a Queue data item, which finally sends it on to the other client. These pieces of data may be in different datacenters, and if they are, communication between data items incurs expensive inter-datacenter traffic.*

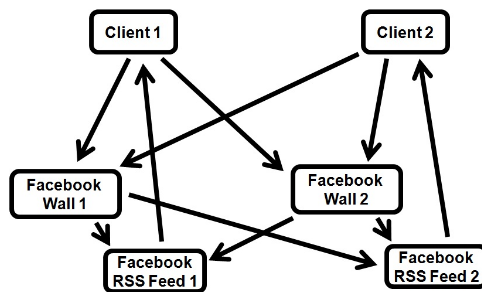


Figure 2. *Two clients, their four data items and the communication between them in the simplified Facebook example. Data placement requires appropriately mapping the four data items to datacenters so as to simultaneously achieve low inter-datacenter traffic, low datacenter capacity skew, and low latency.*

The database snapshot is from June 30th 2009, the very end of our trace period.

We use the traces to study three of the trends motivating Volley: shared data, data inter-dependencies, and user mobility. The other motivating trends for Volley, rapid application changes and reaching datacenter capacity limits, are documented in other data sources, such as developers describing how they build cloud services and how often they have to release updates [1, 35]. To provide some background on how data inter-dependencies arise in commercial cloud services, Figure 1 shows simplified examples from Facebook and Live Mesh. In the Facebook example, Client 1 updates its Facebook wall, which is then published to Client 2; in Facebook, this allows users to learn of each other's activities. In the Live Mesh example, Client 1 publishes its new IP address,

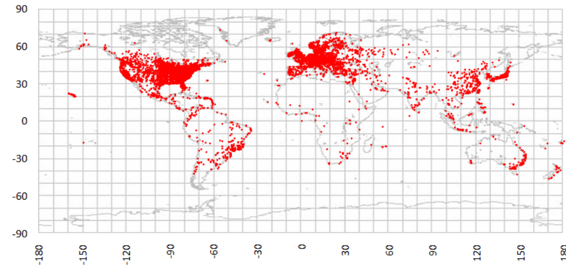


Figure 3. *Distribution of clients in the Mesh trace.*

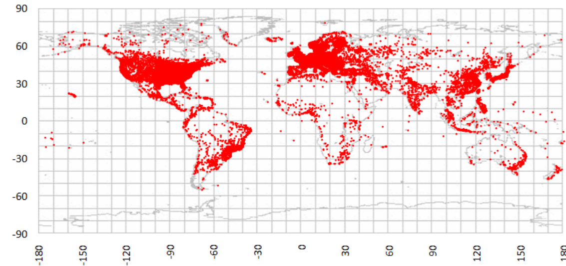


Figure 4. *Distribution of clients in the Messenger trace.*

which is routed to Client 2, enabling Client 2 to connect directly to Client 1; in Live Mesh, this is referred to as a notification session, and it enables both efficient file sharing and remote device access. The Figure caption provides additional details, as do other publications [1]. In both cases, the client operations involve multiple datacenter items; inter-datacenter traffic is minimized by colocating these items, while the latency of this particular request is minimized by placing the data items as close as possible to the two clients.

Figure 2 attempts to convey some intuition for why data sharing and inter-dependencies make data placement challenging. The figure shows the web of connections between just two clients in the simplified Facebook example; these inter-connections determine whether a mapping of data items to datacenters achieves low inter-datacenter traffic, low datacenter capacity skew, and low latency. Actual cloud services face this problem with hundreds of millions of clients. Each client may access many data items, and these data items may need to communicate with each other to deliver results to clients. Furthermore, the clients may access the data items from a variety of devices at different locations. This leads to a large, complicated graph.

In order to understand the potential for this kind of inter-connection to occur between clients that are quite distant, we begin by characterizing the geographic diversity of clients in the traces.

Client Geographic Diversity: We first study the traces to understand the geographic diversity of these services' client populations. Figures 3 and 4 show the distribution of clients in the two traces on a map of the world. The figures show that both traces contain a geographi-

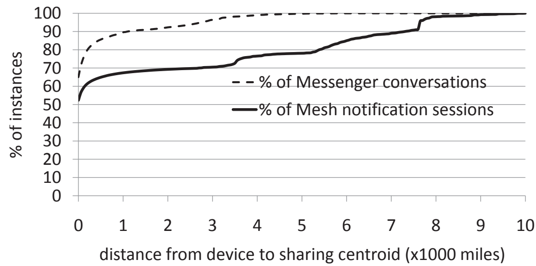


Figure 5. *Sharing of data between geographically distributed clients in the Messenger and Mesh traces. Large amounts of sharing occur between distant clients.*

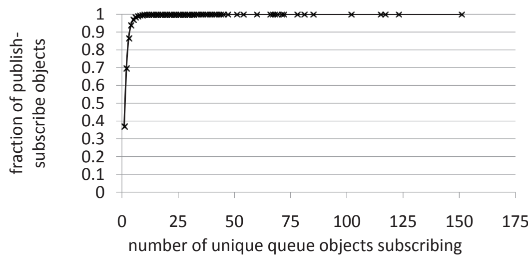


Figure 6. *Data inter-dependencies in Live Mesh between Publish-subscribe objects and Queue objects. A user that updates a hard state data item, such as a document stored in Live Mesh, will cause an update message to be generated at the Publish-subscribe object for that document, and all Queue objects that subscribe to it will receive a copy of the message. Each user or device that is sharing that document will have a unique Queue. Many Publish-subscribe objects are subscribed to by a single Queue, but there is a long tail of popular objects that are subscribed to by many Queues.*

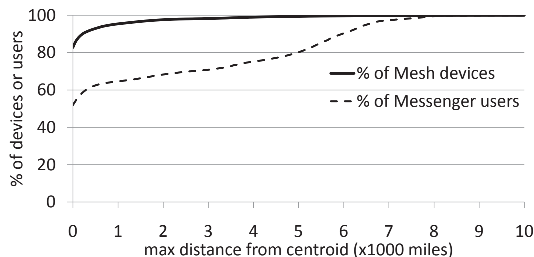


Figure 7. *Mobility of clients in the Messenger and Mesh traces. Most clients do not travel. However, a significant fraction do travel quite far.*

cally diverse set of clients, and thus these service's performance may significantly benefit from intelligent data placement.

Geographically Distant Data Sharing: We next study the traces to understand whether there is significant data sharing among distant users. For each particular data item, we compute its centroid (centroid on a sphere is computed using the weighted spherical mean methodology, which we describe in detail in Section 3). Figure 5 shows a CDF for the distance over which clients access data placed according to its centroid; data that is not shared has an access distance of 0, as does data shared by users whose IP addresses map to the same geographic location. Given the amount of collaboration across nations both within corporations and between them, it is perhaps not surprising that large amounts of sharing happens between very distant clients. This data suggests that even for static clients, there can be significant benefits to placing data closest to those who use it most heavily, rather than just placing it close to some particular client that accesses the data.

Data Inter-dependencies: We proceed to study the traces to understand the prevalence of data inter-dependencies. Our analysis focuses on Live Mesh because data inter-dependencies in Live Messenger have been documented in detail in prior work [24]. Figure 6 shows the number of Queue objects subscribing to receive notifications from each Publish-subscribe object; each such subscription creates a data inter-dependency where the Publish-subscribe object sends messages to the Queue object. We see that some Publish-subscribe objects send out notifications to only a single Queue object, but there is a long tail of popular Publish-subscribe objects. The presence of such data inter-dependencies motivates the need to incorporate them in Volley.

Client Mobility: We finally study the traces to understand the amount of client mobility in these services' client populations. Figure 7 shows a CDF characterizing client mobility over the month of the trace. To compute this CDF, we first computed the location of each client at each point in time that it contacted the Live Mesh or Live Messenger application using the previously described methodology, and we then compute the client's centroid. Next, we compute the maximum distance between each client and its centroid. As expected, we observe that most clients do not move. However, a significant fraction do move (more in the Messenger trace than the Mesh trace), and these movements can be quite dramatic – for comparison purposes, antipodal points on the earth are slightly more than 12,000 miles apart.

From these traces, we cannot characterize the reason for the movement. For example, it could be travel, or it could be that the clients are connecting in through a VPN to a remote office, causing their connection to the public Internet to suddenly emerge in a dramatically different location. For Volley's goal of reducing client latency, there is no need to distinguish between these different causes; even though the client did not physically move in the VPN case, client latency is still minimized by moving data closer to the location of the client's new connection to the public Internet. The long tail of client mobility suggests that for some fraction of clients, the ideal data placement changes significantly during this month.

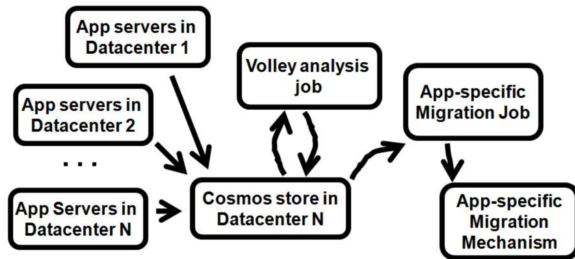


Figure 8. *Dataflow for an application using Volley.*

This data does leave open the possibility that some fraction of the observed clients are bots that do not correspond to an actual user (i.e., they are modified clients driven by a program). The current analysis does filter out the automated clients that the service itself uses for doing performance measurement from various locations. Prior work has looked at identifying bots automatically [45], and Volley might benefit from leveraging such techniques.

3 System Design and Implementation

The overall flow of data in the system is shown in Figure 8. Applications make use of Volley by logging data to the Cosmos [5] distributed storage system. The administrator must also supply some inputs, such as a cost and capacity model for the datacenters. The Volley system frequently runs new analysis jobs over these logs, and computes migration decisions. Application-specific jobs then feed these migration decisions into application-specific data migration mechanisms. We now describe these steps in greater detail.

3.1 Logging Requests

To utilize Volley, applications have to log information on the requests they process. These logs must enable correlating requests into "call trees" or "runtime paths" that capture the logical flow of control across components, as in Pinpoint [7] or X-Trace [14]. If the source or destination of a request is movable (i.e., because it is a data item under the control of the cloud service), we log a GUID identifier rather than its IP address; IP addresses are only used for endpoints that are not movable by Volley, such as the location that a user request came from. Because Volley is responsible for placing all the data named by GUIDs, it already knows their current locations in the steady state. It is sometimes possible for both the source and destination of a request to be referred to by GUIDs—this would happen, for example, in Figure 1, where the GUIDs would refer to Client 1's Facebook wall and Client 2's Facebook RSS feed. The exact fields in the Volley request logs are shown in Table 1. In total, each record requires only 100 bytes.

There has been substantial prior work modifying applications to log this kind of information, and many commercial applications (such as the Live Mesh and Live Messenger services analyzed in Section 2) already log a superset of this data. For such applications, Volley can incorporate simple filters to extract out the relevant subset of the logs.

For the Live Mesh and Live Messenger commercial cloud services, the data volumes from generating Volley logs are much less than the data volumes from processing user requests. For example, recording Volley logs for all the requests for Live Messenger, an IM service with hundreds of millions of users, only requires hundreds of GB per day, which leads to an average bandwidth demand in the tens of Mbps [24]. Though we cannot reveal the exact bandwidth consumption of the Live Mesh and Live Messenger services due to confidentiality concerns, we can state that tens of Mbps is a small fraction of the total bandwidth demands of the services themselves. Based on this calculation, we centralize all the logs in a single datacenter; this then allows Volley to run over the logs multiple times as part of computing a recommended set of migrations.

3.2 Additional Inputs

In addition to the request logs, Volley requires four inputs that change on slower time scales. Because they change on slower time scales, they do not noticeably contribute to the bandwidth required by Volley. These additional inputs are (1) the requirements on RAM, disk, and CPU per transaction for each type of data handled by Volley (e.g., a Facebook wall), (2) a capacity and cost model for all the datacenters, (3) a model of latency between datacenters and between datacenters and clients, and (4) optionally, additional constraints on data placement (e.g., legal constraints). Volley also requires the current location of every data item in order to know whether a computed placement keeps an item in place or requires migration. In the steady state, these locations are simply remembered from previous iterations of Volley.

In the applications we have analyzed thus far, the administrator only needs to estimate the average requirements on RAM, disk and CPU per data item; the administrator can then rely on statistical multiplexing to smooth out the differences between data items that consume more or fewer resources than average. Because of this, resource requirements can be estimated by looking at OS-provided performance counters and calculating the average resource usage for each piece of application data hosted on a given server.

The capacity and cost models for each datacenter specify the RAM, disk and CPU provisioned for the service in that datacenter, the available network bandwidth for both egress and ingress, and the charging model for service use of network bandwidth. While energy usage is a significant cost for datacenter owners, in our expe-

Request Log Record Format

Field	Meaning
Timestamp	Time in seconds when request was received (4B)
Source-Entity	A GUID if the source is another data item, an IP address if it is a client (40B)
Request-Size	Bytes in request (8B)
Destination-Entity	Like Source-Entity, either a GUID or an IP address (40B)
Transaction-Id	Used to group related requests (8B)

Table 1. *To use Volley, the application logs a record with these fields for every request. The meaning and size in bytes of each field are also shown.*

Migration Proposal Record Format

Field	Meaning
Entity	The GUID naming the entity (40B)
Datacenter	The GUID naming the new datacenter for this entity (40B)
Latency-Change	The average change in latency per request to this object (4B)
Ongoing-Bandwidth-Change	The change in egress and ingress bandwidth per day (4B)
Migration-Bandwidth	The one-time bandwidth required to migrate (4B)

Table 2. *Volley constructs a set of proposed migrations described using the records above. Volley then selects the final set of migrations according to the administrator-defined trade-off between performance and cost.*

rience this is incorporated as a fixed cost per server that is factored in at the long timescale of server provisioning. Although datacenter owners may be charged based on peak bandwidth usage on individual peering links, the unpredictability of any given service's contribution to a datacenter-wide peak leads datacenter owners to charge services based on total bandwidth usage, as in Amazon's EC2 [2]. Accordingly, Volley helps services minimize their total bandwidth usage. We expect the capacity and cost models to be stable at the timescale of migration. For fluid provisioning models where additional datacenter capacity can be added dynamically as needed for a service, Volley can be trivially modified to ignore provisioned capacity limits.

Volley needs a latency model to make placement decisions that reduce user perceived latency. It allows different static or dynamic models to be plugged in. Volley migrates state at large timescales (measured in days) and hence it should use a latency model that is stable at that timescale. Based on the large body of work demonstrating the effectiveness of network coordinate systems, we designed Volley to treat latencies between IPs as distances in some n-dimensional space specified by the model. For the purposes of evaluation in the paper, we rely on a static latency model because it is stable over these large timescales. This model is based on a linear regression of great-circle distance between geographic coordinates; it was developed in prior work [36], where it was compared to measured round trip times across millions of clients and shown to be reasonably accurate. This latency model requires translating client IP addresses to geographic coordinates, and for this purpose we rely on the geo-location database mentioned in Section 2. This geo-location database is updated every two weeks. In this work, we focus on improving latency to users and not bandwidth to users. Incorporating bandwidth would require both specifying a desired latency bandwidth tradeoff and a model for bandwidth between arbitrary points in the Internet.

Constraints on data placement can come in many forms. They may reflect legal constraints that data be hosted only in a certain jurisdiction, or they may reflect operational considerations requiring two replicas to be physically located in distant datacenters. Volley models such replicas as two distinct data items that may have a large amount of inter-item communication, along with the constraint that they be located in different datacenters. Although the commercial cloud service operators we spoke with emphasized the need to accommodate such constraints, the commercial applications we study in this paper do not currently face constraints of this form, and so although Volley can incorporate them, we did not explore this in our evaluation.

3.3 Volley Algorithm

Once the data is in Cosmos, Volley periodically analyzes it for migration opportunities. To perform the analysis, Volley relies on the SCOPE [5] distributed execution infrastructure, which at a high level resembles MapReduce [11] with a SQL-like query language. In our current implementation, Volley takes approximately 14 hours to run through one month's worth of log files; we analyze the demands Volley places on SCOPE in more detail in Section 4.4.

Volley's SCOPE jobs are structured into three phases. The search for a solution happens in Phase 2. Prior work [36] has demonstrated that starting this search in a good location improves convergence time, and hence

Recursive Step:

$$wsm\left(\{w_i, \vec{x}_i\}_{i=1}^{N}\right) = interp\left(\frac{w_N}{\sum w_i}, \vec{x}_N, wsm(\{w_i, \vec{x}_i\}_{i=1}^{N-1}\right)$$

Base Case:

$$interp(w, \vec{x}_A, \vec{x}_B) = (\phi_C, \lambda_C) = \vec{x}_C$$

$$\begin{aligned}
d &= \cos^{-1}\left[\cos(\phi_A)\cos(\phi_B) + \sin(\phi_A)\sin(\phi_B)\cos(\lambda_B - \lambda_A)\right] \\
\gamma &= \tan^{-1}\left[\frac{\sin(\phi_B)\sin(\phi_A)\sin(\lambda_B - \lambda_A)}{\cos(\phi_A) - \cos(d)\cos(\phi_B)}\right] \\
\beta &= \tan^{-1}\left[\frac{\sin(\phi_B)\sin(wd)\sin(\gamma)}{\cos(wd) - \cos(\phi_A)\cos(\phi_B)}\right] \\
\phi_C &= \cos^{-1}\left[\cos(wd)\cos(\phi_B) + \sin(wd)\sin(\phi_B)\cos(\gamma)\right] \\
\lambda_C &= \lambda_B - \beta
\end{aligned}$$

Figure 9. *Weighted spherical mean calculation. The weighted spherical mean (wsm()) is defined recursively as a weighted interpolation (interp()) between pairs of points. Here, w_i is the weight assigned to $\vec{x}_i$, and $\vec{x}_i$ (the coordinates for node i) consists of ϕ_i, the latitudinal distance in radians between node i and the North Pole, and λ_i, the longitude in radians of node i. The new (interpolated) node C consists of w parts node A and $1 - w$ parts node B; d is the current distance in radians between A and B; γ is the angle from the North Pole to B to A (which stays the same as A moves); β is the angle from B to the North Pole to A's new location. These are used to compute $\vec{x}_C$, the result of the interp(). For simplicity of presentation, we omit describing the special case for antipodal nodes.*

Phase 1 computes a reasonable initial placement of data items based on client IP addresses. Phase 2 iteratively improves the placement of data items by moving them freely over the surface of the earth—this phase requires the bulk of the computational time and the algorithm code. Phase 3 does the needed fix up to map the data items to datacenters and to satisfy datacenter capacity constraints. The output of the jobs is a set of potential migration actions with the format described in Table 2. Many adaptive systems must incorporate explicit elements to prevent oscillations. Volley does not incorporate an explicit mechanism for oscillation damping. Oscillations would occur only if *user behavior* changed in response to Volley migration in such a way that Volley needed to move that user's state back to a previous location.

Phase 1: Compute Initial Placement. We first map each client to a set of geographic coordinates using the commercial geo-location database mentioned earlier. This IP-to-location mapping may be updated between Volley jobs, but it is not updated within a single Volley job. We then map each data item that is directly accessed by a client to the weighted average of the geographic coordinates for the client IPs that access it. This is done using the weighted spherical mean calculation shown in Figure 9. The weights are given by the amount of communication between the client nodes and the data item whose initial location we are calculating. The weighted spherical mean calculation can be thought of as drawing an arc on the earth between two points, and then finding the point on the arc that interpolates between the two initial points in proportion to their weight. This operation is then repeated to average in additional points. The recursive definition of weighted spherical mean in Figure 9 is conceptually similar to defining the more familiar weighted mean recursively, e.g.,

$$\text{weighted-mean}(\{3, x_k\}, \{2, x_j\}, \{1, x_i\}) = \left(\frac{3}{6} \cdot x_k + \frac{3}{6} \cdot \text{weighted-mean}(\{2, x_j\}, \{1, x_i\})\right)$$

Compared to weighted mean, weighted spherical mean has the subtlety that the rule for averaging two individual points has to use spherical coordinates.

Figure 10 shows an example of this calculation using data from the Live Mesh trace: five different devices access a single shared object from a total of eight different IP addresses; device D accesses the shared object far more than the other devices, and this leads to the weighted spherical mean (labeled "centroid" in the figure) being placed very close to device D.

Finally, for each data item that is never accessed directly by clients (e.g., the Publish-subscribe data item in the Live Mesh example of Figure 1), we map it to the weighted spherical mean of the data items that communicate with it using the positions these other items were already assigned.

Phase 2: Iteratively Move Data to Reduce Latency. Volley iteratively moves data items closer to both clients and to the other data items that they communicate with. This iterative update step incorporates two earlier ideas: a weighted spring model as in Vivaldi [9] and spherical coordinates as in Htrae [36]. Spherical coordinates define the locations of clients and data items in a way that is more conducive to incorporating a latency model for geographic locations. The latency distance between two nodes and the amount of communication between them increase the spring force that is pulling them together. However, unlike a network coordinate system, nodes in

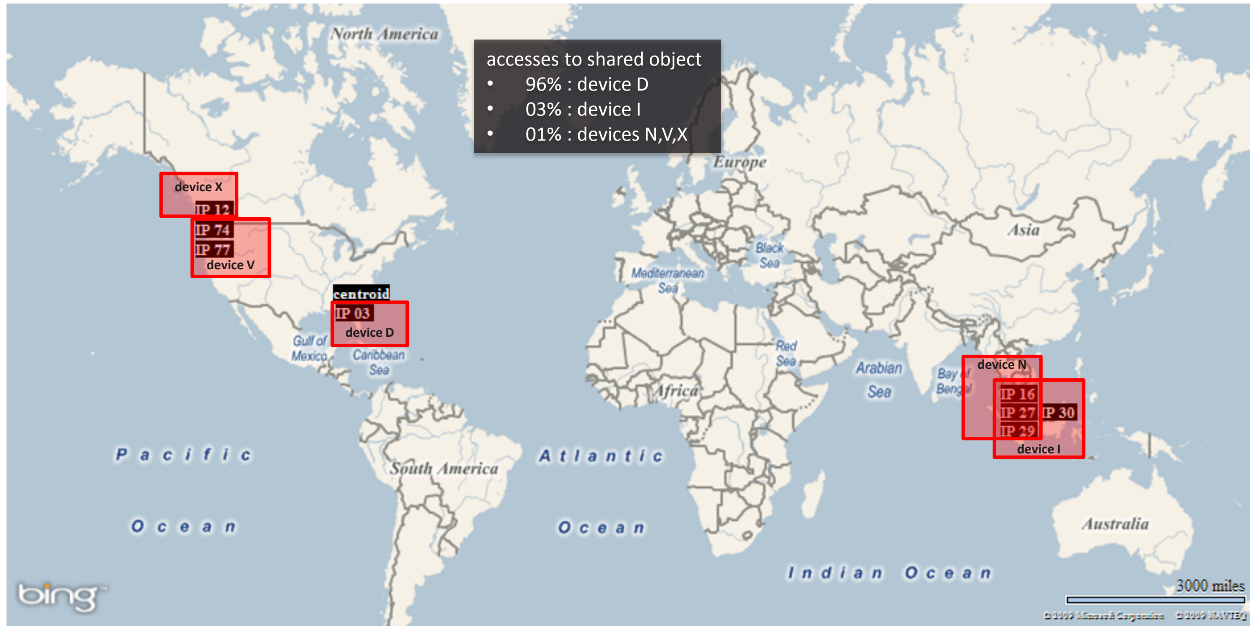


Figure 10. *An example of a shared object being placed at its weighted spherical mean (labeled "centroid" in the Figure). This particular object, the locations of the clients that access it, and their access ratios are drawn from the Live Mesh trace. Because device D is responsible for almost all of the accesses, the weighted spherical mean placement for the object is very close to device D's location.*

$$w = \frac{1}{1 + \kappa \cdot d \cdot l_{AB}}$$
$$\vec{x}_A^{new} = interp(w, \vec{x}_A^{current}, \vec{x}_B^{current})$$

Figure 11. *Update rule applied to iteratively move nodes with more communication closer together. Here, w is a fractional weight that determines how much node A is moved towards node B, l_{AB} is the amount of communication between the two nodes, d is the distance between nodes A and B, $\vec{x}_A^{current}$ and $\vec{x}_B^{current}$ are the current locations of node A and B, $\vec{x}_A^{new}$ is the location of A after the update, and κ is an algorithmic constant.*

Volley only experience contracting forces; the only factor preventing them from collapsing to a single location is the fixed nature of client locations. This yields the update rule shown in Figure 11. In our current implementation, we simply run a fixed number of iterations of this update rule; we show in Section 4 that this suffices for good convergence.

Intuitively, Volley's spring model attempts to bring data items closer to users and to other data items that they communicate with regularly. Thus it is plausible that Volley's spring model will simultaneously reduce latency and reduce inter-datacenter traffic; we show in Section 4 that this is indeed the case for the commercial cloud services that we study.

Phase 3: Iteratively Collapse Data to Datacenters. After computing a nearly ideal placement of the data items on the surface of the earth, we have to modify this placement so that the data items are located in datacenters, and the set of items in each datacenter satisfies its capacity constraints. Like Phase 2, this is done iteratively: initially, every data item is mapped to its closest datacenter. For datacenters that are over their capacity, Volley identifies the items that experience the fewest accesses, and moves all of them to the next closest datacenter. Because this may still exceed the total capacity of some datacenter due to new additions, Volley repeats the process until no datacenter is over capacity. Assuming that the system has enough capacity to successfully host all items, this algorithm always terminates in at most as many iterations as there are datacenters in the system.

For each data item that has moved, Volley outputs a migration proposal containing the new datacenter location, the new values for latency and ongoing inter-datacenter bandwidth, and the one-time bandwidth required for this migration. This is a straightforward calculation using the old data locations, the new data locations, and the inputs supplied by the datacenter service administrator, such as the cost model and the latency model. These migration proposals are then consumed by application-specific migration mechanisms.

3.4 Application-specific Migration

Volley is designed to be usable by many different cloud services. For Volley to compute a recommended placement, the only requirement it imposes on the cloud

service is that it logs the request data described in Table 1. Given these request logs as input, Volley outputs a set of migration proposals described in Table 2, and then leaves the actual migration of the data to the cloud service itself. If the cloud service also provides the initial location of data items, then each migration proposal will include the bandwidth required to migrate, and the expected change in latency and inter-datacenter bandwidth after migration.

Volley's decision to leave migration to application-specific migration mechanisms allows Volley to be more easily applied to a diverse set of datacenter applications. For example, some datacenter applications use migration mechanisms that follow the pattern of marking data read-only in the storage system at one location, copying the data to a new location, updating an application-specific name service to point to the new copy, marking the new copy as writeable, and then deleting the old copy. Other datacenter applications maintain multiple replicas in different datacenters, and migration may simple require designating a different replica as the primary. Independent of the migration mechanism, datacenter applications might desire to employ application-specific throttling policies, such as only migrating user state when an application-specific predictive model suggests the user is unlikely to access their state in the next hour. Because Volley does not attempt to migrate the data itself, it does not interfere with these techniques or any other migration technique that an application may wish to employ.

4 Evaluation

In our evaluation, we compare Volley to three heuristics for where to place data and show that Volley substantially outperforms all of them on the metrics of datacenter capacity skew, inter-datacenter traffic, and user-perceived latency. We focus exclusively on the month-long Live Mesh trace for conciseness. For both the heuristics and Volley, we first compute a data placement using a week of data from the Live Mesh trace, and then evaluate the quality of the resulting placement on the following three weeks of data. For all four placement methodologies, any data that appears in the three-week evaluation window but not in the one-week placement computation window is placed in a single datacenter located in the United States (in production, this new data will be handled the next time the placement methodology is run). Placing all previously unseen data in one datacenter penalizes the different methodologies equally for such data.

The first heuristic we consider is *commonIP* – place data as close as possible to the IP address that most commonly accesses it. The second heuristic is *oneDC* – put all data in one datacenter, a strategy still taken by many companies due to its simplicity. The third heuristic is *hash* – hash data to datacenters so as to optimize for load-balancing. These three heuristics represent reasonable approaches to optimizing for the three different metrics we consider—oneDC and hash optimize for inter-datacenter traffic and datacenter capacity skew respectively, while commonIP is a reasonably sophisticated proposal for optimizing latency.

Throughout our evaluation, we use 12 commercial datacenters as potential locations. These datacenters are distributed across multiple continents, but their exact locations are confidential. Confidentiality concerns also prevent us from revealing the exact amount of bandwidth consumed by our services. Thus, we present the inter-datacenter traffic from different placements using the metric "fraction of messages that are inter-datacenter." This allows an apples-to-apples comparison between the different heuristics and Volley without revealing the underlying bandwidth consumption. The bandwidth consumption from centralizing Volley logs, needed for Volley and commonIP, is so small compared to this inter-datacenter traffic that it does not affect graphs comparing this metric among the heuristics. We configure Volley with a datacenter capacity model such that no one of the 12 datacenters can host more than 10% of all data, a reasonably balanced use of capacity.

All latencies that we compute analytically use the latency model described in Section 3. This requires using the client's IP address in the trace to place them at a geographic location. In this Live Mesh application, client requests require sending a message to a first data item, which then sends a second message to a second data item; the second data item sends a reply, and then the first data item sends the client its reply. If the data items are in the same datacenter, latency is simply the round trip time between the client and the datacenter. If the data items are in separate datacenters, latency is the sum of four one-way delays: client to datacenter 1, datacenter 1 to datacenter 2, datacenter 2 back to datacenter 1, and datacenter 1 back to the client. These latency calculations leave out other potential protocol overheads, such as the need to initially establish a TCP connection or to authenticate; any such protocol overheads encountered in practice would magnify the importance of latency improvements by incurring the latency multiple times. For clarity of presentation, we consistently group latencies into 10 millisecond bins in our graphs. The graphs only present latency up to 250 milliseconds because the better placement methodologies all achieve latency well under this for almost all requests.

Our evaluation begins by comparing Volley and the three heuristics on the metrics of datacenter capacity skew and inter-datacenter traffic (Section 4.1). Next, we evaluate the impact of these placements on the latency of client requests, including evaluating Volley in the con-

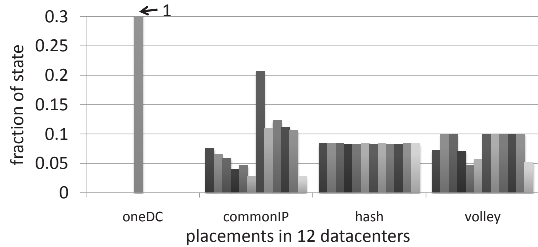


Figure 12. *Datacenter capacity required by three different placement heuristics and Volley.*

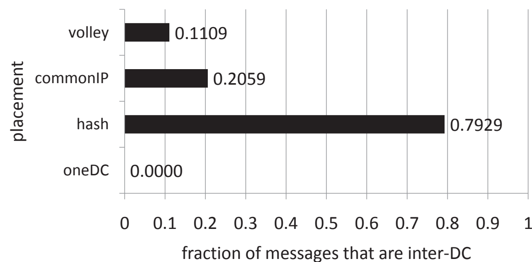


Figure 13. *Inter-datacenter traffic under three different placement heuristics and Volley.*

text of a simple, hypothetical example to understand this impact in detail (Section 4.2). We then evaluate the incremental benefit of Volley as a function of the number of Volley iterations (Section 4.3). Next, we evaluate the resource demands of running Volley on the SCOPE distributed execution infrastructure (Section 4.4). Finally, we evaluate the impact of running Volley more frequently or less frequently (Section 4.5).

4.1 Impact on Datacenter Capacity Skew and Inter-datacenter Traffic

We now compare Volley to the three heuristics for where to place data and show that Volley substantially outperforms all of them on the metrics of datacenter capacity skew and inter-datacenter traffic. Figures 12 and 13 show the results: hash has perfectly balanced use of capacity, but high inter-datacenter traffic; oneDC has zero inter-datacenter traffic (the ideal), but extremely unbalanced use of capacity; and commonIP has a modest amount of inter-datacenter traffic, and capacity skew where 1 datacenter has to support more than twice the load of the average datacenter. Volley is able to meet a reasonably balanced use of capacity while keeping inter-datacenter traffic at a very small fraction of the total number of messages. In particular, compared to commonIP, Volley reduces datacenter skew by over 2× and reduces inter-datacenter traffic by over 1.8×.

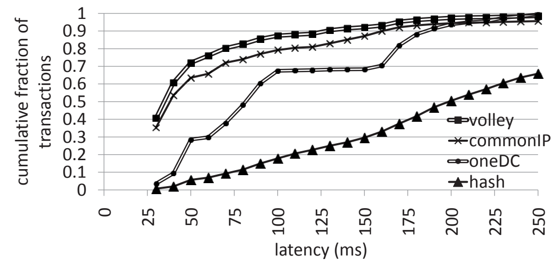


Figure 14. *Client request latency under three different placement heuristics and Volley.*

4.2 Impact on Latency of Client Requests

We now compare Volley to the three heuristics on the metric of user-perceived latency. Figure 14 shows the results: hash has high latency; oneDC has mediocre latency; and commonIP has the best latency among the three heuristics. Although commonIP performs better than oneDC and hash, Volley performs better still, particularly on the tail of users that experience high latency even under the commonIP placement strategy. Compared to commonIP, Volley reduces 75th percentile latency by over 30%.

4.2.1 Multiple Datacenter Testbed

Previously, we evaluated the impact of placement on user-perceived latency analytically use the latency model described in Section 3. In this section, we evaluate Volley's latency impact on a live system using a prototype cloud service. We use the prototype cloud service to emulate Live Mesh for the purpose of replaying a subset of the Live Mesh trace. We deployed the prototype cloud service across 20 virtual machines spread across the 12 geographically distributed datacenters, and we used one node at each of 109 Planetlab sites to act as clients of the system.

The prototype cloud service consists of four components: the frontend, the document service, the publish-subscribe service, and the message queue service. Each of these components run on every VM so as to have every service running in every datacenter. These components of our prototype map directly to the actual Live Mesh component services that run in production. The ways in which the production component services cooperate to provide features in the Live Mesh service is described in detail elsewhere [1], and we provide only a brief overview here.

The prototype cloud service exposes a simple frontend that accepts client requests and routes them to the appropriate component in either its own or another datacenter. In this way, each client can connect directly to any datacenter, and requests that require an additional step (e.g., updating an item, and then sending the update to others) will be forwarded appropriately. This design

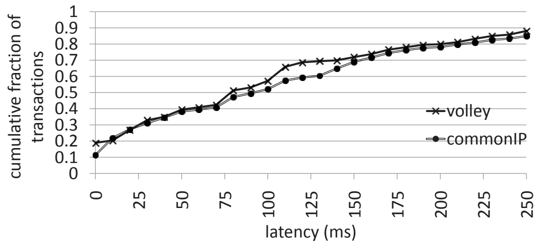


Figure 15. *Comparing Volley to the commonIP heuristic on a live system spanning 12 geographically distributed datacenters and accessed by Planetlab clients. In this Figure, we use a random sample of the Live Mesh trace. We see that Volley provides moderately better latency than the commonIP heuristic.*

allows clients to cache the location of the best datacenter to connect to for any given operation, but requests still succeed if a client request arrives at the wrong datacenter due to cache staleness.

We walk through an example of how two clients can rendezvous by using the document, publish-subscribe, and message queue services. The document service can store arbitrary data; in this case, the first client can store its current IP address, and a second client can then read that IP address from the document service and contact the first client directly. The publish-subscribe service is used to send out messages when data in the document service changes; for example, if the second client subscribes to updates for the first client's IP address, these updates will be pro-actively sent to the second client, instead of the second client having to poll the document service to see if there have been any changes. Finally, the message queue service buffers messages for clients from the publish-subscribe service. If the client goes offline and then reconnects, it can connect to the queue service and dequeue these messages.

To evaluate both Volley and the commonIP heuristic's latency on this live system, we used the same data placements computed on the first week of the Live Mesh trace. Because the actual Live Mesh service requires more than 20 VMs, we had to randomly sample requests from the trace before replaying it. We also mapped each client IP in the trace subset to the closest Planetlab node, and replayed the client requests from these nodes.

Figure 15 shows the measured latency on the sample of the Live Mesh trace; recall that we are grouping latencies into 10 millisecond bins for clarity of presentation. We see that Volley consistently provides better latency than the commonIP placement. These latency benefits are visible despite a relatively large number of external sources of noise, such as the difference between the actual client locations and the Planetlab locations, differences between typical client connectivity (that Volley's latency model relies on) and Planetlab connectivity, and occasional high load on the Planetlab nodes leading to high slice scheduling delays.

Timestamp	Source-Entity	Request-Size	Destination-Entity	Transaction-Id
T_0	PSS^a	100 B	Q^1	1
T_0	Q^1	100 B	IP^1	1
T_0 + 1	PSS^b	100 B	Q^1	2
T_0 + 1	Q^1	100 B	IP^2	2
T_0 + 2	PSS^b	100 B	Q^1	3
T_0 + 2	Q^1	100 B	IP^2	3
T_0 + 5	PSS^b	100 B	Q^2	4
T_0 + 5	Q^2	100 B	IP^1	4

Table 3. *Hypothetical application logs. In this example, IP^1 is located at geographic coordinates (10,110) and IP^2 at (10,10).*

Data	commonIP	Volley Phase 1	Volley Phase 2
PSS^a	(10,110)	(14.7,43.1)	(15.1,49.2)
PSS^b	(10,10)	(15.3,65.6)	(15.3,63.6)
Q^1	(10,10)	(14.7,43.1)	(15.1,50.5)
Q^2	(10,110)	(10,110)	(13.6,88.4)

Table 4. *CommonIP and Volley placements computed using Table 3, assuming a datacenter at every point on Earth and ignoring capacity constraints and inter-datacenter traffic.*

Transaction-Id	commonIP distance	commonIP latency	Volley Phase 2 distance	Volley Phase 2 latency
1	27,070 miles	396 ms	8,202 miles	142 ms
2	0 miles	31 ms	7,246 miles	129 ms
3	0 miles	31 ms	7,246 miles	129 ms
4	13,535 miles	182 ms	6,289 miles	116 ms

Table 5. *Distances traversed and latencies of user requests in Table 3 using commonIP and Volley Phase 2 placements in Table 4. Note that our latency model [36] includes an empirically-determined access penalty for all communication involving a client.*

Other than due to sampling of the request trace, the live experiment has no impact on the datacenter capacity skew and inter-datacenter traffic differences between the two placement methodologies. Thus, Volley offers an improvement over commonIP on every metric simultaneously, with the biggest benefits coming in reduced inter-datacenter traffic and reduced datacenter capacity skew.

4.2.2 Detailed Examination of Latency Impact

To examine in detail how placement decisions impact latencies experienced by user requests, we now consider a simple example. Table 3 lists four hypothetical Live Mesh transactions involving four data objects and clients behind two IP addresses. For the purposes of this simple example, we assume there is a datacenter at every point on Earth with infinite capacity and no inter-datacenter traffic costs. We pick the geographic coor-

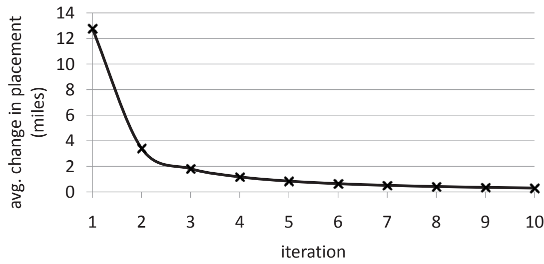


Figure 16. *Average distance traveled by each object during successive Volley Phase 2 iterations. The average incorporates some objects traveling quite far, while many travel very little.*

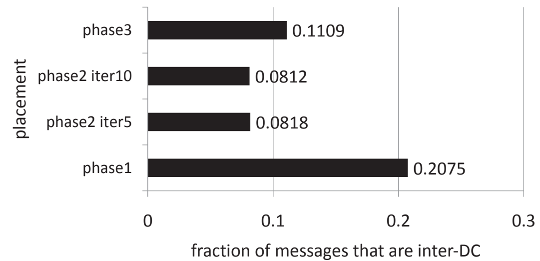


Figure 17. *Inter-datacenter traffic at each Volley iteration.*

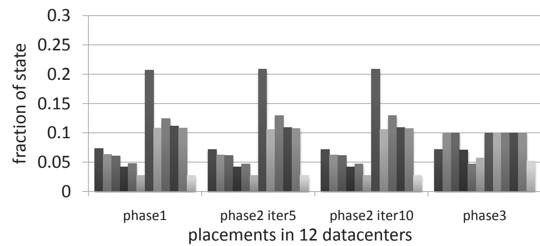


Figure 18. *Datacenter capacity at each Volley iteration.*

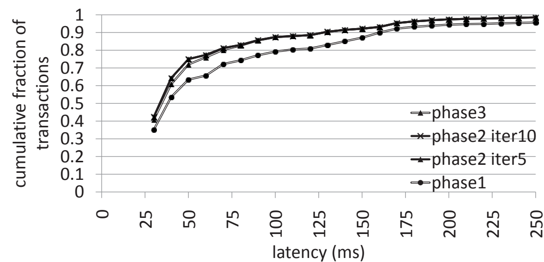


Figure 19. *Client request latency at each Volley iteration.*

dinates of (10,110) and (10,10) for ease of examining how far each object's placement is from the client IP addresses. Table 4 shows the placements calculated by commonIP and Volley in Phases 1 and 2. In Phase 1, Volley calculates the weighted spherical mean of the geographic coordinates for the client IPs that access each "Q" object. Hence Q^1 is placed roughly two-thirds along the great-circle segment from IP^1 to IP^2, while Q^2 is placed at IP^1. Phase 1 similarly calculates the placement of each "PSS" object using these coordinates for "Q" objects. Phase 2 then iteratively refines these coordinates.

We now consider the latency impact of these placements on the *same* set of user requests in Table 3. Table 5 shows for each user request, the physical distance traversed and the corresponding latency (round trip from PSS to Q and from Q to IP). CommonIP optimizes for client locations that are most frequently used, thereby driving down latency to the minimum for some user requests but at a significant expense to others. Volley considers all client locations when calculating placements, and in doing so drives down the worst cases by more than the amount it drives up the common case, leading to an overall better latency distribution. Note that in practice, user requests change over time after placement decisions have been made and our trace-based evaluation does use later sets of user requests to evaluate placements based on earlier requests.

4.3 Impact of Volley Iteration Count

We now show that Volley converges after a small number of iterations; this will allow us to establish in Section 4.5 that Volley runs quickly (i.e., less than a day), and thus can be re-run frequently. Figures 16, 17, 18 and 19 show the performance of Volley as the number of iterations varies. Figure 16 shows that the distance that Volley moves data significantly decreases with each Volley iteration, showing that Volley relatively quickly converges to its ideal placement of data items.

Figures 17, 18 and 19 further break down the changes in Volley's performance in each iteration. Figure 17 shows that inter-datacenter traffic is reasonably good after the initial placement of Phase 1, and is quite similar to the commonIP heuristic. In contrast, recall that the hash heuristic led to almost 80% of messages crossing datacenter boundaries. Inter-datacenter traffic then decreases by over a factor of 2 during the first 5 Phase 2 iterations, decreases by a small amount more during the next 5 Phase 2 iterations, and finally goes back up slightly when Volley's Phase 3 balances the items across datacenters. Of course, the point of re-balancing is to avoid the kind of capacity skew seen in the commonIP heuristic, and in this regard a small increase in inter-datacenter traffic is acceptable.

Turning now to datacenter capacity, we see that Volley's placement is quite skewed (and by an approximately constant amount) until Phase 3, where it smooths out datacenter load according to its configured capacity

Volley Phase	Elapsed Time in Hours	SCOPE Stages	SCOPE Vertices	CPU Hours
1	1:22	39	20,668	89
2	14:15	625	255,228	386
3	0:10	16	200	0:07

Table 6. *Volley's demands on the SCOPE infrastructure to analyze 1 week's worth of traces.*

model (i.e., such that no one of the 12 datacenters hosts more than 10% of the data). Turning finally to latency, Figure 19 shows that latency has reached its minimum after only five Phase 2 iterations. In contrast to the impact on inter-datacenter traffic, there is almost no latency penalty from Phase 3's data movement to satisfy datacenter capacity.

4.4 Volley Resource Demands

Having established that Volley converges after a small number of iterations, we now analyze the resource requirements for this many iterations; this will allow us to conclude that Volley completes quickly and can be re-run frequently. The SCOPE cluster we use consists of well over 1,000 servers. Table 6 shows Volley's demands on the SCOPE infrastructure broken down by Volley's different phases. The elapsed time, SCOPE stages, SCOPE vertices and CPU hours are cumulative over each phase – Phase 1 has only one iteration to compute the initial placement, while Phase 2 has ten iterations to improve the placement, and Phase 3 has 12 iterations to balance out usage over the 12 datacenters. Each SCOPE stage in Table 6 corresponds approximately to a single map or reduce step in MapReduce [11]. There are 680 such stages overall, leading to lots of data shuffling; this is one reason why the total elapsed time is not simply CPU hours divided by the degree of possible parallelism. Every SCOPE vertex in Table 6 corresponds to a node in the computation graph that can be run on a single machine, and thus dividing the total number of vertices by the total number of stages yields the average degree of parallelism within Volley: the average stage parallelizes out to just over 406 machines (some run on substantially more). The SCOPE cluster is not dedicated for Volley but rather is a multi-purpose cluster used for several tasks. The operational cost of using the cluster for 16 hours every week is small compared to the operational savings in bandwidth consumption due to improved data placement. The data analyzed by Volley is measured in the terabytes. We cannot reveal the exact amount because it could be used to infer confidential request volumes since every Volley log record is 100 bytes.

4.5 Impact of Rapid Volley Re-Computation

Having established that Volley can be re-run frequently, we now show that Volley provides substantially

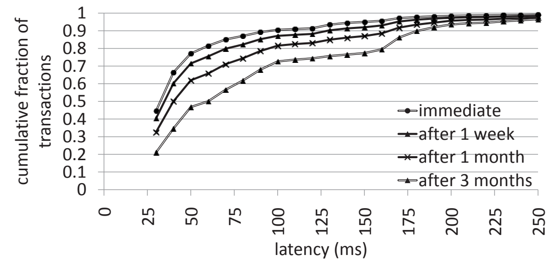


Figure 20. *Client request latency with stale Volley placements.*

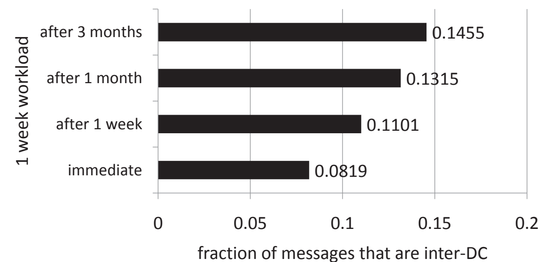


Figure 21. *Inter-datacenter traffic with stale Volley placements.*

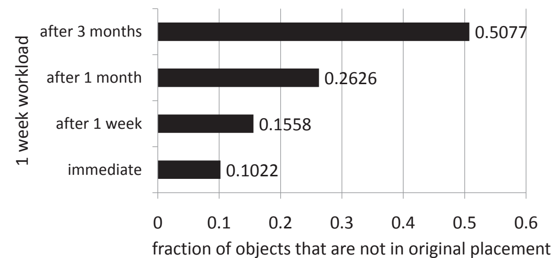


Figure 22. *Previously unseen objects over time.*

better performance by being re-run frequently. For these experiments, we use traces from the Live Mesh service extending from the beginning of June 2009 all the way to the beginning of September 2009. Figures 20, 21 and 22 show the impact of rapidly re-computing placements: Volley computes a data placement using the trace from the first week of June, and we evaluate the performance of this placement on a trace from the immediately following week, the week after the immediately following week, a week starting a month later, and a week starting three months after Volley computed the data placement. The better performance of the placement on the immediately following week demonstrates the significant benefits of running Volley frequently with respect to both latency and inter-datacenter traffic. Figure 20 shows that running Volley even every two weeks is noticeably worse than having just run Volley, and this latency penalty keeps increasing as the Volley placement

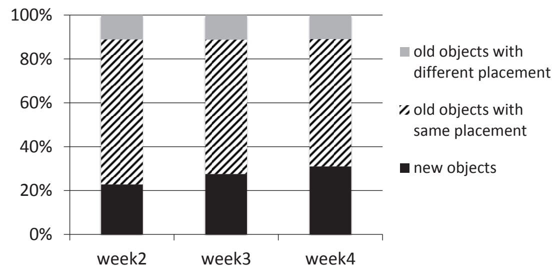


Figure 23. *Fraction of objects moved compared to first week.*

becomes increasingly stale. Figure 21 shows a similar progressively increasing penalty to inter-datacenter traffic; running Volley frequently results in significant inter-datacenter traffic savings.

Figure 22 provides some insight into why running Volley frequently is so helpful; the number of previously unseen objects increases rapidly with time. When run frequently, Volley detects accesses to an object sooner. Note that this inability to intelligently place previously unseen objects is shared by the commonIP heuristic, and so we do not separately evaluate the rate at which it degrades in performance.

In addition to new objects that are created and accessed, previously placed objects may experience significantly different access patterns over time. Running Volley periodically provides the added benefit of migrating these objects to locations that can better serve new access patterns. Figure 23 compares a Volley placement calculated from the first week of June to a placement calculated in the second week, then the first week to the third week, and finally the first week to the fourth week. About 10% of the objects in any week undergo migrations, either as a direct result of access pattern changes or due to more important objects displacing others in capacity-limited datacenters. The majority of objects retain their placement compared to the first week. Running Volley periodically has a third, but minor advantage. Some client requests come from IP addresses that are not present in geo-location databases. Objects that are accessed solely from such locations are not placed by Volley. If additional traces include accesses from other IP addresses that are present in geo-location databases, Volley can then place these objects based on these new accesses.

5 Related Work

The problem of automatic placement of application data re-surfaces with every new distributed computing environment, such as local area networks (LANs), mobile computing, sensor networks, and single cluster web sites. In characterizing related work, we first focus on the mechanisms and policies that were developed for these other distributed computing environments. We then describe prior work that focused on placing static content on CDNs; compared to this prior work, Volley is the first research system to address placement of dynamic application data across geographically distributed datacenters. We finally describe prior work on more theoretical approaches to determining an optimal data placement.

5.1 Placement Mechanisms

Systems such as Emerald [20], SOS [37], Globe [38], and Legion [25] focused on providing location-independent programming abstractions and migration mechanisms for moving data and computation between locations. Systems such as J-Orchestra [42] and Addistant [41] have examined distributed execution of Java applications through rewriting of byte code, but have left placement policy decisions to the user or developer. In contrast, Volley focuses on placement policy, not mechanism. Some prior work incorporated both placement mechanism and policy, e.g., Coign [18], and we characterize its differences with Volley's placement policy in the next subsection.

5.2 Placement Policies for Other Distributed Computing Environments

Prior work on automatic data placement can be broadly grouped by the distributed computing environment that it targeted. Placing data in a LAN was tackled by systems such as Coign [18], IDAP [22], ICOPS [31], CAGES [17], Abacus [3] and the system of Stewart et al [39]. Systems such as Spectra [13], Slingshot [40], MagnetOS [27], Pleaides [23] and Wishbone [33] explored data placement in a wireless context, either between mobile clients and more powerful servers, or in ad hoc and sensor networks. Hilda [44] and Doloto [30] explored splitting data between web clients and web servers, but neither assumed there were multiple geographic locations that could host the web server.

Volley differs from these prior systems in several ways. First, the scale of the data that Volley must process is significantly greater. This required designing the Volley algorithm to work in a scalable data analysis framework such as SCOPE [5] or MapReduce [11]. Second, Volley must place data across a large number of datacenters with widely varying latencies both between datacenters and clients, and between the datacenters themselves; this aspect of the problem is not addressed by the algorithms in prior work. Third, Volley must continuously update its measurements of the client workload, while some (though not all) of these prior approaches used an upfront profiling approach.

5.3 Placement Policies for Static Data

Data placement for Content Delivery Networks (CDNs) has been explored in many pieces of prior work [21, 19]. These systems have focused on static data – the HTTP caching header should be honored, but no other more elaborate synchronization between replicas is needed. Because of this, CDNs can easily employ decentralized algorithms e.g., each individual server or a small set of servers can independently make decisions about what data to cache. In contrast, Volley's need to deal with dynamic data would make a decentralized approach challenging; Volley instead opts to collect request data in a single datacenter and leverage the SCOPE distributed execution framework to analyze the request logs within this single datacenter.

5.4 Optimization Algorithms

Abstractly, Volley seeks to maps objects to locations so as to minimize a cost function. Although there are no known approximation algorithms for this general problem, the theory community has developed approximation algorithms for numerous more specialized settings, such as sparsest cut [4] and various flavors of facility location [6, 8]. To the best of our knowledge, the problem in Volley does not map to any of these previously studied specializations. For example, the problem in Volley differs from facility location in that there is a cost associated with placing two objects at different datacenters, not just costs between clients and objects. This motivates Volley's choice to use a heuristic approach and to experimentally validate the quality of the resulting data placement.

Although Volley offers a significant improvement over a state-of-the-art heuristic, we do not yet know how close it comes to an optimal placement; determining such an optimal placement is challenging because standard commercial optimization packages simply do not scale to the data sizes of large cloud services. This leaves open the tantalizing possibility that further improvements are possible beyond Volley.

6 Conclusion

Cloud services continue to grow to span large numbers of datacenters, making it increasingly urgent to develop automated techniques to place application data across these datacenters. Based on the analysis of month-long traces from two large-scale commercial cloud services, Microsoft's Live Messenger and Live Mesh, we built the Volley system to perform automatic data placement across geographically distributed datacenters. To scale to the large data volumes of cloud service logs, Volley is designed to work in the SCOPE [5] scalable data analysis framework.

We evaluate Volley analytically and on a live system consisting of a prototype cloud service running on a geographically distributed testbed of 12 datacenters. Our evaluation using one of the month-long traces shows that, compared to a state-of-the-art heuristic, Volley simultaneously reduces datacenter capacity skew by over $2\times$, reduces inter-datacenter traffic by over $1.8\times$, and reduces 75th percentile latency by over 30%. This shows the potential of Volley to simultaneously improve the user experience and significantly reduce datacenter costs.

While in this paper we have focused on using Volley to optimize data placement in existing datacenters, service operators could also use Volley to explore future sites for datacenters that would improve performance. By including candidate locations for datacenters in Volley's input, the operator can identify which combination of additional sites improve latency at modest costs in greater inter-datacenter traffic. We hope to explore this more in future work.

Acknowledgments

We greatly appreciate the support of Microsoft's ECN team for donating usage of their VMs, and the support of the Live Mesh and Live Messenger teams in sharing their data with us. We thank our shepherd, Dejan Kostic, and the anonymous reviewers for their detailed feedback and help in improving this paper.

References

[1] A. Adya, J. Dunagan, and A. Wolman. Centrifuge: Integrating Lease Management and Partitioning for Cloud Services. In *NSDI*, 2010.

[2] Amazon Web Services. http://aws.amazon.com.

[3] K. Amiri, D. Petrou, G. Ganger, and G. Gibson. Dynamic Function Placement for Data-intensive Cluster Computing. In *USENIX Annual Technical Conference*, 2000.

[4] S. Arora, S. Rao, and U. Vazirani. Expander Flows, Geometric Embeddings and Graph Partitioning. In *STOC*, 2004.

[5] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and efficient parallel processing of massive data sets. In *VLDB*, 2008.

[6] M. Charikar and S. Guha. Improved Combinatorial Algorithms for the Facility Location and k-Median Problems. In *FOCS*, 1999.

[7] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem Determination in Large, Dynamic Systems. In *DSN*, 2002.

[8] F. Chudak and D. Williamson. Improved approximation algorithms for capacitated facility location problems. *Mathematical Programming*, 102(2):207–222, 2005.

[9] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A Decentralized Network Coordinate System. In *SIGCOMM*, 2004.

[10] Data Center Global Expansion Trend.

http://www.datacenterknowledge.com/archives/2008/03/27/google-data-center-faq/.

[11] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *OSDI*, 2004.

[12] Facebook. http://www.facebook.com.

[13] J. Flinn, S. Park, and M. Satyanarayanan. Balancing Performance, Energy, and Quality in Pervasive Computing. In *ICDCS*, 2002.

[14] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica. X-Trace: A Pervasive Network Tracing Framework. In *NSDI*, 2007.

[15] Global Air travel Trends. http://www.zinnov.com/presentation/Global_Aviation-Markets-An_Ananlysis.pdf.

[16] Google Apps. http://apps.google.com.

[17] G. Hamlin Jr and J. Foley. Configurable applications for graphics employing satellites (CAGES). *ACM SIGGRAPH Computer Graphics*, 9(1):9–19, 1975.

[18] G. Hunt and M. Scott. The Coign Automatic Distributed Partitioning System. In *OSDI*, 1999.

[19] S. Jamin, C. Jin, A. Kurc, D. Raz, and Y. Shavitt. Constrained Mirror Placement on the Internet. In *INFOCOM*, 2001.

[20] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, 1966.

[21] M. Karlsson and M. Mahalingam. Do We Need Replica Placement Algorithms in Content Delivery Networks? In *International Workshop on Web Content Caching and Distribution (WCW)*, 2002.

[22] D. Kimelman, V. Rajan, T. Roth, M. Wegman, B. Lindsey, H. Lindsey, and S. Thomas. Dynamic Application Partitioning in VisualAge Generator Version 3.0. *Lecture Notes In Computer Science; Vol. 1543*, pages 547–548, 1998.

[23] N. Kothari, R. Gummadi, T. Millstein, and R. Govindan. Reliable and Efficient Programming Abstractions for Wireless Sensor Networks. In *PLDI*, 2007.

[24] J. Leskovec and E. Horvitz. Planetary-Scale Views on an Instant-Messaging Network. In *WWW*, 2008.

[25] M. Lewis and A. Grimshaw. The core Legion object model. In *HPDC*, 1996.

[26] LinkedIn. http://linkedin.com.

[27] H. Liu, T. Roeder, K. Walsh, R. Barr, and E. Sirer. Design and Implementation of a Single System Image Operating System for Ad Hoc Networks. In *MobiSys*, 2005.

[28] Live Mesh. http://www.mesh.com.

[29] Live Messenger. http://messenger.live.com.

[30] B. Livshits and E. Kiciman. Doloto: Code Splitting for Network-bound Web 2.0 Applications. In *SIGSOFT FSE*, 2008.

[31] J. Michel and A. van Dam. Experience with distributed processing on a host/satellite graphics system. *ACM SIGGRAPH Computer Graphics*, 10(2):190–195, 1976.

[32] Microsoft Office Online. http://office.microsoft.com/en-us/office_live/default.aspx.

[33] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden. Wishbone: Profile-based Partitioning for Sensornet Applications. In *NSDI*, 2009.

[34] Quova IP Geo-Location Database. http://www.quova.com.

[35] Rapid Release Cycles. http://www.ereleases.com/pr/20070716009.html.

[36] S. Agarwal and J. Lorch. Matchmaking for Online Games and Other Latency-Sensitive P2P Systems. In *SIGCOMM*, 2009.

[37] M. Shapiro, Y. Gourhant, S. Habert, L. Mosseri, and M. Ruffin. SOS: An object-oriented operating system—assessment and perspectives. *Computing Systems*, 1989.

[38] M. Steen, P. Homburg, and A. Tanenbaum. Globe: A wide-area distributed system. *IEEE Concurrency*, pages 70–78, 1999.

[39] C. Stewart, K. Shen, S. Dwarkadas, M. Scott, and J. Yin. Profile-driven component placement for cluster-based online services. *IEEE Distributed Systems Online*, 5(10):1–1, 2004.

[40] Y. Su and J. Flinn. Slingshot: Deploying Stateful Services in Wireless Hotspots. In *MobiSys*, 2005.

[41] M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A bytecode translator for distributed execution of ”legacy” java software. In *ECOOP*, pages 236–255. Springer-Verlag, 2001.

[42] E. Tilevich and Y. Smaragdakis. J-orchestra: Automatic java application partitioning. In *ECOOP*, pages 178–204. Springer-Verlag, 2002.

[43] Twitter. http://www.twitter.com.

[44] F. Yang, J. Shanmugasundaram, M. Riedewald, and J. Gehrke. Hilda: A High-Level Language for Data-DrivenWeb Applications. *ICDE*, 2006.

[45] Y. Zhao, Y. Xie, F. Yu, Q. Ke, Y. Yu, Y. Chen, and E. Gillum. Botgraph: Large Scale Spamming Botnet Detection. In *NSDI*, 2009.

Optimizing Cost and Performance in Online Service Provider Networks

Zheng Zhang
Purdue University

Ming Zhang
Microsoft Research

Albert Greenberg
Microsoft Research

Y. Charlie Hu
Purdue University

Ratul Mahajan
Microsoft Research

Blaine Christian
Microsoft Corporation

Abstract– We present a method to jointly optimize the cost and the performance of delivering traffic from an online service provider (OSP) network to its users. Our method, called Entact, is based on two key techniques. First, it uses a novel route-injection mechanism to measure the performance of alternative paths that are not being currently used, without disturbing current traffic. Second, based on the cost, performance, traffic, and link capacity information, it computes the optimal cost *vs.* performance curve for the OSP. Each point on the curve represents a potential operating point for the OSP such that no other operating point offers a simultaneous improvement in cost and performance. The OSP can then pick the operating point that represents the desired trade-off (e.g., the "sweet spot"). We evaluate the benefit and overhead of Entact using trace-driven evaluation in a large OSP with 11 geographically distributed data centers. We find that by using Entact this OSP can reduce its traffic cost by 40% without any increase in path latency and with acceptably low overheads.

1 Introduction

Providers of online services such as search, maps, and instant messaging are experiencing an enormous growth in demand. Google attracts over 5 billion search queries per month [2], and Microsoft's Live Messenger attracts over 330 million active users each month [5]. To satisfy this global demand, online service providers (OSPs) operate a network of geographically dispersed data centers and connect with many Internet service providers (ISPs). Different users interact with different data centers, and ISPs help the OSPs carry traffic to and from the users.

Two key considerations for OSPs are the cost and the performance of delivering traffic to its users. Large OSPs such as Google, Microsoft, and Yahoo! send and receive traffic that exceeds a petabyte per day. Accordingly, they bear huge costs to transport data.

While cost is clearly of concern, performance of traffic is critical as well because revenue relies directly on it. Even small increments in user-experienced delay (e.g., page load time) can lead to significant loss in revenue through a reduction in purchases, search queries, or advertisement click-through rates [20]. Because application protocols involve multiple round trips, small increments in path latency can lead to large increments in user-experienced delay.

The richness of OSP networks makes it difficult to optimize the cost and performance of traffic. There are numerous destination prefixes and numerous choices for mapping users to data centers and for selecting ISPs. Each choice has different different cost and performance characteristics. For instance, while some ISPs are free, some are exorbitantly expensive. Making matters worse, cost and performance must be optimized jointly because the trade-off between the two factors can be complex. We show that optimizing for cost alone leads to severe performance degradation and optimizing for performance alone leads to significant cost.

To our knowledge, no automatic traffic engineering (TE) methods exist today for OSP networks. TE for OSPs requires a different formulation than that for transit ISPs or multihomed stub networks. In the traditional intra-domain TE for transit ISPs, the goal is to balance load across multiple internal paths [13, 18, 23]. End-to-end user performance is not considered.

Unlike multihomed stub networks, OSPs can source traffic from any of their multiple data centers. This flexibility adds a completely new dimension to the optimization. Further, large OSPs connect to hundreds of ISPs – two orders of magnitude more than multihomed stub networks – which calls for highly scalable solutions. Another assumption in TE schemes for multihomed sites [7, 8, 15] is that each connected ISP offers paths to all Internet destinations. This assumption is not valid in the OSP context.

Given the limitations of the current TE methods, the state of the art for optimizing traffic in OSP networks is rather rudimentary. Operators manually configure a delicate balance between cost and performance. Because of the complexity of large OSP networks, the operating point thus achieved can be far from desirable.

We present the design and evaluation of Entact, the first TE scheme for OSP networks. We identify and address two primary challenges in realizing such a scheme. First, because the interdomain routing protocol (BGP) does not include performance information, performance is unknown for paths that can be used but are not being currently used. We must estimate the performance of such paths without actually redirecting traffic to them as redirection can be disruptive. We overcome this challenge via a novel *route injection* technique. To measure an unused path for a prefix, Entact selects an IP address ip within the prefix and installs a route for $ip/32$ to routers in the OSP network. Because of the longest-prefix match rule, packets destined to ip will follow the installed route while the rest of the traffic will continue to use the current route.

The second challenge is to use the cost, performance, traffic volume, and link capacity information to find in real time a TE strategy that matches the OSP's goals. Previous algorithmic studies of route selection optimize one of the two metrics, performance or cost, with the other as the fixed constraint. However, from conversations with the operators of a large OSP, we learned that often there is no obvious answer for which metric should be selected as the fixed constraint, as profit depends on the complex trade-off between performance and cost. Entact uses a novel joint optimization technique that finds the entire trade-off curve and lets the operator pick a desirable point on that curve. Such a technique provides operators with useful insight and a range of options for configuring the network as desired.

We demonstrate the benefits of Entact in Microsoft's global network (MSN), one of the largest OSPs today. Because we are not allowed to arbitrarily change the paths used by various prefixes, we conduct a trace-driven study. We implement the key components of Entact and measure the relevant routing, traffic, and performance information. We use this information to simulate Entact-based TE in MSN. We find that compared to the common (manual) practices today, Entact can reduce the total traffic cost by up to 40% without compromising performance. We also find that these benefits can be realized with low overhead. Exploring two closest data centers for each destination prefix and one non-default route at each data center tends to be enough, and changing routes once per hour tends to be enough.

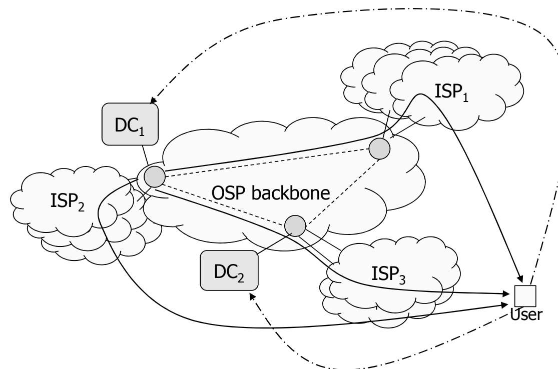


Figure 1: Typical network architecture of a large OSP.

2 Traffic Cost and Performance for OSPs

In this section, we describe the architecture of a typical OSP network. We also outline the unique cost and performance optimization opportunities that arise in OSP networks by exploiting the presence of a diverse set of alternative paths for transporting service traffic.

2.1 OSP network architecture

Figure 1 illustrates the typical network architecture of large OSPs. To satisfy global user demand, such OSPs have data centers (DCs) in multiple geographical locations. Each DC hosts a large number of servers, anywhere from several hundreds to hundreds of thousands. For cost, performance, and robustness, each DC is connected to many ISPs that are responsible for carrying traffic between the OSP and its millions of users. Large OSPs such as Google and Microsoft often also have their own backbone network to interconnect the DCs.

2.2 Cost of carrying traffic

The traffic of an OSP traverses both internal links that connect the DCs and external links that connect to neighboring ISPs. The cost model is different for the two types of links. The internal links are either dedicated or leased. Their cost is incurred during acquisition, and any recurring cost is independent of the traffic volume that they carry. Hence, we can ignore this cost when engineering an OSP's traffic.

The cost of an external link is a function of traffic volume, i.e., $F(v)$, where F is a non-decreasing cost function and v is the charging volume of the traffic. The cost function F is commonly of the form $price \times v$, where $price$ is the unit traffic volume price of a link. The charging volume v is based on actual traffic volume. A common practice is to use the 95th-percentile (*P95*). Under

this scheme, the traffic volume on the link is sampled for every 5-minute interval. At the end of a billing period, *e.g.,* a month, the charging volume is the 95th percentile across all the samples. Thus, the largest 5% of the intervals are not considered, which protects an OSP from being charged for short bursts of traffic.

In principle, the charging volume is the maximum of the P95 traffic in either direction. However, since user requests tend to be much smaller than server replies for online services, the outgoing direction dominates. Hence, we ignore inbound traffic when optimizing the cost of OSP traffic.

2.3 Performance measure of interest

There are several ways to measure the user-perceived performance of an online service. In consultation with OSP operators, we use round trip time (RTT) as the performance measure, which includes the latency between the DC and the end host along both directions. The performance of many online services, such as search, email, maps, and instant messaging, is latency-bound. Small increments in latency can lead to significant losses in revenue [20] .

Some online services may also be interested in other performance measures such as available bandwidth or loss rate along the path. A challenge with using these measures for optimizing OSP traffic is scalable estimation of performance for tens of thousands of paths. Accurate estimation of available bandwidth or loss rate using current techniques requires a large number of probes [17, 19, 25]. We leave for the future the task of extending our work to other performance measures.

2.4 Cost-performance optimization

A consequence of the distributed and rich connectivity of an OSP network is that an OSP can easily have more than a hundred ways to reach a given user in a destination prefix. First, an OSP usually replicates an online service across multiple DCs in order to improve user experience and robustness. An incoming user request can thus be directed to any one of these DCs, *e.g.,* using DNS redirection. Second, the traffic to a given destination prefix can be routed to the user via one of many routes, either provided by one of the ISPs that directly connect to that DC or by one of the ISPs that connect to another DC at another location (by first traversing internal links). Assuming P DCs and an total of Q ISPs, the number of possible *alternative paths* for a request-response round trip is $P * Q$. (An OSP can select which DC will serve a destination prefix, but it typically does not control which link is used by the incoming traffic.)

The large number of possible alternative paths and differences in their cost and performance creates an opportunity for optimizing OSP traffic. This optimization needs to select the target DC and the outgoing route for each destination prefix. The (publicly known) state-of-the-art in optimizing OSP traffic is mostly manual and ad hoc. The default practice is to map a destination prefix to a geographically close DC and to let BGP control the outgoing route from that DC. BGP's route selection is performance-agnostic and can take cost into account in a coarse manner at best. On top of that, exceptions may be configured manually for prefixes that have very poor performance or very high cost.

The complexity of the problem, however, limits the effectiveness of manual methods. Effective optimization requires decisions based on the cost-performance trade-offs of hundreds of thousands of prefixes. Worse, the decisions for various prefixes cannot be made independently because path capacity constraints create complex dependencies among prefixes. Automatic methods are thus needed to manage this complexity. The development of such methods is the focus of our work.

3 Problem Formulation

Consider an OSP as a set of data centers $DC = \{dc_i\}$ and a set of external links $LINK = \{link_j\}$. The DCs may or may not be interconnected with backbone links. The OSP needs to deliver traffic to a set of destination prefixes $D = \{d_k\}$ on the Internet. For each d_k, the OSP has a variety of paths to route the request and reply traffic, as illustrated in Figure 2. A *TE strategy* is defined as a collection of assignments of the traffic (request and reply) for each d_k to a $path(dc_i, link_j)$. Each assignment conceptually consists of two selections, namely *DC selection*, *e.g.,* selecting a dc_i, and *route selection*, *e.g.,* selecting a $link_j$. The assignments are subject to two constraints. First, the traffic carried by an external link should not exceed its capacity. Second, a prefix d_k can use $link_j$ only if the corresponding ISP (which may be a peer ISP instead of a provider) provides routes to d_k.

Each possible TE strategy has a certain level of aggregate performance and incurs certain traffic cost to the OSP. Our goal is to discover the optimal TE strategies that represent the cost-performance trade-offs desired by the OSP. For instance, the OSP might want to maximize performance for a given cost. Additionally, the relevant inputs to this optimization are highly dynamic. Path performance as well as traffic volume of a prefix, which determines cost, change with time. We thus want an efficient, online scheme that adapts the TE strategy as the inputs evolve.

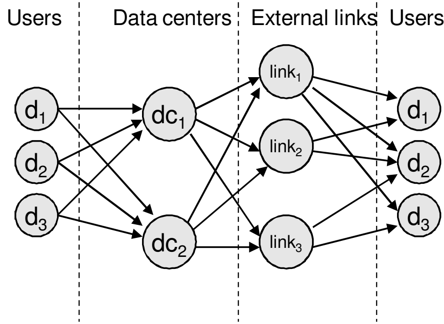


Figure 2: OSP traffic engineering problem.

4 Entact Key Techniques

In this section, we provide an overview of the key techniques in Entact. We present the details of their implementations in the next section. There are two primary challenges in the design of an online TE scheme in a large OSP network. The first challenge is to measure in real time the performance and cost of routing traffic to a destination prefix via any one of its many *alternative paths* that are not currently being used, without actually redirecting the current traffic to those alternative paths. Further, to keep up with temporal changes in network conditions, this measurement must be conducted at sufficiently fine granularity. The second challenge is to use that cost-performance information in finding a TE strategy that matches the OSP's goals.

4.1 Computing cost and performance

To quantify the cost and performance of a TE strategy, we first measure the performance of individual prefixes along various alternative paths. This information is then used to compute the aggregate performance and cost across all prefixes.

4.1.1 Measuring performance of individual prefixes

Our goal is to measure the latency of an alternative path for a prefix with minimal impact on the current traffic, *e.g.,* without actually changing the path being currently used for that prefix. One possible approach is to infer this latency based on indirect measurements. Previous studies have proposed various techniques for predicting the latency between two end points on the Internet [10,14,22,27]. However, they are designed to predict the latency of the current path between two end points in the Internet, and hence are not applicable to our task of measuring alternative paths.

We measure the RTT of alternative paths directly using a novel *route injection* technique. To measure an alternative path which uses a non-default route R for prefix p, we select an IP address ip within p and install the route R for $ip/32$ in the network. This special route is installed to the routers in the OSP by a BGP daemon that maintains iBGP peering sessions with them. Because of the longest-prefix match rule, packets destined to ip will follow the route R and the rest of the traffic will follow the default route. Once the alternative route is installed, we can measure the RTT to p along the route R using data-plane probes to ip (details in §5.1). Simultaneous measurements of multiple alternative paths can be achieved by choosing a distinct IP address for each alternative path.

4.1.2 Computing performance of a TE strategy

The measurements of individual prefixes can be used to compute the aggregate performance of any given TE strategy. We use the weighted average RTT ($wRTT$), $\frac{\sum vol_p \times RTT_p}{\sum vol_p}$, of all the traffic as the aggregate performance measure, where vol_p is the volume of traffic to prefix p, and RTT_p is the RTT of the path to p in the given TE strategy. The traffic volume vol_p is estimated based on the Netflow data collected in the OSP.

4.1.3 Computing cost of a TE strategy

A challenge in optimizing traffic cost is that the actual traffic cost is calculated based on the 95% link utilization over a long billing period (*e.g.,* a month), while an online TE scheme needs to operate at intervals of minutes or hours. While there exist online TE schemes that optimize P95 traffic cost [15], the complexity of such schemes makes them inapplicable to a large OSP network with hundreds of neighbor ISPs. We thus choose to only consider short-term cost in TE optimization rather than directly optimizing P95 cost. Our hypothesis is that, by consistently employing low-cost strategies in each short interval, we can lower the actual traffic cost over the billing period. We present results that validate this hypothesis in §7.

We use a simple computation to quantify the cost of a TE strategy in an interval. As discussed in §2.2, we need to focus only on the external links. For each external link L, we add the traffic volume to all prefixes that choose that link in the TE strategy, *e.g.,* $Vol_L = \sum_p vol_p$, where prefix p uses link L for vol_p amount of traffic. The total traffic cost of the OSP is $\sum_L F_L(Vol_L)$, where $F_L(.)$ is the pricing function of the link L. Because this measure of cost is not the actual traffic cost over the billing period, we refer to this measure as *pseudo cost*.

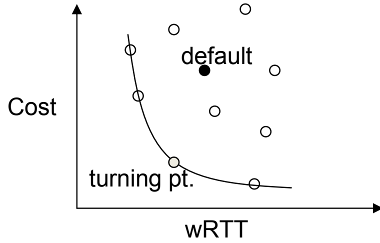


Figure 3: The cost-performance tradeoff in TE strategy space.

4.2 Computing optimal TE strategies

We now present our optimization framework that uses the cost and performance information to derive the desirable TE strategy for an OSP. We first assume the traffic to a destination prefix can be arbitrarily divided among multiple alternative paths and obtain a class of *optimal* TE strategies. In this class of strategies, one cannot improve performance without sacrificing cost or vice versa. Second, we describe how we select a strategy in this class that best matches the cost-performance trade-off that the OSP desires. Third, since in practice the traffic to a prefix cannot be arbitrarily split among multiple alternative paths, we devise an efficient heuristic to find an integral solution that approximates the desired fractional one.

4.2.1 Searching for optimal strategy curve

Given a TE strategy, we can plot its cost and performance (weighted average RTT or $wRTT$) on a 2-D plane. This is illustrated in Figure 3 where each dot represents a strategy. The number of strategies is combinatorial, ${N_p}^{N_a}$ for N_p prefixes and N_a alternative paths per prefix. A key observation is that not all strategies are worth exploring. In fact, we only need to consider a small subset of *optimal* strategies that form the lower-left boundary of all the dots on the plane. A strategy is optimal if no other strategy has both lower $wRTT$ and lower cost. Effectively, the curve connecting all the optimal strategies forms an *optimal strategy curve* on the plane.

To compute this curve, we sweep from a lower bound on possible $wRTT$ values to an upper bound on possible $wRTT$ values at small increments, *e.g.,* 1 ms, and compute the minimum cost for each $wRTT$ value in this range. These bounds are set loosely, *e.g.,* the lower bound can be zero and the upper bound can be ten times the $wRTT$ of the default strategy.

Given a $wRTT$ R in this range, we compute the minimum cost using linear programing (LP). Following the notations in Figure 2, let f_{kij} be the fraction of traffic to d_k that traverses $path(dc_i, link_j)$ and rtt_{kij} be the RTT to d_k via $path(dc_i, link_j)$. The problem of computing cost can then be described as:

$$\min\ pseudoCost = \sum_j (price_j \times \sum_k \sum_i (f_{kij} \times vol_k)),$$

subject to:

$$\sum_k \sum_i (f_{kij} \times vol_k) \leq \mu \times cap_j \quad (1)$$

$$\sum_k \sum_i \sum_j (f_{kij} \times vol_k \times rtt_{kij}) \leq \sum_k vol_k \times R \quad (2)$$

$$\sum_i \sum_j f_{kij} = 1 \quad (3)$$

$$\forall k, i, j \quad 0 \leq f_{kij} \leq 1 \quad (4)$$

Condition 1 represents the capacity constraint for each external link and μ is a constant (by default 0.95) that reserves some spare capacity to accommodate potential traffic variations for online TE. Condition 2 represents the $wRTT$ constraint. Condition 3 ensures all the traffic to a destination is carried. The objective is to find feasible values for variables f_{kij} that minimize the total pseudo cost. Solving such an LP for all possible values of R and connecting the TE strategy points thus obtained yield the optimal strategy curve.

4.2.2 Selecting a desirable optimal strategy

Each strategy on the optimal strategy curve represents a particular tradeoff between performance and cost. Based on its desired tradeoff, an OSP will typically be interested in one or more of these strategies. Some of these strategies are easy to identify, such as minimum cost for a given performance or minimum wRTT for a given cost budget. Sometimes, an OSP may desire a more complex tradeoff between cost and performance. For such an OSP, we take a parameter K as an input. This parameter represents the additional unit cost the OSP is willing to bear for a unit decrease in wRTT.

The desirable strategy for a given K corresponds to the point in the optimal strategy curve where the slope of the curve becomes higher than K when going from right to left. More intuitively, this point is also the "turning point" or the "sweet spot" when the optimal strategy curve is plotted after scaling the wRTT by K. We can automatically identify this point along the curve as the one with the minimum value of $pseudoCost + K \cdot wRTT$.

This point is guaranteed to be unique because the optimal strategy curve is convex. For convenience, we define $pseudoCost + K \cdot wRTT$ as the $utility$ of a strategy. Lower utility values are better. We can directly find this turning point by slightly modifying the original optimization problem to minimize utility instead of by solving the original optimization problem for all possible wRTT values.

4.2.3 Finding a practical strategy

The desirable strategy identified above assumes that traffic to a prefix can be split arbitrarily across multiple paths. In practice, however, the traffic to a prefix can only take one alternative path at a time, and hence variables f_{kij} must be either 0 or 1. Imposing this requirement makes the optimization problem an Integer Linear Programming (ILP) problem, which is NP-hard. We devise a heuristic to approximate the fractional solution to an optimal strategy with an integral solution. Intuitively, our heuristic searches for an integral solution "near" the desired fractional one.

We start with the fractional solution and sort all the destination prefixes d_k in the ascending order based on $avail_k = \sum_{j \in R_k} \lfloor \frac{availCap_j}{vol_k} \rfloor$, where vol_k is the traffic volume to d_k, R_k is the set of external links that have routes to reach d_k, and $availCap_j$ is the available capacity at $link_j$. The $availCap_j$ is initialized to be the capacity of $link_j$ and updated each time a prefix is assigned to use this link. The $avail_k$ measure gives high priority to prefixes with large traffic volume and small available capacity. We then greedily assign the prefixes to paths in the sorted order.

Given a destination d_k and its corresponding f_{kij}'s in the fractional solution, we randomly assign all of its traffic to one of the paths $path(dc_i, link_j)$ that has enough residual capacity for d_k with a probability proportional to f_{kij}. Compared to assigning the traffic to the path with the largest f_{kij}, random assignment is more robust to a bad decision for one particular destination. Once a prefix is assigned, the available capacity of the selected link is adjusted accordingly, and the $avail_k$-based ordering of the remaining unassigned prefixes is updated as well. In theory, better integral solutions can be obtained using more sophisticated methods [26]. But as we show later, our simple heuristic approximates the fractional solution closely.

5 Prototype Implementation

In this section, we describe our implementation of Entact. As shown in Figure 4, there are three inputs to Entact. The first input is Netflow data from all routers in the OSP network, which gives us information on flows currently traversing the network. The second input is routing tables from all routers, which gives us information not only on routes currently being used and but also on alternative routes offered by neighbor ISPs. The third input is the information on link capacities and prices. The output of Entact is a recommended TE strategy.

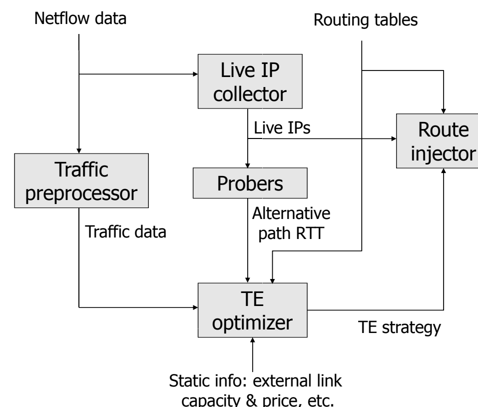


Figure 4: The Entact architecture

Entact divides time into fixed-length windows of size TE_{win} and a new output is produced in every window. To compute the TE strategy in window i, the measurements of traffic volume and path performance from the previous window are used. We assume that these quantities change at a rate that is much slower than TE_{win}. We later validate this assumption and also evaluate the impact of TE_{win}. The recommended TE strategy is applied to the OSP network by injecting the selected routes, similar to the route injection of /32 IP addresses.

5.1 Measuring path performance

As mentioned before, to obtain measurements on the performance of alternative paths to a prefix, we inject special routes to IP addresses in that prefix and then measure performance by sending probes to those IP addresses. We identify IP addresses within a prefix that respond to our probes using the *Live IP collector* component (Figure 4). The *Route Injector* component injects routes to those IP addresses, and the *Probers* measure the path performance. We describe each of these components below.

Live IP collector. Live IP collector is responsible for efficiently discovering IP addresses in a prefix that respond to our probes. A randomly chosen IP address in a prefix is unlikely to be responsive. We use a combination of two methods to discover live IP addresses. The first method is to probe a subset of IP addresses that are found in Netflow data. The second method is the heuristic proposed in [28]. This heuristic prioritizes and orders probes to a

small subset of IP addresses that are likely to respond, *e.g.,* *.1 or *.127 addresses, and hence is more efficient than random scanning of IP addresses.

Discovering one responsive IP address in a prefix is not enough; we need multiple IP addresses to probe multiple paths simultaneously and also to verify if the prefix is in a single geographical location (see §6.1). Even the combination of our two methods does not always find enough responsive IP addresses for every Internet prefix. In this paper, we restrict ourselves to those prefixes for which we can find enough responsive IP addresses. We show, however, that our results likely apply to all prefixes. In the future, we plan to overcome this responsive IP limitation by enlisting user machines, *e.g.,* through browser toolbars.

Route injector. Route injector selects alternative routes from the routing table obtained from routers in the OSP network, and installs the selected alternative routes on the routers. The route injector is a BGP daemon that maintains iBGP session with all core and edge routers in the OSP network. The daemon dynamically sends and withdraws crafted routes to those routers. We explain the details of the injection process using a simple example. We denote a path for a prefix p from data center DC as $path(DC, egress - nexthop)$, where $egress$ is the OSP's edge router along the path, and $nexthop$ is the ISP's next hop router that is willing to forward traffic from $egress$ to p. In Figure 5, suppose the default BGP route of p follows $path(DC, E_1 - N_1)$ and we have two other alternative paths. Given an IP address IP_2 within p, to measure an alternative path $path(DC, E_2 - N_2)$ we do the following,

- Inject $IP_2/32$ with nexthop as E_2 into all the core routers C_1, C_2, and C_3
- Inject $IP_2/32$ with nexthop as N_2 into E_2.

Now, traffic to IP_2 will traverse the alternative path that we want to measure, while all traffic to other IP addresses in p, *e.g.,* IP_1, will still follow the default path. Similarly, we can inject another IP address $IP_3/32$ within p and simultaneously measure the performance of the two alternative paths. With n IP addresses in a prefix, we can simultaneously measure the performance of n alternative paths from each DC. The route injection only needs to be performed once. The injected routes are re-used across all TE windows, and updated only when there are routing changes. If more than n paths need to be measured, we can divide a TE window into smaller slots, and measure only n paths in each slot. In this case, the route injector needs to refresh the injected routes for each slot.

We implement the daemon that achieves the above functionality by feeding configuration commands to

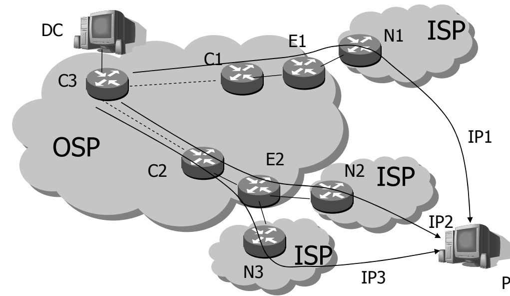


Figure 5: Route injection in a large OSP network.

drive `bgpd`, an existing BGP daemon [3]. We omit implementation details due to space limit. It is important, however, to note that the core and edge routers should be configured to keep the injected routes only to themselves. Therefore, route injection does not encounter route convergence problems, or trigger any route propagation in or outside the OSP network.

Probers. Probers are located at all data centers in the OSP network and probe the live IPs along the selected alternative paths to measure their performance. For each path, a prober takes five RTT samples and uses the median as the representative estimate of that path. The probing module sends a TCP ACK packet to a random high port of the destination. This will often trigger the destination to return a TCP RST packet. Compared with using ICMP probes, the RTT measured by TCP ACK/RST is closer to the latency experienced by applications because ICMP packets may be forwarded in the network with lower priority [16].

5.2 Computing TE strategy

The computation of the TE strategy is based on the path performance data, the prefix traffic volume information, and the desired operating point of the OSP. The prefix traffic volume is computed by the *traffic preprocessor* component in Figure 4. It uses Netflow streams from all core routers and computes the traffic volume to each prefix by mapping each destination IP address to a prefix. For scalability, the Netflow data in our implementation is sampled at the rate of 1/1000.

Finally, the *TE optimizer* component implements the optimization process described in §4.2. It uses MOSEK [6] to solve the LP problems required to generate the optimal strategy. After identifying the optimal fractional strategy, the optimizer converts it to an integer strategy which becomes the output of the optimization process.

Figure 6: Location of the 11 DCs used in experiments.

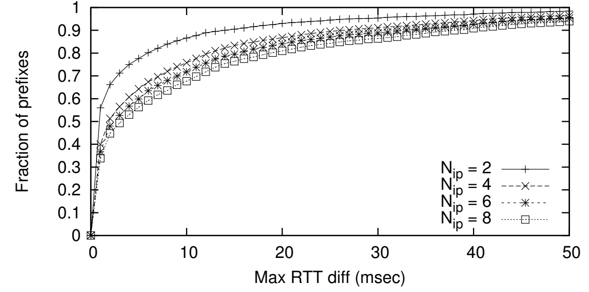


Figure 7: Maximum RTT difference among N_{ip} IPs within a prefix.

Region	N.Amer.	Europe	A.Pac.	Lat.Amer.	Africa
%prefix	58	28	8	5	< 1
%traffic	59	29	6	6	< 1

Table 1: Locations of the 6K prefixes in our experiments.

6 Experimental Setup

We conduct experiments in Microsoft's global network (MSN), one of the largest OSPs today. Figure 6 shows the location of the 11 MSN DCs that we use. These DCs span North America, Europe, and Asia Pacific and are inter-connected with high-speed dedicated and leased links that form the backbone of MSN. MSN has roughly 2K external links, many of which are free peering because that helps to lower transit cost for both MSN and its neighbors. The number of external links per DC varies from fewer than ten to several hundreds, depending on the location. We assume that services and corresponding user data are replicated to all DCs. In reality, some services may not be present at some of the the DCs. The remainder of this section describes how we select destination prefixes and how we quantify the performance and cost of a TE strategy.

6.1 Targeted destination prefixes

To reduce the overhead of TE, we focus on the high-volume prefixes that carry the bulk of traffic and whose optimization has significant effects on the aggregate cost and performance. We start with the top 30K prefixes which account for 90% of the total traffic volume. A large prefix advertised in global routing sometimes spans multiple geographical locations [21]. We could handle multi-location prefixes by splitting them into smaller sub-prefixes. However, as explained below, we would need enough live IP addresses in each sub-prefix to determine whether a sub-prefix is single-location or not. Due to the limited number of live IP addresses we can discover for each prefix (§5.1), we bypass the multi-location or low-volume prefixes in this paper.

We consider a prefix to be at a single location if the difference between the RTTs to any pair of IP addresses in it is under 5 ms. This is the typical RTT value between two nodes in the same metropolitan region [21]. A key parameter in this method is N_{ip}, the number of live IP addresses to which the RTTs are measured. On the one hand, we need to measure enough live IP addresses in order not to mis-classify a multi-location prefix as a single-location one. On the other hand, we can only identify a limited number of live IP addresses in a prefix.

To choose an appropriate N_{ip}, we examine the 4.1K prefixes that have at least 8 live IP addresses. Figure 7 illustrates the distributions of the maximum RTT difference of each of these prefixes as N_{ip} varies from 2 to 8. While the gap is significant between the distributions of N_{ip}=2 and N_{ip}=4, it becomes less pronounced as N_{ip} increases beyond 4. There is only an 8% difference between the distributions of N_{ip}= 4 and N_{ip}=8 when the maximum RTT difference is 5 ms. We thus pick $N_{ip} = 4$ to balance the accuracy of single-location prefix identification and the number of prefixes available for use.

After discarding prefixes with fewer than 4 live IP addresses, we are left with 15K prefixes. After further discarding prefixes that are deemed multi-location, we are left with 6K prefixes which we use in our study. Table 1 characterizes these prefixes by continents and traffic volumes. While a large portion of the prefixes and traffic are from North America and Europe, we also have some coverage in the remaining three continents. The prefixes are in 2,791 distinct ASes and account for 26% of the total MSN traffic. The number of alternative routes for a prefix varies at different DC locations. Among the 66K DC-prefix pairs, 61% have 1 to 4 routes, 27% has 5 to 8 routes, and the remaining 11% has more than 8 routes.

Our focus on a subset of prefixes raises two questions. First, are the results based on these prefixes applicable to

all prefixes? Second, how should we appropriately scale link capacities? We consider both questions next.

6.1.1 Representativeness of selected prefixes

We argue that the subset of prefixes that we study lets us estimate well the cost-performance trade-off for all traffic carried by MSN. For a given set of prefixes, the benefits of TE optimization hinge on the existence of alternative paths that are shorter or cheaper than the one used in the default TE strategy. We find that in this respect our chosen set of prefixes (P_s) is similar to other prefixes. We randomly select 14K high-volume prefixes (P_h) and 4K low-volume prefixes (P_l), which account for 29% and 0.8% of the total MSN traffic respectively. For each prefix p in P_h or P_l, we can identify 2 live IP addresses at the same location (with RTT difference under 5 ms). This means at least some sub-prefix of p will be at a single-location, even though p could span multiple locations.

For each prefix in P_s, P_h and P_l, we measure the RTT of the default route and three other randomly selected alternative routes from all the 11 DCs every 20 minutes for 1 day. We compare the default path used by the default TE strategy, *e.g.,* the path chosen by BGP from the closest DC, with all other 43 (may be fewer due to the availability of routes) alternative paths. Figure 8 illustrates the number of alternative paths that are better than the default path in terms of (a) performance, (b) cost, or (c) both. We see that the distributions are similar for the three sets of prefixes, which suggests that each set has similar cost-performance trade-off characteristics. Thus, our TE optimization results based on P_s are likely to hold for other traffic in MSN.

6.1.2 Scaling link capacity

Each external link has a fixed capacity that limits the traffic volume that it can carry. We extract link capacities from router configuration files in MSN. Because we only study a subset of prefixes, we must appropriately scale link capacities for our evaluation.

Let P_{all} and P_s denote the set of all the prefixes and the set of prefixes that we study. One simple approach is to scale down the capacity of all links by a constant $ratio = \frac{vol_{all}}{vol_s}$, where vol_{all} and vol_s are the traffic volumes of the two set of prefixes in a given period. The problem with this approach is that it overlooks the spatial and temporal variations of traffic, since $ratio$ actually depends on which link or which period we consider. This prompts us to compute a $ratio$ for each link separately. Our observation is that a link is provisioned for certain utilization level during peak time. Given $link_j$, we set $ratio_j = \frac{peak_j^{all}}{peak_j^{s}}$, where $peak_j^{all}$ and $peak_j^{s}$ are the peak traffic volume to P_{all} and to P_s under the default TE strategy during any 5-minute interval. This ensures the peak utilization of $link_j$ is the same before and after scaling. Note that $peak_{all}$ and $peak_s$ may occur in different 5-minute intervals.

Our method for scaling down link capacity is influenced by the default TE strategy. For instance, if $link_j$ never carries traffic to any prefix in P_s in the default strategy, its capacity will be scaled down to zero. This limits the alternative paths that can be explored in TE optimization, *e.g.,* any alternative strategies that use $link_j$ will not be considered even though they may help to lower wRTT and/or cost. Due to this limitation, our results, which show significant benefits for an OSP, actually represent a lower bound on the benefits achievable in practice.

6.2 Quantifying performance and cost

To quantify the cost of a given TE strategy, we record the traffic volume to each prefix and compute the traffic volume on each external link in each 5-minute interval. We then use this information to compute the 95% traffic cost (P95) over the entire evaluation period. Thus, even though Entact does not directly optimize for P95 cost, our evaluation measures the cost that the OSP will bear under the P95 scheme. We consider only the P95 scheme in our evaluation because it is the dominant charging model in MSN. Some ISPs do offer other charging models, such as long-term flat rate. Some ISPs also impose penalties if traffic volume falls below or exceeds a certain threshold. We leave for future work evaluating Entact under non-P95 schemes.

To quantify the performance, we compute the wRTT for each 5-minute interval and take the weighted average across the entire evaluation period. A minor complication is that we do not have fine time-scale RTT measurements for a prefix. To control overhead of active probing and route injection, we obtain two measurements (where each measurement is based on sending 5 RTT probes) in a 20-minute interval.

We find, however, that these coarse time-scale measurements are a good proxy for predicting finer time-scale performance. To illustrate this, we randomly select 500 prefixes and 2 alternate routes for each selected prefix. From each DC, we measure each of these 1,000 paths once a minute during a 20-minute interval. We then divide the interval into four 5-minute intervals. For each path and a 5-minute interval, we compute $\overline{rtt}_5$ by averaging the 5 measurements in that interval. For the same path, we also compute $\widetilde{rtt}_{20}$ by averaging two randomly selected measurements in the 20-minute interval. We conduct this experiment for 1 day and calculate the

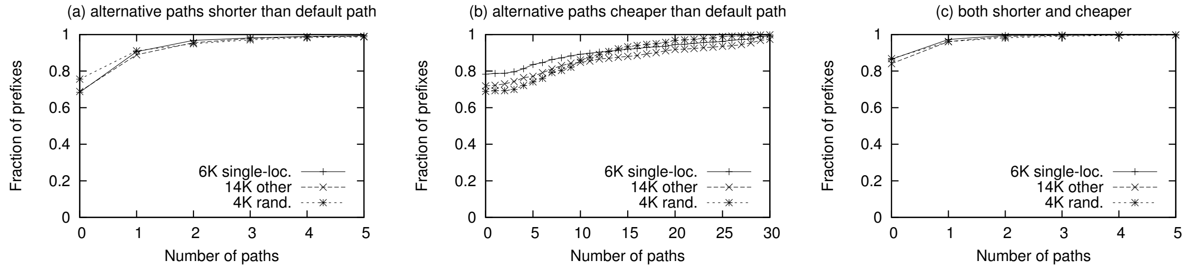


Figure 8: Number of alternative paths that are better than the default path in the set of 6K single-location prefixes, the set of 14K other high-volume prefixes, and the set of 4K randomly selected low-volume prefixes.

difference between $\overline{rtt}_5$ and $\widetilde{rtt}_{20}$ of all paths. It turns out that $\widetilde{rtt}_{20}$ are indeed very close to $\overline{rtt}_5$. The difference is under 1 ms and 5 ms in 78% and 92% of the cases respectively.

7 Results

In this section, we demonstrate and explain the benefits of online TE optimization in MSN. We also study how the TE optimization results are affected by a few key parameters in Entact, including the number of DCs, number of alternative routes, and TE optimization window. Our results are based on one-week of data collected in September 2009, which allows us to capture the time-of-day and day-of-week patterns. Since the traffic and performance characteristics in MSN are usually quite stable over several weeks, we expect our results to be applicable to longer duration as well.

Currently, the operators of MSN only allow us to inject /32 prefixes into the network in order to restrict the impact of Entact on customer traffic. As a result, we have limited capability in implementing a non-default TE strategy since we cannot arbitrarily change the DC selection or route selection for any prefix. Instead, we can only simulate a non-default TE strategy based on the routing, performance and traffic data collected under the default TE strategy in MSN. When presenting the following TE optimization results, we assume that the routing, performance and traffic to each prefix do not change under different TE strategies. This is a common assumption made by most of the existing work on TE [9, 12, 15]. We hope to study the effectiveness of Entact without such restrictions in the future.

7.1 Benefits of TE optimization

Figure 9 compares the wRTT and cost of four TE strategies, including the default, Entact_{10} ($K = 10$), Lowest-Cost (minimizing cost with $K = 0$), and BestPerf (minimizing wRTT with $K = inf$). We use 20-minute TE window and 4 alternative routes from each DC for TE optimization. The x-axis is the wRTT in milliseconds and the y-axis is the relative cost. We cannot reveal the actual dollar cost for confidentiality reason. There is a big gap between the default strategy and Entact_{10}, which indicates the former is far from optimal. In fact, Entact_{10} can reduce the default cost by 40% without inflating wRTT. This could lead to enormous amount of savings for MSN since it spends tens of millions of dollars a year on transit traffic cost.

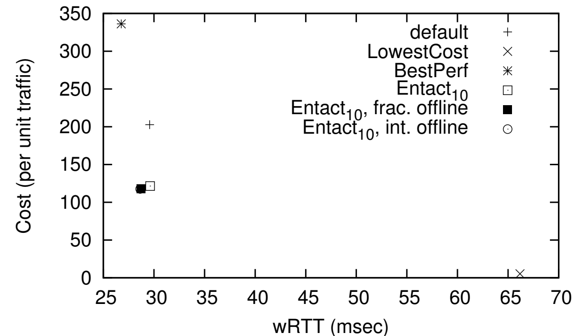


Figure 9: Comparison of various TE strategies.

We also notice there is significant tradeoff between cost and performance among the optimal strategies. In one extreme, the LowestCost strategy can eliminate almost all the transit cost by diverting traffic to free peering links. But this comes at the expense of inflating the default wRTT by 38 ms. Such a large RTT increase will notably degrade user-perceived performance when amplified by the many round trips involved in downloading content-rich Web pages. In the other extreme, the BestPerf strategy can reduce the default wRTT by 3 ms while increasing the default cost by 66%. This is not an appropriate strategy either given the relatively large cost increase and small performance gain. Entact_{10} appears to be at a “sweet-spot” between the two extremes. By exposing the performance and cost of various opti-

path type	prefix	wRTT (ms)	pseudo cost
same	88.2%	29.6	41.1
pricier, longer	0.1%	74.5→75.1	195.1→195.1
pricier, shorter	4.6%	44.7→30.2	13.7→55.8
cheaper, longer	5.5%	27.6→39.8	738.3→177.8
cheaper, shorter	1.7%	55.5→47.8	483.7→174.4

Table 2: Comparison of paths under the default and Entact_{10} strategies in terms of performance and cost.

path type	prefix
non-default DC, default route	2.1%
non-default DC, non-default route	2.5%
default DC, non-default route	7.2%

Table 3: Comparison of paths under the default and Entact_{10} strategies in terms of DC selection and route selection.

mal strategies, the operators can make a more informed decision regarding which is a desirable operating point.

To better understand the source of the improvement offered by Entact_{10}, we compare Entact_{10} with the default strategy during a typical 20-minute TE window. Table 2 breaks down the prefixes based on their relative pseudo cost and performance under these two strategies. Overall, the majority (88.2%) of the prefixes are assigned to the default path in Entact_{10}. Among the remaining prefixes, very few (0.1%) use a non-default path that is both longer and pricier than the default path (which is well expected). Only a small number of prefixes (1.7%) use a non-default path that is both cheaper and shorter. In contrast, 10.1% of the prefixes use a non-default path that is better in one metric but worse in the other. This means Entact_{10} is actually making some "intelligent" performance-cost tradeoff for different prefixes instead of simply assigning each prefix to a "better" non-default path. For instance, 4.6% of the prefixes use a shorter but pricier non-default path. While this slightly increases the pseudo cost by 42.1, it helps to reduce the wRTT of these prefixes by 14.5 ms. More importantly, it frees up the capacity on some cheap peering links which can be used to carry traffic for certain prefixes that incur high pseudo cost under the default strategy. 5.5% of the prefixes use a cheaper but longer non-default path. This helps to drastically cut the pseudo cost by 560.5 at the expense of a moderate increase of wRTT (12.2 ms) for these prefixes. Note that Entact_{10} may not find a free path for every prefix due to the performance and capacity constraints. The complexity of the TE strategy within each TE window and the dynamics of TE optimization across time underscore the importance of employing an automated TE scheme like Entact in a large OSP.

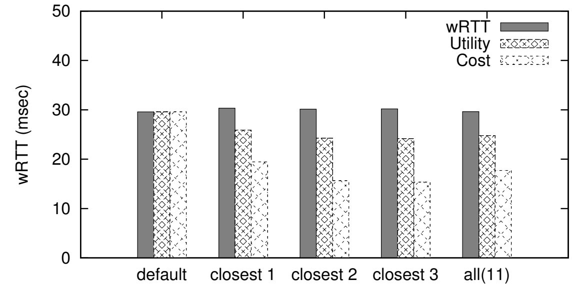


Figure 10: Effect of DC selection on TE optimization. (Utility and cost are scaled according to wRTT of the default strategy.)

Table 3 breaks down the prefixes that use a non-default path under Entact_{10} during the 20-minute TE window by whether a non-default DC or a non-default route from a DC is used. Both non-default DCs and non-default routes are used under Entact_{10} — 4.6% of the prefixes use a non-default DC and 9.7% of them use a non-default route from a DC. Non-default routes appear to be more important than non-default DCs in TE optimization. We will further study the effect of DC selection and route selection in §7.2 and §7.3.

Figure 9 shows that the difference between the integral and fractional solutions of Entact_{10} is negligibly small. In TE optimization, the traffic to a prefix will be split across multiple alternative paths only when some alternative paths do not have enough capacity to accommodate all the traffic to that prefix. This seldom happens because the traffic volume to a prefix is relatively small compared to the capacity of a peering link in MSN.

We also compare the online Entact_{10} with the offline one. In the latter case, we directly use the routing, performance, and traffic volume information of a 20-minute TE window to optimize TE in the same window. This represents the ideal case where there is no prediction error. Figure 9 shows the online Entact_{10} incurs only a little extra wRTT and cost compared to the offline one (The two strategy points almost completely overlap). This is because the RTT and traffic to most of the prefixes are quite stable during such a short period (*e.g.,* 20 minutes). We will study to what extent the TE window affects the optimization results in §7.4.

7.2 Effects of DC selection

We now study the effects of DC selection on TE optimization. A larger number of DCs will provide more alternative paths for TE optimization, which in turn should lead to better improvement over the default strategy.

Nonetheless, this will also incur greater overhead in RTT measurement and TE optimization. We want to understand how many DCs are required to attain most of the TE optimization benefits. For each prefix, we sort the 11 DCs based on the RTT of the default route from each DC. We only use the RTT measurements taken in the first TE window of the evaluation period to sort the DCs. The ordering of the DCs should be quite stable and can be updated at a coarse-granularity, *e.g.,* once a week. We develop a slightly modified Entact$_k^n$ which only considers the alternative paths from the closest n DCs to each prefix for TE optimization.

Figure 10 compares the wRTT, cost, and utility (§4.2.2) of Entact$_{10}^n$ as n varies from 1 to 11. We use 4 alternative routes from each DC to each prefix. Given a TE window, as n changes, the optimal strategy curve and the optimal strategy selected by Entact$_{10}^n$ will change accordingly. This complicates the comparison between two different Entact$_{10}^n$'s since one of them may have higher cost but smaller wRTT. For this reason, we focus on comparing the utility for different values of n. As shown in the figure, Entact$_{10}^1$ (only with route selection but no DC selection) and Entact$_{10}^2$ can cut the utility by 12% and 18% respectively compared to the default strategy. The utility reduction diminishes as n exceeds 2. This suggests that TE optimization benefits can be attributed to both route selection and DC selection. Moreover, selecting the closest two DCs for each prefix seems to attain almost all the TE optimization benefits. Further investigation reveals that most prefixes have at most two nearby DCs. Using more DCs generally will not help TE optimization because the RTT from those DCs is too large.

Note that the utility of Entact$_{10}^{11}$ is slightly higher than that of Entact$_{10}^2$. This is because the utility of Entact$_k^n$ is computed from the 95% traffic cost during the entire evaluation period. However, Entact$_k^n$ only minimizes pseudo utility computed from pseudo cost in each TE window. Even though the pseudo utility obtained by Entact$_k^n$ in a TE window always decreases as n grows, the utility over the entire evaluation period may actually move in the opposite direction.

7.3 Effects of alternative routes

We evaluate how TE optimization is affected by the number of alternative routes (m) from each DC. A larger m will not only offer more flexibility in TE optimization but also incur greater overhead in terms of route injection, optimization, and RTT measurement. In this experiment, we measure the RTT of 8 alternative routes from each DC to each prefix every 20 minutes for 1 day. Figure 11 illustrates the wRTT, cost, and utility of Entact$_{10}$ under different m. For the same reason as in the previous section, we focus on comparing utility. As m grows

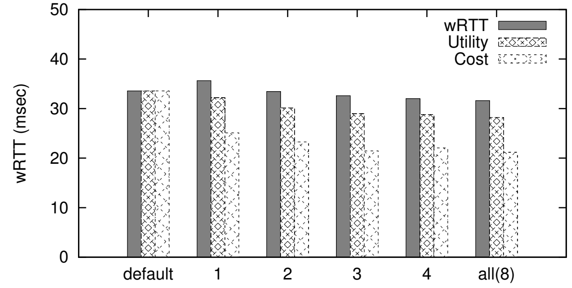


Figure 11: Effect of the number of alternative routes on TE optimization.

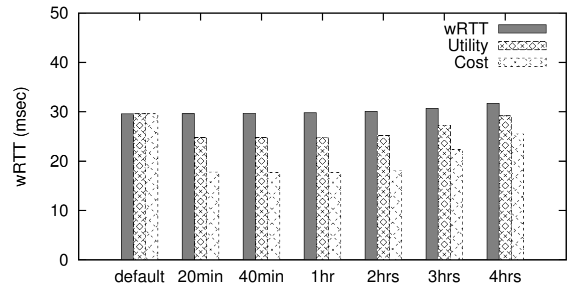


Figure 12: Effect of the TE window on TE optimization.

from 1 to 3, the utility gradually decreases up to 14% compared to the default strategy. The utility almost remains the same after m exceeds 3. This suggests that 2 to 3 alternative routes are sufficient for TE optimization in MSN.

7.4 Effects of TE window

Finally, we study the impact of TE window on optimization results. Entact performs online TE in a TE window using predicted performance and traffic information (§5). On the one hand, both performance and traffic volume can vary significantly within a large TE window. It will be extremely difficult to find a fixed TE strategy that performs well during the entire TE window. On the other hand, a small TE window will incur high overhead in route injection, RTT measurement, and TE optimization. It may even lead to frequent user-perceived performance variations.

Figure 12 illustrates the wRTT, cost, and utility of Entact$_{10}$ under different TE window sizes from 20 minutes to 4 hours. As before, we focus on comparing the utility. We still use 4 alternative routes from each DC to each prefix. Entact$_{10}$ can attain about the same utility reduction compared to the default strategy when the TE

# routes	injection time (sec)	CPU (%)	RIB (MB)	FIB (MB)
5,000	9	3	0.81	0.99
10,000	15	2	1.61	1.72
20,000	30	3	3.22	3.18
30,000	51	4	4.84	4.64
50,000	73	7	8.06	7.57
100,000	147	17	16.12	14.88

Table 4: Route injection overhead measured on a testbed.

window is under 1 hour. This is because the performance and traffic volume are relatively stable during such time scale. As the TE window exceeds 1 hour, the utility noticeably increases. With a 4-hour TE window, Entact_{10} can only reduce the default utility by 1%. In fact, because the traffic volume can fluctuate over a wide range during 4 hours, Entact_{10} effectively optimizes TE for the peak interval to avoid link congestion. This leads to a sub-optimal TE strategy for many non-peak intervals. In §8, we show that an 1-hour TE window imposes reasonably low overhead.

8 Online TE Optimization Overhead

So far, we have demonstrated the benefits provided by Entact. In this section, we study the feasibility of deploying Entact to perform full-scale online TE optimization in a large OSP. The key factor that determines the overheads of Entact is the number of prefixes. While there are roughly 300K Internet prefixes in total, we will focus on the top 30K high-volume prefixes that account for 90% of the traffic in MSN (§6.1). Multi-location prefixes may inflate the actual number of prefixes beyond 30K; we leave the study of multi-location prefixes as future work. The results in §7.3 and §7.4 suggest that Entact can attain most of the benefits by using 2 alternative routes from each DC and an 1-hour TE window. We now evaluate the performance and scalability of key Entact components under these settings.

8.1 Route injection

We evaluate the route injection overhead by setting up a router testbed in the Schooner lab [4]. The testbed comprises a Cisco 12000 router and a PC running our route injector. Cisco 12000 routers are commonly used in the backbone network of large OSPs. When Entact initializes, it needs to inject 30K routes into each router in order to measure the RTT of the default route and one non-default route simultaneously. This injection process can be spread over several days to avoid overloading routers. Table 4 shows the size of the RIB (routing information base) and FIB (forwarding information base) as the number of injected routes grows. 30K routes merely occupy about 4.8 MB in the RIB and FIB. Such memory overhead is relatively small given that today's routers typically hold roughly 300K routes (the number of all Internet prefixes).

After the initial injection is done, Entact needs to continually inject routes to apply the output of the online TE optimization. Table 4 also shows the injection time of different number of routes. It takes only 51 seconds to inject 30K routes, which is negligibly small compared to the 1-hour TE window. We expect the actual number of injected routes in a TE window to be much smaller because most prefixes will simply use a default route (§7.1).

8.2 Impact on downstream ISPs

Compared to the default TE strategy, the online TE optimization performed by Entact may cause traffic to shift more frequently. This is because Entact needs to continually adapt to changes in performance and traffic volume in an OSP. A large traffic shift may even overload certain links in downstream ISPs, raising challenges in the TE of these downstream ISPs. This problem may exacerbate if multiple large OSPs perform such online TE optimization simultaneously. Given a 5-minute interval i, we define a *total traffic shift* to quantify the impact of an online TE strategy on downstream ISPs:

$$TotalShift_i = \sum_p shift_i(p) / \sum_p vol_i(p)$$

Here, $vol_i(p)$ is the traffic volume to prefix p and $shift_i(p)$ is the traffic shift to p in interval i. If p stays on the same path in intervals i and $i-1$, $shift_i(p)$ is computed as the increase of $vol_i(p)$ over $vol_{i-1}(p)$. Otherwise, $shift_i(p) = vol_i(p)$. In essence, $shift_i(p)$ captures the additional traffic load imposed on downstream ISPs relative to the previous interval. The additional traffic load is either due to path change or due to natural traffic demand growth.

Figure 13 compares the $TotalShift$ under the static TE strategy, the default TE strategy, and Entact_{10} over the entire evaluation period. In the static strategy, the TE remains the same across different intervals, and its traffic shift is entirely caused by natural traffic demand variations. We observe that most of the traffic shift is actually caused by natural traffic demand variations. The traffic shift of Entact_{10} is only slightly larger than that of the default strategy. As explained in §7.1, Entact_{10} assigns a majority of the prefixes to a default path and only reshuffles the traffic to roughly 10% of the prefixes. Moreover, the paths of those 10% prefixes do not always

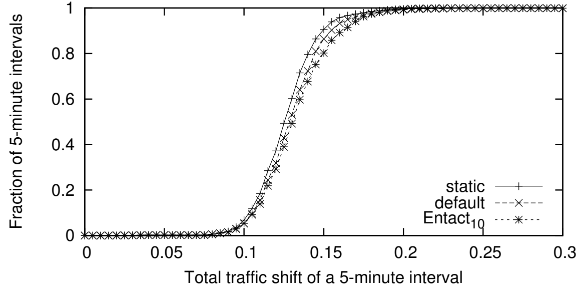


Figure 13: Traffic shift under the static, default, and Entact$_{10}$ TE strategies

change across different intervals. As a result, Entact$_{10}$ incurs limited extra traffic shift compared to the default strategy.

8.3 Computation time

Entact$_k$ computes an optimal strategy in two steps: i) solving an LP problem to find a fractional solution; ii) converting the fractional solution into an integer one. Let n be the number of prefixes, d be the number of DCs, and l be the number of peering links. The number of variables f_{ijk} in the LP problem is $n \times d \times l$. Since d and l are usually much smaller than n and do not grow with n, we consider the size of the LP problem to be $O(n)$. The worst case complexity of an LP problem is $O(n^{3.5})$ [1]. The heuristic for converting the fractional solution into an integer one (§4.2.3) requires n iterations to assign n prefixes. In each iteration, it takes $O(n \log(n))$ to sort the unassigned prefixes in the worst case. Therefore, the complexity of this step is $O(n^2 \log(n))$.

We evaluate the time to solve the LP problem since it is the computation bottleneck in TE optimization. We use Mosek [6] as the LP solver and measure the optimization time of one TE window on a Windows Server 2008 machine with two 2.5 GHz Xeon processors and 16 GB memory. We run two experiments using the top 20K high-volume prefixes and all the 300K prefixes respectively. The RTTs of the 20K prefixes are from real measurement while the RTTs of the 300K prefixes are generated based on the RTT distribution of the 20K prefixes. We consider 2 alternative routes from each of the 11 DCs to each prefix. The traffic volume, routing, and link price and capacity information are directly drawn from the MSN dataset. The running time of the two experiments are 9 and 171 seconds respectively, representing a small fraction of an 1-hour TE window.

8.4 Probing requirement

To probe 30K prefixes in an 1-hour TE window, the bandwidth usage of each prober will be 30K (prefixes) x 2 (alternative routes) x 2 (RTT measurements) x 5 (TCP packets) x 80 (bytes) / 3600 (seconds) = 0.1 Mbps. Such overhead is negligibly small.

8.5 Processing traffic data

We use a Windows Server 2008 machine with two 2.5 GHz Xeon processors and 16 GB memory to collect and process the Netflow data from all the routers in MSN. It takes about 80 seconds to process the traffic data of one 5-minute interval during peak time. Because Netflow data is processed on-the-fly as the data is streamed to Entact, such processing speed is fast enough for online TE optimization.

9 Related Work

Our work is closely related to the recent work on exploring route diversity in multihoming, which broadly falls into two categories. The first category includes measurement studies that aim to quantify the potential performance benefits of exploring route diversity, including the comparative study of overlay routing *vs.* multihoming [7,8,11,24]. These studies typically ignore the cost of the multihoming connectivity. In [7], Akella *et al.* quantify the potential performance benefits of multihoming using traces collected from a large CDN network. Their results show that smart route selection has the potential to achieve an average performance improvement of 25% or more for a 2-multihomed customer in most cases, and most of the benefits of multihoming can be achieved using 4 providers. Our work differs from these studies in that it considers both performance and cost.

The second category of work on multihoming includes algorithmic studies of route selection to optimize cost, or performance under certain cost constraint [12, 15]. For example, Goldenberg *et al.* [15] design a number of algorithms that assign individual flows to multiple providers to optimize the total cost or the total latency for all the flows under fixed cost constraint. Dhamdhere and Dovrolis [12] develop algorithms for selecting ISPs for multihoming to minimize cost and maximize availability, and for egress route selection that minimizes the total cost under the constraint of no congestion. Our work differs from these algorithmic studies in a few major ways. First, we propose a novel joint TE optimization technique that searches for the optimal "sweet-spot" in the performance-cost continuum. Second, we present the design and implementation details of a route-injection-

based technique that measures the performance of alternative paths in real-time. Finally, to our knowledge, we provide the first TE study on a large OSP network which exhibits significantly different characteristics from multihoming stub networks previously studied.

Our work as well as previous work on route selection in multihoming differ from numerous work on intra- and inter-domain traffic engineering, *e.g.,* [13, 18, 23]. The focus of these later studies is on balancing the utilization of ISP links instead of on optimizing end-to-end user performance.

10 Conclusion

We studied the problem of optimizing cost and performance of carrying traffic for an OSP network. This problem is unique in that an OSP has the flexibility to source traffic from different data centers around the globe and has hundreds of connections to ISPs, many of which carry traffic to only parts of the Internet. We formulated the TE optimization problem in OSP networks, and presented the design of the Entact online TE scheme. Using our prototype implementation, we conducted a trace-driven evaluation of Entact for a large OSP with 11 data centers. We found that that Entact can help this OSP reduce the traffic cost by 40% without compromising performance. We also found these benefits can be realized with acceptably low overheads.

11 Acknowledgments

Murtaza Motiwala and Marek Jedrzejewicz helped with collecting Netflow data. Iona Yuan and Mark Kasten helped with maintaining BGP sessions with the routers in MSN. We thank them all.

We also thank Jennifer Rexford for shepherding this paper and the NSDI 2010 reviewers for their feedback.

References

[1] Linear programming. http://en.wikipedia.org/wiki/Linear_programming. Retrieved March 2010.

[2] Nielsen Online. http://www.nielsen-online.com/.

[3] Quagga Routing Suite. http://www.quagga.net/.

[4] Schooner User-Configurable Lab Environment. http://www.schooner.wail.wisc.edu/.

[5] Share your favorite personal Windows Live Messenger story with the world! http://messengersays.spaces.live.com/Blog/cns!5B410F7FD930829E!73591.entry. Retrieved March 2010.

[6] The MOSEK Optimization Software. http://www.mosek.com/.

[7] AKELLA, A., MAGGS, B., SESHAN, S., SHAIKH, A., AND SITARAMAN, R. A Measurement-Based Analysis of Multihoming. In *Proc. of ACM SIGCOMM* (2003).

[8] AKELLA, A., PANG, J., MAGGS, B., SESHAN, S., AND SHAIKH, A. A Comparison of Overlay Routing and Multihoming Route Control. In *Proc. of ACM SIGCOMM* (2004).

[9] CAO, Z., WANG, Z., AND ZEGURA, E. Performance of hashing-based schemes for Internet load balancing. In *Proc. of IEEE INFOCOM* (2001).

[10] DABEK, F., COX, R., KAASHOEK, F., AND MORRIS, R. Vivaldi: A Decentralized Network Coordinate System. In *Proc. of ACM SIGCOMM* (2004).

[11] DAI, R., STAHL, D. O., AND WHINSTON, A. B. The Economics of Smart Routing and QoS. In *Proc. of the 5th International Workshop on Networked Group Communications (NGC)* (2003).

[12] DHAMDHERE, A., AND DOVROLIS, C. ISP and Egress Path Selection for Multihomed Networks. In *Proc. of IEEE INFOCOM* (2006).

[13] FEAMSTER, N., BORKENHAGEN, J., AND REXFORD, J. Guidelines for interdomain traffic engineering. *ACM SIGCOMM Computer Communications Review* (2003).

[14] FRANCIS, P., JAMIN, S., JIN, C., JIN, Y., RAZ, D., SHAVITT, Y., AND ZHANG, L. IDMaps: A Global Internet Host Distance Estimation Service. *IEEE/ACM Transactions on Networking* (2001).

[15] GOLDENBERG, D., QIU, L., XIE, H., YANG, Y. R., AND ZHANG, Y. Optimizing Cost and Performance for Multihoming. In *Proc. of ACM SIGCOMM* (2004).

[16] GUMMADI, K. P., MADHYASTHA, H., GRIBBLE, S. D., LEVY, H. M., AND WETHERALL, D. J. Improving the Reliability of Internet Paths with One-hop Source Routing. In *Proc. of OSDI* (2004).

[17] HU, N., AND STEENKISTE, P. Evaluation and characterization of available bandwidth probing techniques. *IEEE JSAC Special Issue in Internet and WWW Measurement, Mapping, and Modeling* (2003).

[18] IETF TRAFFIC ENGINEERING WORKING GROUP. http://www.ietf.org/htmk.chaters/tewg-charter.html.

[19] JAIN, M., AND DOVROLIS, C. End-to-end available bandwidth: Measurement methodology, dynamics, and relation with tcp throughput. In *Proc. of ACM SIGCOMM* (2002).

[20] KOHAVI, R., HENNE, R. M., AND SOMMERFIELD, D. Practical guide to controlled experiments on the web: Listen to your customers not to the hippo. In *Proc. of SIGKDD* (2007).

[21] MYTHILI, M. F., VUTUKURU, M. J. F. M., FEAMSTER, N., AND BALAKRISHNAN, H. Geographic locality of ip prefixes. In *Proc. of the Internet Measurement Conference (IMC)* (2005).

[22] NG, T. S. E., AND ZHANG, H. Predicting Internet Network Distance with Coordinates-Based Approaches. In *Proc. of IEEE INFOCOM* (2002).

[23] ROUGHAN, M., THORUP, M., AND ZHANG, Y. Traffic Engineering with Estimated Traffic Matrices. In *Proc. of the Internet Measurement Conference (IMC)* (2003).

[24] SEVCIK, P., AND BARTLETT, J. Improving User Experience with Route Control. Tech. Rep. 5062, NetForecast Inc., 2002.

[25] STRAUSS, J., KATABI, D., KAASHOEK, F., AND PRABHAKAR, B. Spruce: A lightweight end-to-end tool for measuring available bandwidth. In *Proc. of the Internet Measurement Conference (IMC)* (2003).

[26] VAZIRANI, V. V. *Approximation Algorithms*. Springer-Verlag, 2001.

[27] WONG, B., SLIVKINS, A., AND SIRER, E. G. Meridian: a lightweight network location service without virtual coordinates. In *Proc. of ACM SIGCOMM* (2005).

[28] ZEITOUN, A., AND JAMIN, S. Rapid Exploration of Internet Live Address Space Using Optimal Discovery Path. In *Proc. of IEEE Globecom* (2003).

Exploring Link Correlation for Efficient Flooding in Wireless Sensor Networks

Ting Zhu, Ziguo Zhong, Tian He, and Zhi-Li Zhang
University of Minnesota, Twin Cities

Abstract

Existing flooding algorithms have demonstrated their effectiveness in achieving communication efficiency and reliability in wireless sensor networks. However, further performance improvement has been hampered by the assumption of link independence, a design premise imposing the need for costly acknowledgements (ACKs) from every receiver. In this paper, we present Collective Flooding (CF), which exploits the *link correlation* to achieve flooding reliability using the concept of *collective ACKs*. CF requires only 1-hop information from a sender, making the design highly distributed and scalable with low complexity. We evaluate CF extensively in real-world settings, using three different types of testbeds: a single hop network with 20 MICAz nodes, a multi-hop network with 37 nodes, and a linear outdoor network with 48 nodes along a 326-meter-long bridge. System evaluation and extensive simulation show that CF achieves the same reliability as the state-of-the art solutions, while reducing the total number of packet transmission and dissemination delay by 30 $\sim$ 50% and 35 $\sim$ 50%, respectively.

1 Introduction

In wireless sensor networks, flooding is a protocol that delivers a message from one node to all the other nodes. Flooding is a fundamental operation for time synchronization [15], data dissemination [25, 26, 17, 10], group formation [14], node localization [39], and routing tree formation [6, 29].

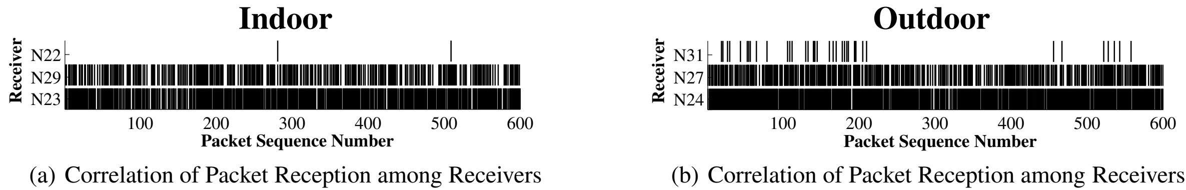
Existing flooding algorithms [18, 34, 24, 12] have demonstrated their effectiveness in achieving communication efficiency and reliability in wireless sensor networks. Further performance improvement, however, has been hampered by the implicit assumption of *link independence* adopted in previous designs. In other words, existing flooding algorithms assume that the reception of a flooding message by multiple neighboring nodes is probabilistically independent of each other. Under such an assumption, it is necessary to have an acknowledgement (ACK) *directly* from the intended receiver for reliable flooding. This is because a node's ACK cannot be used to estimate the reception at other neighboring nodes if link independence is assumed.

However, *direct ACKs* per receiver may lead to high collision [11, 8], congestion [2], and possibly the ACK storm problem [24] in wireless networks. To address this inefficiency in ACKs, this work presents the first comprehensive study to exploit link correlation in the context of flooding design in wireless sensor networks. The driving idea behind our design is *collective ACKs*. Previously, a sender estimated whether a transmission was successful based only on the feedback from the *intended* receiver. Instead, the mechanism of *collective ACKs* allows the sender to infer the success of a transmission to a receiver based on the ACKs from *other* neighboring receivers by utilizing the link correlation among them. Specifically, we use the Conditional Packet Reception Probability (CPRP) as a metric to characterize the correlation among links. The CPRP is the probability of a node's successfully receiving a packet, given the condition that its neighbor has received the same packet. Based on the environment's stability, this metric is measured and calculated online among neighboring nodes using a form of hello message at an adaptive time interval (i.e., small interval when the environment is dynamic and large interval when the environment is stable).

With link correlation information (CPRP) available among neighboring nodes, *collective ACKs* are achieved in an accumulative manner. The success of a transmission to a node (defined as the *coverage probability* of a node) is no longer a binary (0/1) estimation, but a probability value between 0 and 1. Using *collective ACKs*, a sender updates the coverage probability values of neighboring receivers whenever (i) it transmits or (ii) overhears a rebroadcast message. To improve efficiency, a transmission is considered necessary only when the coverage probability of a neighboring node has not reached a certain user-desired reliability threshold.

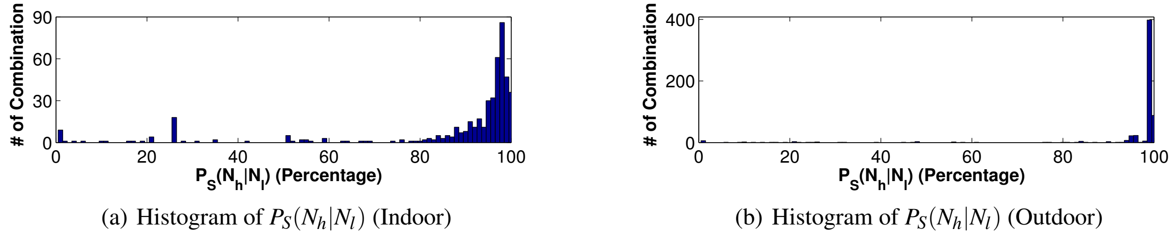
In addition to *collective ACKs*, we propose a dynamic forwarding technique to exploit link correlation further. In Collective Flooding, only a small set of nodes is selected dynamically as the forwarders of a flooding message via self-organized competition among neighboring nodes. Every node estimates its transmission effectiveness based on three factors: (i) neighborhood size, (ii) link quality, and (iii) link correlation among neighboring nodes. The most effective node will start to rebroadcast early to suppress the less effective nodes' rebroadcasts, consequently reducing the message redundancy.

In summary, our contributions are as follows:



(a) Correlation of Packet Reception among Receivers (b) Correlation of Packet Reception among Receivers

Figure 1. Correlation of Packet Reception



(a) Histogram of $P_S(N_h|N_l)$ (Indoor) (b) Histogram of $P_S(N_h|N_l)$ (Outdoor)

Figure 2. Distribution of Conditional Packet Reception Probability

• To our knowledge, *collective ACK* is a new concept that can improve the efficiency of reliable flooding operations. It transforms the traditional *direct* ACKs per receiver into *correlated* and *accumulative* ACKs.

• Although the phenomena of link correlation has been mentioned in the literature [31], we provide the first extensive study to exploit this phenomena for communication improvement. We reveal that link correlation can be used to achieve (i) *collective ACKs*, as well as (ii) efficient forwarder selection.

• Our design is simple and symmetric. Rebroadcast decisions at individual nodes are based on the coverage probability values of neighbors, which in turn are updated by overhearing rebroadcasts from their neighbors. All the operations only need 1-hop neighbors' information, making our protocol highly distributed and scalable.

• We evaluate our work extensively in multiple real-world testbeds and large scale simulation. The results indicate that our design is practical, reliable, and outperforms several existing state-of-the-art designs.

The rest of the paper is organized as follows: Section 2 presents the motivation behind the work. Section 3 introduces two key mechanisms. Section 4 describes the design. Sections 5 and 6 evaluate the work with testbeds and simulation. After discussing related work in Section 7, Section 8 concludes the paper.

2 Motivation

Previous studies on wireless links focus on packet receptions of individual receivers with single [31, 33, 4, 43, 22] or multiple [21] radios. Little systematic study has investigated the packet reception correlation among neighboring receivers. To fill the gap, this section reports our empirical study on wireless link correlation. More specifically, we observed the following phenomena, which serves as the foundation of this work.

Observation: *For packets transmitted from the same sender, if a packet is received by a node with a low packet reception ratio (PRR), it is highly probable that this packet is also received by the nodes with a high PRR.*

2.1 Experiment Setup

In our experiments, 42 MICAz nodes were used. The experiments were conducted with multiple randomly generated layouts under two different scenarios: (i) an open parking lot, and (ii) an indoor office. In each scenario, two types of experiments were conducted: Fixed Single Sender and Round Robin Sender. In the Fixed Single Sender experiment, the sender was placed in the center of the topology, while the other 41 nodes were randomly deployed as receivers. The sender broadcasted a packet in every 200*ms*. Each packet was identified by a sequence number. The total number of packets broadcasted was 6000. In the Round Robin Sender experiment, each node in turn broadcasted 200 packets with time intervals of 200*ms*. The receivers kept track of the received packets through the sender's ID and packet sequence number.

2.2 Correlated Packet Reception

In both indoor and outdoor experiments, we discovered that if a packet is received by a sensor node with low PRR, most of the time this packet is also received by the high PRR nodes. Figure 1(a) and 1(b) illustrate the first 600 packet receptions of two groups of three nodes in indoor and outdoor experiments, respectively. The locations of the nodes are shown in Figure 3 and Figure 4. The black bands correspond to the packets received at the nodes. Clearly, there exists a strong correlation of packet receptions among the neighboring nodes. For example, in Figure 1(a), given the two packets (sequence number 282 and 508) received by $N22$, these two packets were also received by $N29$ and $N23$. In order to quantify this correlation, we define the Conditional Packet Reception Probability (CPRP), as follows:

Definition: The *Conditional Packet Reception Probability is the probability that a high PRR node N_h receives a packet M from sender node S, given the condition that the packet M is received by a low PRR node N_l.*

We use $P_S(N_h|N_l)$ to denote CPRP, where N_h and N_l

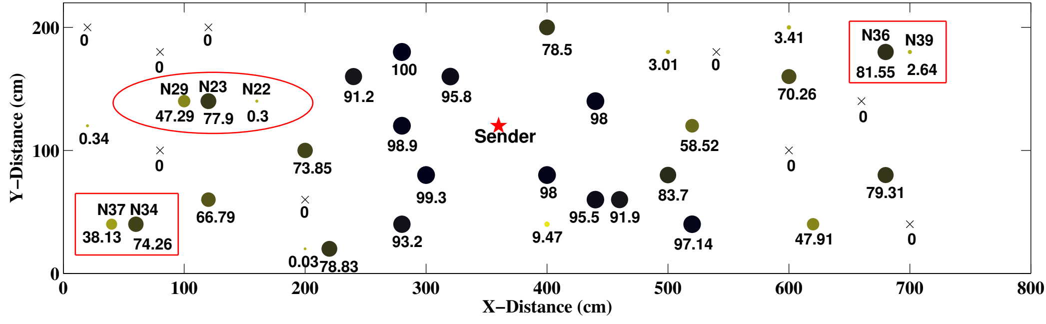


Figure 3. Packet Reception Ratios (PRR) of Individual Nodes in an Indoor Experiment

are neighboring receivers of the sender S. For example, in Figure 1(b), node $N31$ received 38 packets and 37 out of these packets were also received by node $N27$, so $P_S(N27|N31) = 97.4\%$. If the assumption on link independence holds, we would expect $P_S(N27|N31) = P_S(N27)$. However, this is not the case; as shown by the experiment, $P_S(N27)$ is 64.9% instead of 97.4%. This indicates a packet reception correlation between $N31$ and 27, which is also valid for node pairs $N31 \Leftrightarrow N24$ and $N27 \Leftrightarrow N24$.

To analyze the CPRP among the pairwise receivers more systematically, we computed the CPRP for all node pairs with non-zero PRR values. In the indoor experiment, we had 32 non-zero PRR nodes, which generates $\frac{32\times31}{2} = 496$ combinations of $P_S(N_h|N_l)$. Figure 2(a) shows the distribution of these combinations. Figure 2(b) illustrates the distribution for the outdoor experiment.

Figure 2(a) and 2(b) show that the conditional packet reception probability $P_S(N_h|N_l)$ is collectively distributed close to 100%. This result verified our observation that if a packet is received by a low PRR node, this packet has a high probability of also being received by a high PRR node. Due to physical constraints, we only evaluated the link correlation by using MICAz platform. The background traffic or interference would also cause the link correlation on the other radio platforms [32].

2.3 Spatial Diversity in PRR

Besides the link correlation, we also confirmed that the packet reception ratios (PRR) of the receivers had a diverse spatial distribution [4]. Figure 3 shows the spatial distribution of PRR in the indoor Fixed Single Sender experiment. The centers of "•" and "x" indicate the nodes' locations. The larger the size of a "•", the higher the PRR value of this node, while "x" represents nodes that do not receive any packet. The numbers underneath the nodes' locations are Packet Reception Ratio (PRR) values.

Our Fixed Single Sender experiments show that even when two receivers are physically very close to each other, these receivers may have totally different PRRs. For example, in Figure 3, although $N36$ and $N39$ are located near each other at the upper-right corner, their PRRs are significantly different, 81.55% and 2.64%, respectively. There are many node pairs with such features, such as $N34$ and $N37$, $N23$ and $N22$. A similar phenomenon also occurs in the outdoor experiment, such as $N19$ and $N10$, shown in Figure 4.

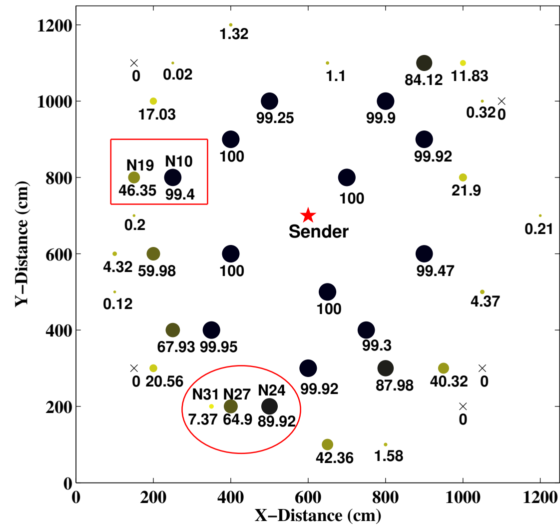


Figure 4. PRR(%) for Each Node (Outdoor)

2.4 Opportunity and Challenges on Flooding

While these observations would impact many protocol designs, this paper focuses on the flooding protocol design in particular.

Link Correlation: Existing flooding protocols did not take advantage of this correlated reception feature. As a result, direct ACKs from receivers is normally used when high reliability is desired. In other words, every receiver needs to send ACKs in response to the reception of a packet, leading to high communication overhead (when explicit ACKs are used) or high redundancy in rebroadcasting (when implicit ACKs are used). The research challenge here is how to exploit link correlation, so that the overhead of ACKs is reduced.

PRR Spatial Diversity: Section 2.3 shows that within the radio range of a sender, even when receivers are located close to each other, they may have dramatically different PRRs due to environmental effects such as multi–

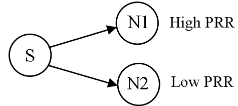


Figure 5. Collective ACKs

path fading. If a flooding protocol selects a fixed forwarder, this forwarder has to retransmit a large number of times to accommodate the receivers with low PRRs, introducing excessive duplicated reception for those receivers with high PRRs. The challenge here is how to reduce the impact of spatial diversity, so that the overhead of redundant transmissions is reduced.

In the rest of the paper, we present two corresponding mechanisms to deal with these two challenges respectively. We explain the ideas conceptually first in Section 3, followed by detailed design in Section 4.

3 Key Mechanisms in Collective Flooding

The main objective of collective flooding (CF) is to reduce redundant transmissions inside the network while providing reliable message dissemination. In CF, we call a node a *covered* node if it has already received the broadcasting packet. Covered nodes are responsible for rebroadcasting the packet to uncovered nodes in the network. In our design, rebroadcasting is used as an implicit ACK to the sender to save protocol overhead. We note that CF can be also applied when explicit ACKs are used. Specifically, there are two key mechanisms in the CF protocol:

- **Collective ACKs:** In CF, the overhearing of a node's rebroadcasting not only indicates that this node has received the packet, but also serves as a *collective ACK* of reception for some other neighboring nodes.
- **Dynamic Forwarder Selection:** The forwarder is selected dynamically through competition among nodes that have already received the broadcasting packet.

3.1 Benefit of Collective ACKs

The mechanism of *collective ACKs* allows a node to extract information about the status of its neighboring nodes via receiving or overhearing a packet from its neighbors. For example, in Figure 5, suppose that node S is a covered node while $N1$ and $N2$ are uncovered. They are within 1-hop communication range of each other, where $N1$ is a low PRR receiver of S and $N2$ is a high PRR receiver of S. When S broadcasts, if $N1$ receives the packet, in traditional flooding protocols without considering the correlation, $N1$ only knows that S is covered, but still considers $N2$ as uncovered until $N1$ overhears $N2$'s rebroadcasting.

CF takes a different approach. From $N1$'s viewpoint, a packet from S serves two purposes. First, it is a *direct ACK* that S is a covered node. Second, it also serves as a *collective ACK* to $N1$ that $N2$ has a reception probability of $P_S(N2|N1)$. Similarly, from S's viewpoint, if S later overhears the rebroadcasting (i.e., an implicit ACK)

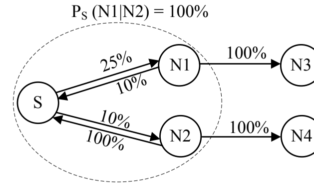


Figure 6. Example of Collective ACKs

from $N1$, S not only gets a direct ACK that $N1$ is covered, but is also able to compute the coverage probability of $N2$ according to the link correlation metric $P_{N1}(N2|S)$. We note that in traditional designs, overhearing a packet serves only as a *direct ACK* that the packet sender is covered. In CF, the ACK is achieved in a *collective* manner, i.e., overhearing a packet serves as both direct and correlated ACKs from the packet sender and its neighbors.

Collective ACKs can greatly reduce the redundant transmission. For the sake of clarity, let us consider the simplified network shown in Figure 6. The link qualities from node S to $N1$ and $N2$ are 25% and 10%, respectively; the link qualities from $N1$ and $N2$ to S are 10% and 100%, respectively. We assume the CPRP $P_S(N1|N2) = 100\%$, which means that if $N2$ receives a packet from S, $N1$ also receives that packet.

In traditional flooding protocols, the sender S treats the receivers' packet receptions as independent. To provide reliable broadcasting, S needs to keep on transmitting until it receives ACKs or overhears the transmissions from both $N1$ and $N2$. Due to the low link quality from $N1$ back to S (10%), S might conduct many redundant retransmissions. In contrast, *collective ACKs* in CF allow node S to terminate the transmission earlier if $N2$ receives the flooding packet with a smaller number of retransmissions than expected. For example, if $N2$ receives the packet at the first attempt (*luckily*) and rebroadcasts, node S can immediately terminate the retransmission to $N1$, based on the assumption $P_S(N1|N2) = 100\%$. Therefore, in this case, the number of transmissions at node S can be reduced to one. As we can see from the above simplified example, *collective ACKs* can improve the efficiency of the reliable flooding protocol by utilizing the link correlation.

3.2 Benefit of Dynamic Forwarder Selection

As discussed in Section 2.4, a fixed-forwarder scheme has to accommodate the receiver with the lowest reception ratio, leading to high redundancy for the nodes with high reception ratios. To address this problem, in CF the covered nodes compete for becoming the forwarder node based on their *transmission effectiveness*, which is defined in Section 4.4 and calculated according to three factors: (i) neighborhood size, (ii) link quality, and (iii) coverage probability based on link correlation metric CPRP. A node's transmission is considered more effective if the node has more uncovered neighbors and good link qualities to them. This node wins the competition and rebroadcasts with the shortest back-off time. The nodes

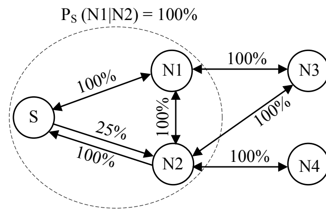


Figure 7. Example of Dynamic Forwarder Selection

transmission would change the transmission effectiveness value of this node and its neighboring nodes. Based on the transmission effectiveness, the forwarder is dynamically selected. This design avoids the same node being selected as the forwarder all the time.

To illustrate the benefit further, we give an example to demonstrate the process of the dynamic forwarder selection. Again, let us consider a simplified scenario as in Figure 7. The link quality from source node (S) to $N2$ is 25%. All the other link qualities are 100%. $N3$ and $N4$ are 2 hops away from S. In order to minimize the total number of transmissions, traditional approaches, such as [18, 12], intend to select a fixed-forwarder among neighboring nodes according to their uncovered neighbor size, in other words, capability of covering more uncovered nodes. For example, in Figure 7, S selects $N2$ as a dedicated forwarder to rebroadcast the packet. The reason is that $N2$ has more uncovered neighbors ($N3$ and $N4$) than $N1$, which only has one uncovered neighbor ($N3$). However, due to the unreliable link between S and $N2$, a packet needs to be transmitted 4 times on average from S before it is received by $N2$. Then, another transmission is needed by $N2$ to cover $N3$ and $N4$. In total, an average of 5 transmissions are needed for a single network-wide broadcast.

In contrast, CF adopts a dynamic and opportunistic approach. After S broadcasts, S, $N1$, and $N2$ compete to be a forwarding node instead of using a dedicated forwarder. Based on the actual reception status, there are two cases:

Case 1: If $N2$ receives the packet (*luckily*) at first attempt, $N2$ can tell that $N1$ is covered based on CPRP $P_S(N1|N2) = 100\%$. After marking $N1$ as a covered node, $N2$ still has more uncovered neighbors ($N3$ and $N4$) than S and $N1$, and thus $N2$ wins the forwarder selection competition and rebroadcasts. After $N2$'s rebroadcasting, all the nodes update their neighbors' coverage probabilities and find that all their neighbors are covered. Therefore, the competition stops and no more transmissions are needed. The total number of transmissions for the network is 2.

Case 2: If $N2$ does not receive the packet, a competition occurs between S and $N1$. $N1$ is supposed to win because it has both $N2$ and $N3$ as uncovered nodes, while S has only $N2$ as an uncovered node. After $N1$'s broadcasting, $N2$ will receive the packet and win the competition because it is the only node that has uncovered neighbor $N4$. One more transmission is needed from $N2$ to cover $N4$. Therefore, the total number of transmissions for the network is 3.

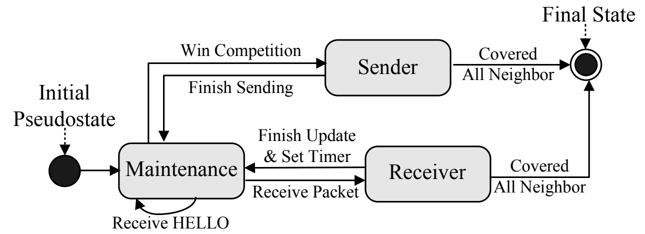


Figure 8. State Machine Diagram of CF

By introducing competition among the covered nodes, CF reduces the redundant transmissions. In the above example, even in the worst case (Case 2), CF only needs 3 transmissions, which is smaller than the traditional fixed-forwarder approaches' 5 times.

4 The Collective Flooding Protocol

This section presents the main design of the CF, which is a simple finite state machine. As shown in Figure 8, a node running CF is in one of three states at any time: (i) maintenance, (ii) receiver, and (iii) sender. Transitions between the states are triggered by events.

After the CF protocol is initiated, the node enters the maintenance state, in which all of its 1-hop neighbor information is periodically maintained. Here, two nodes are considered as neighbors if the link quality between them is larger than 0%. Whenever the node receives a broadcasting data packet, the node enters the receiver state and uses this packet as a *collective ACK* to update its neighbors' coverage probabilities. If the node has uncovered neighbors, it sets its back-off timer based on its transmission effectiveness, then goes back to the maintenance state. When the node's back-off timer fires, which means it wins the competition, it enters the sender state, in which it sends out the packet and updates its neighbors' coverage status; after that, it goes back to the maintenance state. This procedure repeats until the node estimates that all its neighbors are covered. In the rest of this section, we explain the operations in each state in detail.

4.1 Maintenance State

Wireless links in sensor networks are known to be dynamic. Therefore, maintenance is needed to keep track of the link quality. In CF, every node periodically sends out a hello message at an adaptive time interval T which is increased or decreased based on the link's stability. Every hello message is identified by the node ID and a packet sequence number. The hello message is used not only for 1-hop neighbor discovery, but also for updating the link qualities and calculating the Conditional Packet Reception Probability (CPRP) among neighboring nodes. While link quality calculation is straightforward, the calculation of CPRP deserves a little more explanation. Every node maintains a reception record of all hello messages from its neighboring nodes within a time window wT. In order to reduce the required mem-

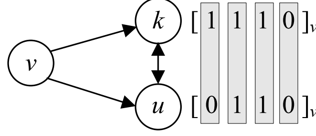


Figure 9. Example of Calculating CPRP

ory space and mitigate the overhead of control messages, the record is represented in a bitmap format (e.g., [0110]) for each neighbor. Such records are exchanged within a hello message every wT seconds among neighboring nodes. CPRP is calculated as follows:

$$P_v(k|u) = \frac{\sum_{i=1}^{w} B_{vk}(i) \ \& \ B_{vu}(i)}{\sum_{i=1}^{w} B_{vu}(i)} \quad (1)$$

Here v is the sender; k and u are the two receivers. $B_{vk}(i)$ is a bit representing node k's reception status of the ith hello message sent from node v. $B_{vk}(i) = 1$ if k receives this message, otherwise $B_{vk}(i) = 0$. For example, in Figure 9, a bitmap of $[1110]_v$ from node k indicates that k does not receive node v's 4th transmission. When node u receives this bitmap, it can use Equation 1 to calculate CPRP by performing the bit-wise *AND* operation with its own bitmap ($[0110]_v$). For example, The CPRP is calculated as $P_v(k|u) = \frac{1\&0+1\&1+1\&1+0\&0}{0+1+1+0} = 100\%$. We note that the length of the bitmap w strikes a balance between the control overhead and statistical confidence of the CPRP value.

4.2 Receiver State

A node enters the receiver state once it receives or overhears a broadcasting packet. Nodes in the receiver state compete to be selected as a forwarder. Without losing generality, suppose node u is the receiver of sender v. Node u maintains two pieces of information:

• **Coverage Probability:** This is the probability of a neighboring node's being covered in a broadcast from the viewpoint of a node. For example, $CP_u(k)$ is node u's estimated probability that u's neighbor k has received the broadcasting packet. Node u maintains $CP_u(k)$ for all its 1-hop neighboring nodes $k \in N(u)$, where $N(u)$ is u's neighboring node set.

• **Estimated Uncovered Node Set $U(u)$:** Here $U(u) \subseteq N(u)$. Initially, node u considers all of its 1-hop neighbors as uncovered. So for any node $k \in N(u)$, $CP_u(k) = 0$. u's uncovered node set is $U(u) = N(u)$.

Supposing node u receives a broadcasting packet M from its neighboring sender v, this packet serves two purposes. First, since the packet M is received from node v, u updates the coverage probability of v as $CP_u(v) = 1$, meaning that u is sure that v has already received the packet (note that this is actually a direct implicit ACK). Second, the packet also serves as a *collective ACK* for all other neighbors $k \in N(u)$. Based on the conditional packet reception probability $P_v(k|u)$ stored in its neighbor table, the coverage probability of other nodes $k \in \{N(u) - v\}$ is updated as follows:

$$CP_u(k) \leftarrow 1 - (1 - CP_u(k)) \cdot (1 - P_v(k|u)) \quad (2)$$

where the term $(1 - CP_u(k))$ is the probability that k had not received the packet M before v's forwarding; the term $(1 - P_v(k|u))$ is the probability of k's failure to receive M from v given the condition that u received M. So $1 - (1 - CP_u(k)) \cdot (1 - P_v(k|u))$ is the probability of node k's being covered either by (i) previous transmission in the network or (ii) current forwarding from v. We note that $CP_u(k)$ is the coverage probability estimated by node u. Formula 2 utilizes $P_v(k|u)$ to accumulate node u's confidence in treating k as a covered node. Namely, u's receiving from v also serves as a *collective ACK* for k. In the worst case when there is no link correlation, the conditional packet reception probability of a node will be equal to the link quality (i.e., $P_v(k|u) = P_v(k)$). In this case, our flooding protocol uses the link quality information (i.e., $P_v(k)$) to update the coverage probability via Formula 2.

When coverage probability $CP_u(k)$ reaches a user's pre-specified threshold $\alpha \leq 1$, node k is considered by node u as covered and is removed from node u's uncovered node set $U(u)$. If $U(u)$ is not empty, node u joins the competition for being the next local forwarder by setting its *back-off timer* according to its transmission effectiveness, which is detailed later in Section 4.4. If $U(u)$ is empty, node u exits the receiver state and completes its broadcasting mission, as shown in Figure 8.

We note that before the timer expires, if node u overhears the broadcast packet M again from one of its neighbors, u cancels the current running timer and repeats the above coverage probability updating procedure and resets its timer. If u's timer fires before all other competitors', u is selected as the forwarder. It enters the sender state to send out broadcast packet M and perform related updates, as explained in the next subsection.

4.3 Sender State

By winning the forwarder competition, node u enters the sender state and sends out the packet. Then, it updates the coverage probabilities of its uncovered neighbors $k \in U(u)$ with

$$CP_u(k) \leftarrow 1 - (1 - CP_u(k)) \cdot (1 - L(u,k)) \quad (3)$$

Here $L(u,k)$ is the link quality between u and k, so the term $(1 - L(u,k))$ indicates the probability of k's failure to receive the broadcasting packet from u. From u's point of view, k has a probability of $(1 - CP_u(k))$ of being uncovered before u's forwarding, and therefore the term $1 - (1 - CP_u(k)) \cdot (1 - L(u,k))$ shows the probability of the event that node k either was covered previously or is covered by u's current forwarding.

As in the receiver state, when the coverage probability $CP_u(k)$ of node k reaches a user-specified threshold α, node k is considered covered and is removed from the uncovered node set $U(u)$. If $U(u)$ is empty, node u terminates the flooding task; otherwise, node u joins the forwarder competition again by setting its back-off timer and returning to the maintenance state. We note that the

value of α is used as a threshold to terminate the retransmission at each node. Different α values can achieve different reliabilities for different applications.

4.4 Back-off Timer Design

The back-off timer is used to conduct dynamic forwarder selection. In the forwarder competition, the duration of the back-off timer is carefully set according to a combination of factors, including (i) neighborhood size, (ii) link quality, and (iii) neighbors' CPRPs. Intuitively, if a node has more uncovered neighbors with good link quality, this node should be the next forwarder and thus should have a short duration before the timer fires.

In CF, we define *Transmission Effectiveness (TE)* as a reference metric for setting the back-off time period.

Definition: *$TE(u)$ equals the number of uncovered nodes that are expected to be covered if the sender u transmits once.*

In general, the value of TE for node u can be calculated with the following Equation

$$TE(u) = \sum_{k \in U(u)} L(u,k) \cdot (1 - CP_u(k)) \quad (4)$$

The meaning of Equation 4 is straightforward. The transmission effectiveness of u equals the summation of the probabilities of covering u's uncovered neighbors, namely the expected number of nodes to be covered by u's forwarding. The higher the $TE(u)$ value, the more effective node u's transmission is. For example, in a perfect link ($L(u,k) = 100\%$) scenario, if node u has 2 uncovered nodes, the TE value of u is $TE(u) = 1 \times 1 + 1 \times 1 = 2$, meaning that if u transmits once, 2 neighbors get covered. In another example, if the link qualities from u to these uncovered nodes are all 50%, the TE value of u is $TE(u) = 0.5 \times 1 + 0.5 \times 1 = 1$, meaning that if u transmits once, one neighbor is expected to get covered. Since node u updates the value of $CP_u(k)$ whenever it sends or receives a broadcast packet from its neighbors, $TE(u)$ changes dynamically during the dissemination of the packet.

Every node continuously updates its TE value and sets the back-off timer based on the TE value. Intuitively, the higher the TE value, the smaller the back-off time period should be. The rationale behind this is that we always select the forwarder which is able to cover more nodes in the network with one transmission.

4.5 The Detailed Protocol

Combining all the design components, CF can be specified by the pseudo code shown in Protocol 1.

The design is simple and requires only 1-hop information. Each node maintains a state machine with three states. A state transition is triggered by the *event* of either receiving a broadcast packet (Line 4) or sending timer fired (Line 12). Lines 4 to 11 handle the event of receiving a packet. Lines 12 to 18 handle the timer fired event by sending out the packet and updating the coverage probability values of neighboring nodes. Lines 22 to 25 update the uncovered set of a node, and lines 26 to 29 determine whether the flooding task has been finished.

Protocol 1: **Collective Flooding**

```
1  Initially, U(u)←N(u), ∀ k ∈ U(u), CP_u(k)←0;
2  repeat
3  |  switch Event do
4  |  |  case u receives packet from v
5  |  |  |  for k ∈ U(u) do
6  |  |  |  |  if k = v then CP_u(k)←1;
7  |  |  |  |  else Updates CP_u(k) via Formula 2;
8  |  |  |  |  end
9  |  |  |  |  Call Update U(u);
10 |  |  |  end
11 |  |  |  Call Test U(u);
12 |  |  case timer fired
13 |  |  |  u sends out the packet;
14 |  |  |  for k ∈ U(u) do
15 |  |  |  |  Update CP_u(k) via Formula 3;
16 |  |  |  |  Call Update U(u);
17 |  |  |  end
18 |  |  |  Call Test U(u);
19 |  |  end
20 |  end
21 until U(u) = φ ;
22 Update U(u) method :
23 if CP_u(k) ≥ α then
24 |  U(u) ← U(u) − {k};
25 end
26 Test U(u) method :
27 if U(u) ≠ φ then Set back-off timer;
28 else Terminate the timer;
29 end
```

In summary, the CF protocol has three efficient features: (i) it can be implemented with a simple finite state machine with 3 states, which is suitable for resource constrained sensor nodes; (ii) it deals with the spatial diversity of packet reception with dynamic forwarder selection; and (iii) it reduces the communication redundancy through *collective ACKs*, eliminating costly direct ACKs from every receiver.

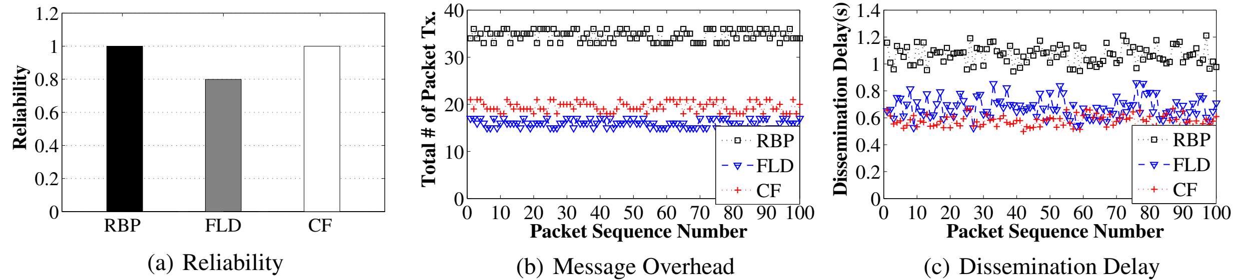
5 Implementation and Evaluation

We have implemented a complete version of CF on the TinyOS [27]/MICAz platform in nesC [5]. The following two protocols are also implemented as a baseline:

- **Standard Flooding (FLD):** Every node rebroadcasts the first-time received packet exactly once.
- **Reliable Broadcast Propagation (RBP):** RBP [34] was proposed in SenSys'06. As in standard flooding, in RBP, every node unconditionally rebroadcasts the first-time received packet once. Then the node adjusts the number of retries based on the neighborhood density. If there exists a bottleneck link from current node (N) to its downstream node (D), node N performs up to the maximum number of retries when it does not receive the ACK from D.

Four metrics are used to evaluate the protocols:

- **Reliability:** Reliability is quantified by the percentage of nodes in a network that receive the flooding packets.
- **Message Overhead:** Message overhead is measured



(a) Reliability (b) Message Overhead (c) Dissemination Delay

Figure 10. The Performance of Single Hop Indoor Experiment

by the total number of data packets transmitted during the experiment period. We do not count hello messages into the overhead for two reasons. First, the overhead of hello messages is highly environment–dependent. It can be very high in extremely dynamic environments, and very low in static environments. For example, in our in–door experiment, the average hourly hello messages are 73 packets per node during the day and 17 packets per node during the night. Second, most flooding designs need hello messages for neighbor information mainte–nance. We acknowledge that CF has extra overhead to exchange bitmap within a hello message every wT time interval to calculate CPRP. However, this overhead is in–dependent of data traffic, hence the cost is amortized over multiple flooding operations.

• **Dissemination Delay:** Dissemination delay is the du–ration from the time that either the source initiates the packet to the time the last node receives the packet or no more nodes resend the packet for a single flood.

• **Load Balance:** This is indicated by the standard devi–ation of the number of packets transmitted per node per flood. This metric measures how evenly the rebroadcast–ing activities are distributed in the network.

5.1 Experiment Setup

During the experiments, we placed MICAz nodes in indoor and outdoor environments and tuned the trans–mission power to ensure the multi–hop communication between the source node and the other nodes. In the experiments, after deployment all the nodes were syn–chronized and started the neighbor discovery by sending out the hello messages. After all the nodes got the link quality and conditional packet reception ratio informa–tion about their 1–hop neighbors, a sender was selected to send out 100 data packets with a time interval of 10 seconds. For performance analysis purposes, in each data packet we included information such as hop count, time stamp, and the previous hop's node ID. Upon re–ceiving the data packet, the intermediate node recorded this information in its flash memory. Every node also recorded the number of transmissions it conducted for each data packet, which was identified by the sequence number. For all the protocols, we maintained the same network placement during the experiments. Unless ex–

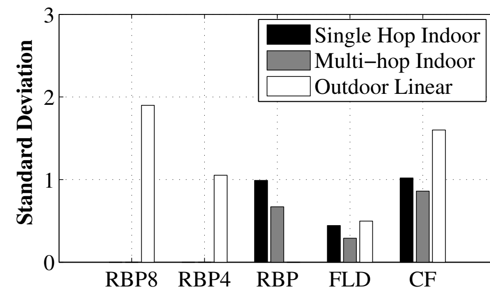


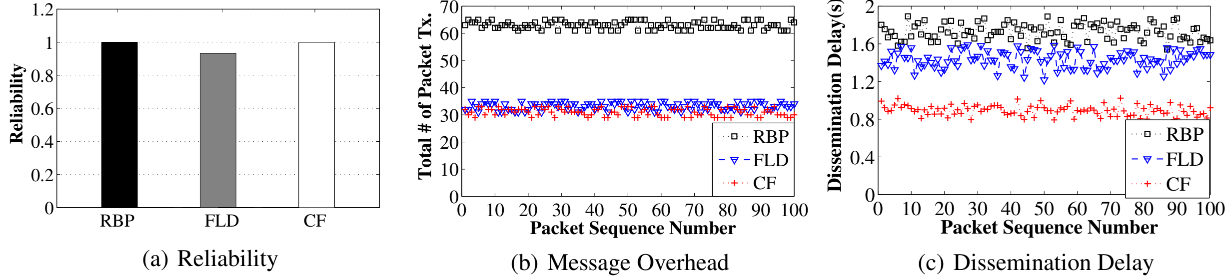
Figure 11. Standard Deviation

plicitly stated otherwise, we used the above default val–ues in all the experiments.

5.2 Single Hop Indoor Experiments

The indoor scenario represents potential applications including facility management [38], data center sensing [16], and structural monitoring [23]. In this experiment, one MICAz node was placed as a sender in the center of the indoor 7.5m $\times$ 2.5m testbed, and the other 19 MICAz nodes were randomly deployed around the sender. The transmission power was tuned to ensure that all of these 19 nodes were within the sender's transmission range, although not necessarily within each other's transmission range. The link qualities (i.e., packet reception ratios of the receivers) from the sender to these 19 nodes varied between 100% and 7%.

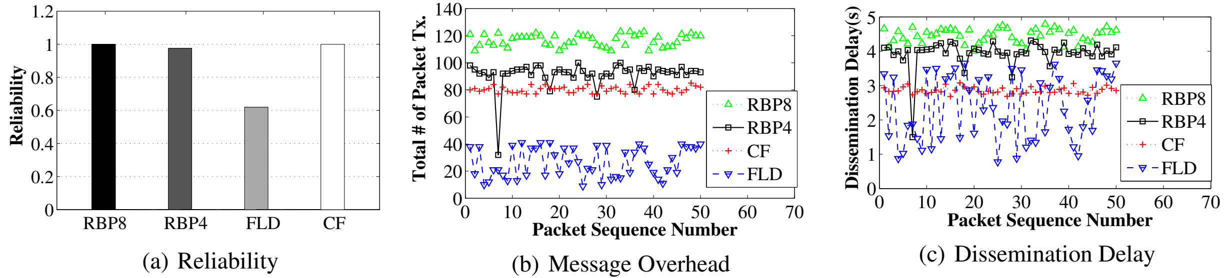
As shown in Figure 10(a), due to the unreliable wire–less links, standard flooding (FLD) has only 79.9% reli–ability. CF, however, achieves the same 100% reliability as RBP. This is because in CF, based on the link qual–ity and strong conditional packet reception ratio infor–mation, every node can accurately estimate whether all its 1–hop neighbors receive the packet. This accurate es–timation also results in a lower number of transmissions inside the network. Figure 10(b) compares the total num–ber of packets transmitted for every data packet initiated from the sender when running RBP, standard flooding (FLD), and CF protocols. The average values of the to–tal number of packets transmitted for RBP, FLD, and CF are 34.64, 15.98, and 19.5, respectively. Compared with RBP, CF reduces 43.7% of the total number of packets transmitted, while maintaining the same 100% reliabil–



(a) Reliability (b) Message Overhead (c) Dissemination Delay

Figure 12. The Performance of Multi-hop Indoor Experiment

Figure 13. Outdoor Experiment Site: On a Bridge



(a) Reliability (b) Message Overhead (c) Dissemination Delay

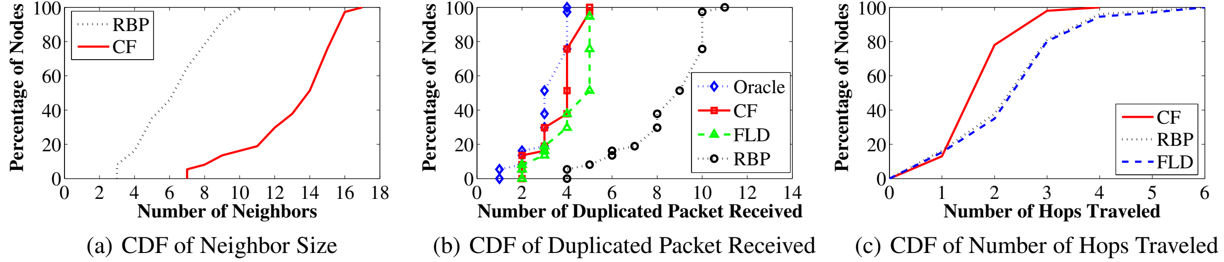
Figure 14. The Performance of Outdoor Linear Network Experiment

ity. Due to the low reliability of FLD, 20.1% of nodes do not receive the packet, and thus FLD has the smallest total number of transmissions.

Figure 10(c) compares the dissemination delays of the three protocols for all 100 data packets initiated by the sender. The average delay for CF is 0.571s, for RBP is 1.09s, and for FLD is 0.69s. Since in CF the node with the largest transmission effectiveness has the smallest back-off time, CF's dissemination delay is 47.6% less than RBP's. As shown in Figure 11, the standard deviations of CF and RBP are very close, but the average numbers of transmissions per node are different (1.73 for RBP, 0.975 for CF). The reason is that in RBP every node unconditionally rebroadcasts the first-time received packet once. In this way, RBP has a larger number of transmissions and a smaller standard deviation than CF.

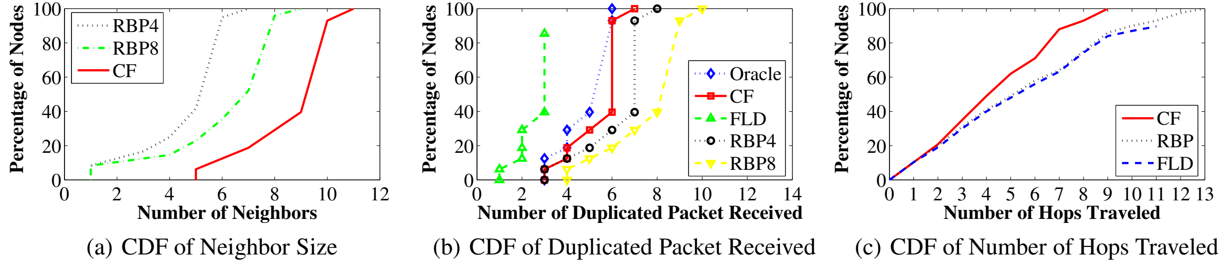
5.3 Multi-hop Indoor Experiments

To further investigate the performance of CF in a larger scale and denser network, a multi-hop indoor experiment was conducted. In this experiment, the MICAz node as a sender was placed in the bottom left boundary of the indoor 7.5m $\times$ 2.5m testbed and the other 36 MICAz nodes were randomly deployed on the testbed. Figure 12(a) shows that the reliability of CF, standard flooding (FLD), and RBP is 99.97%, 93.25%, and 99.89%, respectively. CF achieves the same reliability as RBP, but CF reduces the number of transmissions by 50.7%, as shown in Figure 12(b). The reduction is larger than the 43.7% reduction of number of transmissions in the sparser network (discussed in Section 5.2). This is because in the denser network, the node running CF has more 1-hop neighbors, which would further help the node accurately predict whether its 1-hop neighbors have received the packet. CF also has fewer transmissions than does standard flooding. For instance, the average values of the total number of packets transmitted by using RBP, standard flooding, and CF are 62.94, 33.21, and 31.04, respectively. CF has less number of packets transmitted, which translates to the less amount of energy consumption. In addition, CF does not prevent MAC layer energy management such as low power listening (LPL) [28] and SCP-MAC [41]. For example, in LPL, nodes briefly wake up to check channel activity without actually receiving data. If the channel is not idle, the node stays awake to receive data. Otherwise it immediately goes back to sleep. In this way, LPL protocols consume much less energy than they would listening for full contention period.



(a) CDF of Neighbor Size (b) CDF of Duplicated Packet Received (c) CDF of Number of Hops Traveled

Figure 15. Insight Analysis of Multi-hop Indoor Experiments



(a) CDF of Neighbor Size (b) CDF of Duplicated Packet Received (c) CDF of Number of Hops Traveled

Figure 16. Insight Analysis of Outdoor Experiments

Figure 12(c) compares the dissemination delay of these three protocols. The average delay for RBP, standard flooding, and CF is 1.73s, 1.41s, and 0.89s, respectively. By relying on the node with the largest transmission effectiveness to do the transmission first, the average delay of CF is 48.5% less than that of RBP. The standard deviation of CF, standard flooding, and RBP is 0.866, 0.288, and 0.669, respectively, as shown in Figure 11. CF has a slightly higher standard deviation than RBP. Compared with the sparser network (i.e., single hop indoor), the standard deviation of CF decreases in a denser network (i.e., multi-hop indoor). This is because in a denser network, when running CF, the nodes can more dynamically choose the path to propagate the packets.

5.4 Outdoor Linear Network Experiments

The outdoor experiment represents such potential applications as monitoring remote infrastructures or environments [30, 37, 42]. In the experiment, 48 MICAz nodes were deployed along a 326-meter-long bridge, as shown in Figure 13. For a fair comparison, we implemented two versions of RBP: RBP4 and RBP8. In RBP4, we set the bidirectional link quality threshold for two nodes to be considered neighboring nodes at 50%, and the maximum number of retries when an insufficient number of neighbors got the packet or an important neighbor did not get it at 4. To improve the reliability, in RBP8, the bidirectional link quality threshold was reduced to 30% and the number of retries was set at 8. $N1$ was selected to be the sender that initiated the broadcasting of the data packets.

As shown in Figure 14(a), the reliability of RBP8, RBP4, CF, and standard flooding is 99.96%, 97.6%, 99.93%, and 61.96%, respectively. With the link quality and conditional packet reception ratio information, CF achieves the same reliability as RBP8 and reduces the number of packets transmitted by 31.2% (shown in Figure 14(b)). The average values of the number of packets transmitted for RBP8, RBP4, CF, and standard flooding are 116.64, 91.54, 80.26, and 26.58, respectively.

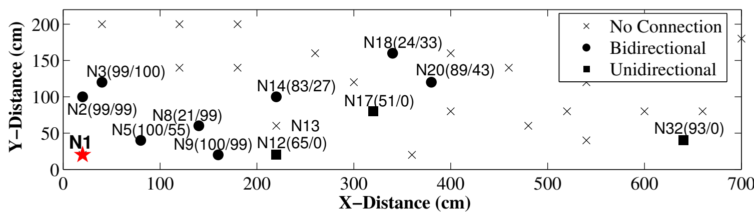
Due to the bottleneck links (further discussed in Section 5.5.4), in RBP4, the data packet with sequence number 7 is received by only 12 nodes in the network, which results in a deep drop of the total number of packets transmitted in Figure 14(b). As shown in Figure 14(c), the average dissemination delay of RBP8, RBP4, CF, and standard flooding is 4.46s, 3.93s, 2.85s, and 2.34s, respectively. The average delay of CF is 36% less than that of RBP8. As shown in Figure 11, the standard deviation of RBP8, RBP4, CF, and standard flooding is 1.9, 1.05, 1.6, and 0.5, respectively. Compared with the indoor experiments, in the outdoor experiment each node has fewer neighbors, resulting in more unbalanced transmissions among these nodes, which explains why the standard deviation of the outdoor experiment is larger.

5.5 System Insight Analysis

In the previous sections, we showed that CF has better performance than standard flooding and RBP. In this section, we explain why this is the case by revealing some system insights.

5.5.1 Number of Neighbors

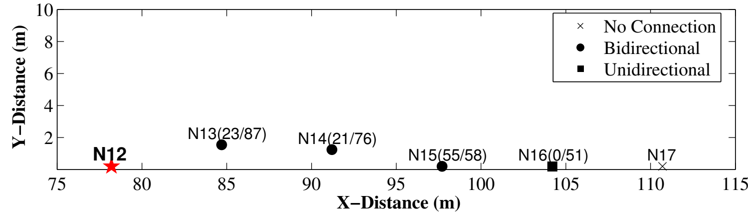
Figure 15(a) and 16(a) compare the CDF of each node's neighbor size when running CF and RBP for the multi-hop indoor and outdoor experiments. In RBP, two nodes are considered as neighbors if and only if the bidirectional link qualities between them are higher than a threshold. While in CF, there is no constraint on the link quality, thus nodes have more neighbors when running CF than when running RBP. The maximum number of neighbors that CF and RBP have is 17 and 9, respectively, for the indoor experiment, and 11 and 7, respec-



(a) Layout of Some Sensor Nodes in Indoor Experiment

(b) On–site View

Figure 17. Asymmetric Links in Indoor Experiment



(a) Layout of Some Sensor Nodes in Outdoor Experiment

(b) On–site View

Figure 18. Asymmetric and Bottleneck Links in Outdoor Experiment

tively, for the outdoor experiment. In CF, more neighbors indicates that more information can be utilized by the node to predict the coverage of its neighbors using *collective ACKs*.

5.5.2 Prediction Accuracy

In the previous section, we illustrated that CF has more 1–hop neighbors than RBP. In this section, we show that these 1–hop neighbors provide more information for the node running CF, which can more accurately predict whether its neighbors receive the packet. *Duplicated transmissions happen when the sender does not realize that the receiver already received the packet and retransmits the packet.* Both RBP and standard flooding do not utilize the information of conditional packet reception ratio to predict the packet reception of neighboring nodes, which results in a higher number of duplicated transmissions. Figure 15(b) and 16(b) show the CDF curve of the number of duplicated packets received for the same sequence number of the data packet by all the nodes running CF, standard flooding, and RBP, respectively. By tracing the logged data, we also include an Oracle solution, in which the node running CF, instead of doing the coverage probability estimation, stops transmission once its neighboring nodes receive the packet. From the figure, we can see that CF has a smaller number of duplicated packets received than does RBP in both the outdoor and indoor experiments. *The Oracle and CF curves are very close*, which indicates that the node running CF can accurately predict whether its neighbors have received the packet. Due to the accurate prediction, CF achieves the same reliability as RBP, while it has fewer duplicated transmissions.

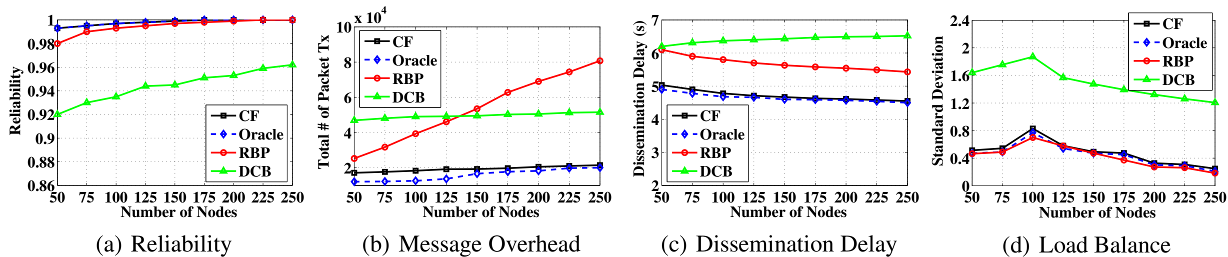
5.5.3 Efficiency in Delivery Paths

To trace the experiment, hop count information was attached to the data packet. Figure 15(c) and 16(c) shows the CDF of the number of hops the data packets traveled in order to reach the node. In CF, the node with the largest transmission effectiveness has the smallest back–off timer. This back–off mechanism significantly reduces the number of hops the data packets traveled. For example, the maximum number of hops for CF and RBP is 4 and 6, respectively, in the indoor experiment, and 9 and 13, respectively, in the outdoor experiment. Standard flooding has a similar path length as RBP, but in the outdoor experiment the maximum number of nodes covered by standard flooding was 41 out of 48. That is why the FLD curve terminates in the upper–right corner of Figure 16(c). Due to the smaller number of hops traveled, CF has a smaller dissemination delay than RBP.

5.5.4 Asymmetric and Bottleneck Links

Figure 17 shows the link qualities between node $N1$ and all its neighboring nodes. In the figure, a cross represents a node that does not have a direct link with $N1$; a square represents a node that has a unidirectional link with $N1$; and a round dot represents a node that has bidirectional links with $N1$. We use $N8(21/99)$ to represent that the link quality from $N1$ to $N8$ is 21%, while the link quality from $N8$ to $N1$ is 99%. Similar notation is used for all the other nodes. From the figure, we find a large number of asymmetric or even unidirectional links. If we run RBP, $N1$ selects only three nodes ($N2$, $N3$, and $N9$) as neighbors that have bidirectional link qualities higher than the threshold (60%). We also note that some nodes, such as $N13$, have shorter distances to $N1$ than $N17$ but have no connection to $N1$.

A similar phenomenon also happens in the outdoor experiment. Figure 18 shows the link qualities between node $N12$ and some of its neighboring nodes. Due to the environmental effect, the link quality from $N12$ to $N13$ is only 23%, while the backward link quality from $N13$ to $N12$ is 87%. Moreover, although the physical distance between $N12$ and $N13$ is closer than the distance between



(a) Reliability (b) Message Overhead (c) Dissemination Delay (d) Load Balance

Figure 19. Impact of Node Density

$N12$ and $N15$, the link quality from $N12$ to $N13$ is lower than the link quality from $N12$ to $N15$, which is 55%.

Since the bidirectional link qualities between node $N12$ and nodes $N13$ and $N14$ are below the thresholds of RBP4 and RBP8, which are 50% and 30%, respectively, both RBP4 and RBP8 exclude $N13$ and $N14$ from $N12$'s neighbor table. This introduces two effects: (i) $N13$ and $N14$ may not receive the packet from $N12$, leading to (ii) a bottleneck link between $N12$ and $N15$, because in RBP, the node only retries up to the maximum number of retransmissions if it does not hear the downstream node's ACK. For RBP4, the maximum number of retransmission is 4, meaning that in this specific topology, there is still a 4.1% probability that $N15$ will not receive the packet after $N12$'s 4th retry. As shown in Figure 14(b), the data packet with sequence number 7 could not be received by $N15$; thus the packet stopped propagation in the network.

As discussed in Section 3.1, the CF protocol can overcome the difficulties brought about by asymmetric links through the information on link quality and conditional packet receptions. In this scenario, as a receiver, $N12$ can estimate the packet reception probability of $N13$ and $N15$ based on overhearing $N14$'s transmission because $N14$ has larger transmission effectiveness than $N15$. Therefore, $N14$ transmits earlier than $N15$. Moreover, as a sender, $N12$ can also estimate the packet reception probabilities of $N13$ and $N15$ based on $N12$'s own transmission.

6 Simulation Evaluation

In order to understand the performance of the proposed CF scheme under numerous network settings, in this section we provide extensive simulation results. We compared the performance of CF with the following two state-of-the-art solutions and an Oracle approach:

- **Reliable Broadcast Propagation (RBP)** [34] by F. Stann et al. in SenSys'06.
- **Double-Covered Broadcast (DCB)** [18] by Wei Lou and Jie Wu in INFOCOM'04. In DCB, every node maintains 2-hop neighbor information. When a sender broadcasts a packet, it greedily selects the forwarders from its 1-hop neighbor set based on two criteria: (i) the rebroadcasts by the forwarders cover all the sender's 2-hop neighbors, and (ii) the sender's 1-hop non-forwarder neighbors need to be covered by at least two forwarders, including the sender itself.
- **Oracle:** In addition to the state-of-the-art solutions, we also include a theoretical "best-case" bound provided by an Oracle. In the Oracle approach, we assume there exists a perfect cost-free ACK in CF, so instead of doing the coverage probability estimation, the node will exactly know whether or not its neighbors have received the packet. We note that the Oracle approach is not optimal, but it serves as a good baseline.

6.1 Simulation Setup

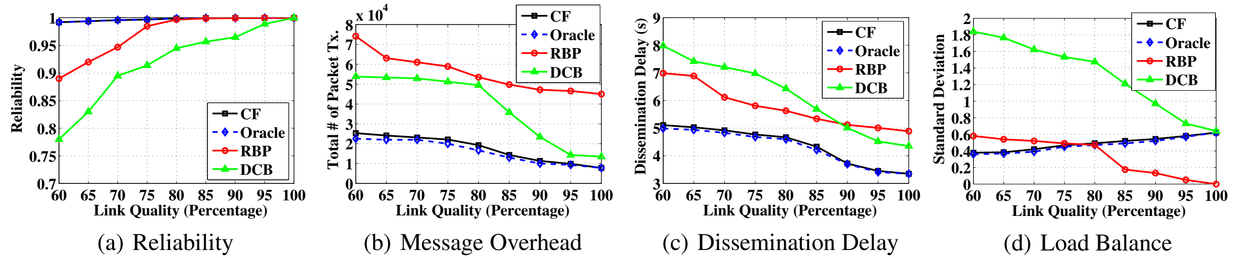
We simulated our design with ns-2. Our simulation MAC layer provided multiple access with collision avoidance. The MAC layer worked in the broadcast mode with no ACKs and retransmissions. The radio model was implemented based on our empirical data, which has the CPRP feature as described in Section 2.

In the simulation, we randomly deployed 250 sensor nodes in a 200m$\times$200m square field. A source node was positioned near the boundary of the field, and the source sent out the data packet with 29 bytes payload every 10 seconds. The total simulation time was set at 3200 seconds. In order to avoid the initialization bias of the system state on the broadcast operation, the source did not send out the data packet in the first 100 seconds, but exchanged only hello messages between neighboring nodes to establish the neighborhood information. Similarly, to make sure that all the broadcast packets propagated throughout the network, the source stopped sending out the data packet after 3100 seconds. Every data point on a graph represents the averaged value of 10 runs, and 95% confidence intervals for the data are within 2$\sim$8% of the mean shown. Unless explicitly stated otherwise, we used the above default values in our simulation.

6.2 Impact of Node Density

In this experiment, we analyzed the effect of node density by varying the number of nodes in the field from 50 nodes to 250 nodes.

Figure 19(a) shows that the reliability of all the protocols increases as the network density increases. When the node density varies, CF has more than 99% reliability, while the mean value of reliability of DCB varies from 0.92 to 0.962. This is because DCB uses only two forwarding nodes to cover the non-forwarding nodes. If the transmissions from both of the forwarders failed, the non-forwarding node will not be covered. When the node density is low, RBP has lower reliability than CF. The reason is that in RBP two nodes are considered as neigh-



(a) Reliability (b) Message Overhead (c) Dissemination Delay (d) Load Balance

Figure 20. Impact of Lossy Links

bors only when the link quality between them is larger than a set threshold (60% according to [34]). In a sparse network, some nodes in a sparse area may not be considered as neighboring nodes of any other nodes in the network. Those in sparse area nodes thus will have a lower chance of receiving the packet from their neighbors when running RBP. However, in CF the node estimates its neighbors' packet receptions based on the link quality information. As long as these nodes are physically connected, CF can provide high reliability.

Figure 19(b) shows that when the network density increases, the total number of packets transmitted linearly increases in RBP but slightly increases in both DCB and CF. The reason is that in RBP, every node needs to retransmit the packet that it receives for the first time. DCB does not provide an optimal algorithm such that the number of forwarders is the minimum and the non-forward nodes are covered exactly by only two forwarders. In DCB, when the network density increases, the number of forwarders slightly increases, which results in an increase in the total number of packet transmissions. In CF, when the network density increases, every node needs to cover more neighbors, which results in the slight increase in the total number of packets transmitted. More neighboring nodes also helps the node predict its neighbors' coverage probability, which results in the decrease in the gap between CF and Oracle in Figure 19(b).

Figure 19(c) shows that when the network density increases, the end-to-end delay decreases in CF and RBP but increases in DCB. This is because when the network density increases, the number of retransmissions used in CF and RBP decreases, while in DCB, when the network density increases, the number of forwarders slightly increases. Every forwarder needs to do back-off and retransmits the packet if it does not hear the retransmissions from its successors that are selected as forwarders.

Figure 19(d) shows that when the network density increases, the standard deviation of the number of data packets transmitted per node for all the protocols first increases and then decreases. This is because when the node density increases, the network will switch from *forwarder dominating* to *non-forwarder dominating* for all the protocols. DCB has the highest deviation, which is because it always selects the forwarders to do the retransmission. In high-density networks, the number of transmissions needed by the nodes is balanced, but RBP has a slightly lower standard deviation than CF. This is because RBP requires every node to do retransmission at least once, while in CF some nodes do not need to do retransmission if *collective ACKs* provide sufficient evidence that their neighboring nodes are covered already.

6.3 Impact of Lossy Links

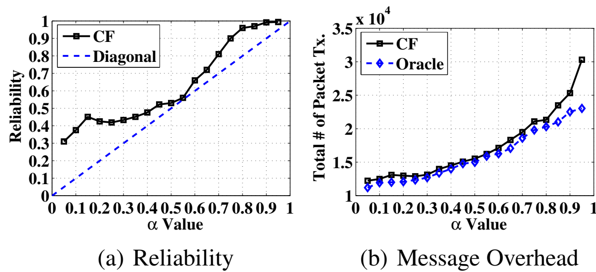
In this experiment, we analyze the effects of varying link qualities. The average link quality varies from 60% to 100%.

Figure 20(a) and Figure 20(b) show that as the link quality increases, the total number of packets transmitted decreases while the reliability increases for all the protocols. When the average link quality is 60%, the total number of packets transmitted by CF, DCB, and RBP is 25323, 53812, and 74132, respectively, while the mean value of reliability is 0.992, 0.78, and 0.89 for CF, DCB, and RBP, respectively. The total number of packets transmitted in RBP is 2.9 times more than that in CF. In order to let DCB achieve higher reliability, we set the maximum number of retransmissions at 4 for the forwarding nodes, which causes the flat period of DCB in Figure 20(b) when the link quality increases from 60% to 80%.

Figure 20(c) shows that the end-to-end delay decreases for all the protocols as the link quality increases. This is because the better the link quality, the fewer backoffs and retries needed by all the protocols. Figure 20(d) shows that when the link quality increases, the standard deviation decreases for RBP and DCB. For RBP, when the link quality is 100%, every node still needs to retransmit the packet that is received for the first time, which results in the standard deviation value of 0, while for CF, the standard deviation increases as the link quality increases. This is because as link quality increases, the nodes that do retransmission become centralized. When the link quality equals 100%, CF becomes the protocol that relies on the nodes with high connectivity to do the retransmission.

6.4 Impact of the Reliability Threshold

CF uses α to control the reliability desired by the users. Technically, the α value is the threshold that is used by each node to check whether its neighbors can be considered as covered. In this section, we evaluate the impact of α value. The total number of nodes in our sim-



(a) Reliability (b) Message Overhead

Figure 21. Impact of α Value

ulation is 150. α value varies from 0.05 to 0.95 with step 0.05. Figure 21(a) shows that the reliability curve of CF is above the diagonal line, indicating that CF satisfies the user requirement well. We note that the closer the reliability curve of CF from the diagonal line, the better CF can track the requirement. Figure 21(a) shows the largest difference between desired and actual α values is about 30.1% when $\alpha = 0.15$. The average difference is about 12.9%, which is a satisfactory performance under high link dynamics.

Figure 21(b) shows that as the α value increases, the total number of packets transmitted also increases. When the α value increases from 0.55 to 0.95, the gap between CF and Oracle also increases. For example, when the α value is 0.95, the difference in the total number of packets transmitted between CF and Oracle is 7280.

Figure 21(a) and 21(b) hint that setting α value to be 0.9 is a good choice for reliable flooding. It achieves the reliability of 0.992 with 4987 fewer packets transmitted than when the α is 0.95, indicating that approaching 100% reliability would be prohibitively expensive under unreliable communication environments.

7 State of the Art

The literature in flooding protocol designs can be classified into two categories: deterministic approaches and probabilistic approaches.

In the deterministic approaches, a fixed node within a connected dominating set is determined as a forwarding node. These approaches are also called fixed-forwarder approaches. In these approaches, the connected dominating set is calculated by using global or local information. It has been proved [20] that the creation of a minimum connected dominating set (MCDS) is NP-complete, so most approaches [12, 18] attempt to find a good approximation to the MCDS. Double-Covered Broadcast (DCB) [18] provides high reliability when the packet loss ratio is low. Compared with CF, DCB introduces more overhead to maintain 2-hop neighbor information. Its reliability is affected by the link quality between the forwarders.

In a probabilistic approach, when a node receives a packet, it forwards the packet with probability p. The value of p is determined by relevant information gathered at each node. Simple probabilistic approaches, such as [24], predefine a single probability for every node to rebroadcast the received packet. When running the above protocols in a network with different node densities, the nodes in a dense area may receive a lot of redundant transmissions. More complicated and efficient protocols, such as distance-based and location-based [40] schemes, use either area or precise position information to reduce the number of redundant transmissions.

Although the probabilistic scheme without ACKs provides a good stochastic result, it has relatively low reliability under unreliable wireless environments. The gossip-based approach [13] provides high reliability, using multiple rounds of message exchanges. Moreover, instead of overhearing, it needs the exchange of metadata. For applications, such as reprogramming an entire sensor network, perfectly propagating the code to all the nodes in the network is required. Trickle [26] uses gossiping and link-layer broadcasting to propagate small code updates. RBP [34] is used for the applications such as routing and resource discovery. In ADB [35], the node uses the footer in DATA and ACK frames to make the rebroadcast decisions. However, under unreliable wireless environment, the loss of an ACK from a receiver will cause the sender to treat the receiver as a 100% uncovered node and redundant transmissions are conducted. Moreover, such explicit ACKs may cause collision [11] in dense networks.

As a flooding protocol, CF is different from ExOR [3] which is a data forwarding protocol. In ExOR, all the nodes work together to forward the data from a source to a single destination. The “batch map” used in ExOR serves as an explicit ACK for each received packet, which is different from the *collective ACKs* which are achieved in an accumulative manner.

Despite this rich literature, the existing approaches do not exploit link correlation for performance improvement. Using link correlation, *collective ACK* becomes the key difference between CF and previous work. More specifically, the CF protocol has two new contributions: (i) instead of using an implicit or explicit ACK [1], each node dynamically estimates and accumulates its neighbors’ coverage status through *collective ACKs* by using the correlation of packet receptions among neighboring nodes; (ii) the forwarders are dynamically selected in a distributed fashion based on the nodes’ realtime estimations of their neighbors’ packet reception status. These two features lead to reliable and efficient message dissemination. Finally, we note that the concept of *collective ACKs* is independent of specific protocol designs. It could be used as an add-on feature to other routing [19, 7, 36, 44] and flooding [9] protocols. We leave this as future work.

8 Conclusions

In this paper, we propose CF to provide efficient and reliable message dissemination service with low complexity. We demonstrate that CF is effective through two

main mechanisms: *collective ACKs* and dynamic forwarder selection. Both mechanisms take advantage of link correlation among neighboring receivers. This is the first work that transforms the direct ACKs per receiver into a collective one. This unique design noticeably reduces the redundancy in rebroadcasting, as shown in our evaluation. We fully implemented and evaluated the CF protocol in several testbeds including a single hop network with 20 MICAz nodes, a multi-hop network with 37 MICAz nodes, and a linear outdoor network with 48 nodes along a 326-meter-long bridge. We also performed extensive simulation with various network configurations to reveal its performance at scale. The results show that the CF protocol has low overhead, low dissemination delay, and high reliability in unreliable wireless environments. Conceptually, the design of CF protocol is generic enough to be applied to wireless mesh networks and other stationary wireless networks. However, it is necessary to systematically investigate the implication of running CF in these types of networks in the future.

9 Acknowledgements

This work was supported in part by NSF grants CNS-0917097, CNS-0845994, and CNS-0626609. We also received partial support from InterDigital and Microsoft Research. We thank our shepherd Dr. Philip Levis and the reviewers for their valuable comments.

10 References

[1] M. Afanasyev, D. G. Andersen, and A. C. Snoeren. Efficiency through Eavesdropping: Link-layer Packet Caching. In *NSDI '08*.

[2] B. Hull, K. Jamieson, and H. Balakrishnan. Mitigating Congestion in Wireless Sensor Networks. In *SenSys '04*.

[3] S. Biswas and R. Morris. ExOR: Opportunistic Multi-Hop Routing for Wireless Networks. In *SIGCOMM '05*.

[4] C. Reis, R. Mahajan, D. Wetherall and J. Zahorjan. Measurement-based Models of Delivery and Interference. In *SIGCOMM '06*.

[5] D. Gay, P. Levis, R. Behren, M. Welsh, E. Brewer, and D. Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *PLDI '03*.

[6] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis. Collection Tree Protocol. In *SenSys '09*.

[7] Y. Gu, T. Zhu, and T. He. ESC: Energy Synchronized Communication in Sustainable Sensor Networks. In *ICNP '09*.

[8] R. Gummadi, D. Wetherall, B. Greenstein, and S. Seshan. Understanding and Mitigating the Impact of Rf Interference on 802.11 Networks. In *SIGCOMM '07*.

[9] S. Guo, Y. Gu, B. Jiang, and T. He. Opportunistic Flooding in Low-Duty-Cycle Wireless Sensor Networks with Unreliable Links. In *MobiCom '09*.

[10] J. Considine, F. Li, G. Kollios, and J. W. Byers. Approximate Aggregation Techniques for Sensor Databases. In *ICDE '04*.

[11] J. Padhye, S. Agarwal, V. N. Padmanabhan, L. Qiu, A. Rao, and B. Zill. Estimation of Link Interference in Static Multi-hop Wireless Networks. In *IMC '05*.

[12] J. Wu, W. Lou, and F. Dai. Extended Multipoint Relays to Determine Connected Dominating Sets in MANETs. In *IEEE Transcations on computing*, 55(3): 334-347, 2006.

[13] L. Alvisi, J. Doumen, R. Guerraoui, B. Koldehofe, H. Li, R. van Renesse, G. Tredan. How Robust Are Gossip-Based Communication Protocols? In *SIGOPS Operating Systems Review 41,5*, 2007.

[14] L. Luo, T. F. Abdelzaher, T. He, and J. A. Stankovic. Envirosuite: An environmentally immersive programming framework for sensor networks. In *TECS*, 5(3): 543-576, 2006.

[15] C. Lenzen, P. Sommer, and R. Wattenhofer. Optimal Clock Synchronization in Networks. In *SenSys '09*.

[16] C. M. Liang, J. Liu, L. Luo, A. Terzis, and F. Zhao. DCNet: A High-Fidelity Data Center Sensing Network. In *SenSys '09*.

[17] K. Lin and P. Levis. Data Discovery and Dissemination with DIP. In *IPSN '08*.

[18] W. Lou and J. Wu. Double-Covered Broadcast (DCB): A Simple Reliable Broadcast Algorithm in MANETs. In *INFOCOM '04*.

[19] M. Caesar, M. Castro, E. Nightingale, G. O'Shea, and A. Rowstron. Virtual Ring Routing: Network Routing Inspired by DHTs. In *SIGCOMM '06*.

[20] M. V. marathe, H. Breu, H.B. Hunt III, S.S. Ravi, and D.J. Rosenkrantz. Simple Heuristics for Unit Disk Graphs. In *Networks*, 25: 59-68, 1995.

[21] A. Miu, H. Balakrishnan, and C. E. Koksal. Improving Loss Resilience with Multi-Radio Diversity in Wireless Networks. In *MobiCom '05*.

[22] M. Z. Murtaza, J. Heidemann, and F. Stann. Studying the Spatial Correlation of Loss Patterns Among Communicating Wireless Sensor Nodes. In *Technical Report ISI-TR-2007-626, USC/Information Sciences Institute*, 2007.

[23] N. Xu, S. Rangwala, K. K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin. A Wireless Sensor Network for Structural Monitoring. In *SenSys '04*.

[24] S. Ni, Y. Tseng, Y. Chen, and J. Sheu. The Broadcast Storm Problem in a Mobile Ad Hoc Network. In *MobiCom '99*.

[25] P. Levis, and D. Culler. Mate: A Tiny Virtual Machine for Sensor Networks. In *ASPLOS X*, 2002.

[26] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code maintenance and propagation in wireless sensor networks. In *NSDI '04*.

[27] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. TinyOS: An Operating System for Wireless Sensor Networks. In *the book Ambient Intelligence*, 2005.

[28] J. Polastre, J. Hill, and D. Culler. Versatile Low Power Media Access for Wireless Sensor Networks. In *SenSys '04*.

[29] R. Fonseca, S. Ratnasamy, J. Zhao, C. T. Ee, D. Culler, S. Shenker, and I. Stoica. Beacon-Vector Routing: Scalable Point-to-Point Routing in Wireless Sensor Networks. In *NSDI '05*.

[30] S. Kim, S. Pakzad, D. Culler, J. Demmel, G. Fenves, S. Glaser, and M. Turon. Health Monitoring of Civil Infrastructures Using Wireless Sensor Networks. In *IPSN '07*.

[31] K. Srinivasan, P. Dutta, A. Tavakoli, and P. Levis. An Empirical Study of Low Power Wireless. In *SING Tech Report, SING-08-03*, Oct. 2008.

[32] K. Srinivasan, M. Jain, J. Choi, T. Azim, E. Kim, P. Levis, and B. Krishnamachari. The k-Factor: Inferring Protocol Performance Using Inter-link Reception Correlation. In *Technical Report SING-09-02*.

[33] K. Srinivasan, M. Kazandijeva, S. Agarwal, and P. Levis. The β-factor: Measuring Wireless Link Burstiness. In *SenSys '08*.

[34] F. Stann, J. Heidemann, R. Shroff, and M. Z. Murtaza. RBP: Robust Broadcast Propagation in Wireless Networks. In *SenSys '06*.

[35] Y. Sun, O. Gurewitz, S. Du, L. Tang, and D. B. Johnson. ADB: An Efficient Multihop Broadcast Protocol Based on Asynchronous Duty-Cycling in Wireless Sensor Networks. In *SenSys '09*.

[36] T. Zhu and M. Yu. A Dynamic Secure QoS Routing Protocol for Wireless Ad Hoc Networks. In *Sarnoff '06*.

[37] T. Zhu, Z. Zhong, Y. Gu, T. He, and Z. Zhang. Leakage-Aware Energy Synchronization for Wireless Sensor Networks. In *MobiSys '09*.

[38] V. Singhvi, A. Krause, C. Guestrin, J. James, H. Garrett, and H. S. Matthews. Intelligent Light Control Using Sensor Networks. In *SenSys '05*.

[39] K. Whitehouse and D. Culler. A Robustness Analysis of Multi-hop Ranging-based Localization Approximations. In *IPSN '06*.

[40] Y. B. Ko, and N. Vaidya. Flooding-based Geocasting Protocols for Mobile Ad Hoc Networks. In *MONET*, 2002: 471-480.

[41] W. Ye, F. Silva, and J. Heidemann. Ultra-Low Duty Cycle MAC with Scheduled Channel Polling. In *SenSys '06*.

[42] Z. Zhong, T. Zhu, D. Wang, and T. He. Tracking with Unreliable Node Sequence. In *INFOCOM '09*.

[43] G. Zhou, T. He, and J. A. Stankovic. Impact of Radio Irregularity on Wireless Sensor Networks. In *MobiSys '04*.

[44] T. Zhu and M. Yu. A Secure Quality of Service Routing Protocol for Wireless Ad Hoc Networks. In *GLOBECOM '06*.

Supporting Demanding Wireless Applications with Frequency-agile Radios

Lei Yang, Wei Hou[†], Lili Cao, Ben Y. Zhao, Haitao Zheng
Department of Computer Science, University of California, Santa Barbara
[†]*Department of Electronic Engineering, Tsinghua University*
{*leiyang, lilicao, ravenben, htzheng*}*@cs.ucsb.edu, hou-w05@mails.tsinghua.edu.cn*

Abstract – With the advent of new FCC policies on spectrum allocation for next generation wireless devices, we have a rare opportunity to redesign spectrum access protocols to support demanding, latency-sensitive applications such as high-def media streaming in home networks. Given their low tolerance for traffic delays and disruptions, these applications are ill-suited for traditional, contention-based CSMA protocols.

In this paper, we explore an alternative approach to spectrum access that relies on frequency-agile radios to perform interference-free transmission across orthogonal frequencies. We describe *Jello*, a MAC overlay where devices sense and occupy unused spectrum without central coordination or dedicated radio for control. We show that over time, *spectrum fragmentation* can significantly reduce usable spectrum in the system. Jello addresses this using two complementary techniques: *online spectrum defragmentation*, where active devices periodically migrate spectrum usage, and *non-contiguous access*, which allows a single flow to utilize multiple spectrum fragments. Our prototype on an 8-node GNU radio testbed shows that Jello significantly reduces spectrum fragmentation and provides high utilization while adapting to client flows' changing traffic demands.

1 Introduction

The future is bright for next-generation wireless devices. While current technologies are limited to operating in fixed ranges of increasingly congested spectrum, reforms in spectrum management policy promise to free up spectrum in the near future. The Federal Communications Commission (FCC) has auctioned recently vacated wireless spectrum to service providers [9]. To further democratize the use of this spectrum, online spectrum trading services such as SpecEX (*www.spectrumbridge.com*) now allow small service providers to purchase/rent spectrum directly from regional owners.

Unlike unlicensed bands used by current wireless devices, these new spectrum ranges are large and uncongested. We can take advantage of the opportunity to redesign access mechanisms to support a broader range of wireless applications. For example, current wireless access mechanisms are designed for best effort traffic, and generally rely on spectrum contention as used in CSMA protocols and their variants. The network partitions spectrum into fixed channels, lets each transmission choose a channel and contend in time with its peers. While this approach works quite well for file transfers and interactive applications, past work shows that supporting applications with real-time requirements requires additional modifications that incur significant overheads [25, 27].

In this paper, we reconsider the design of spectrum access mechanisms in dynamic spectrum networks to support applications within more restrictive traffic classes. Specifically, we consider supporting applications with strong quality of service requirements such as high-definition multimedia flows in media rich environments like the home. Traffic demands for these flows can vary significantly over time, but can generally be predicted ahead of time. Unlike best-effort traffic applications, these multimedia flows require dedicated spectrum access to minimize disruptions to their transmissions and to maintain the expected quality of user experience.

We make two observations that make existing contention-based systems unsuitable for these applications. First, *per-packet contention* produces frequent and unpredictable transmission disruptions, which would interfere with our desired traffic delivery constraints. In contrast, if multiple transmissions were allocated isolated frequencies, each flow would obtain necessary dedicated spectrum, while avoiding costly interference that traditionally leads to contention and communication delays [18]. Second, splitting spectrum into *fixed channel partitions* is also unattractive for applications with time-varying bandwidth demands. Fixed partitions prevent flows from using or releasing available spectrum as necessary, and would lead to inefficient spectrum usage [8]. In this respect, new hardware in the form of *frequency-agile radios* can be extremely useful. With these radios, a

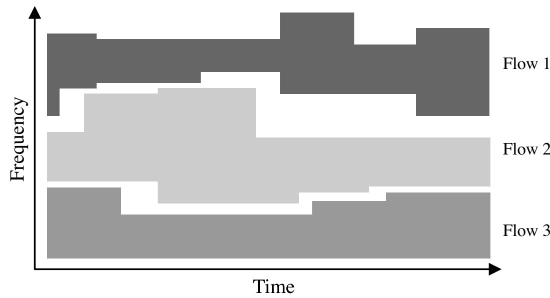


Figure 1: Per-session FDMA: Simultaneous media sessions work in parallel on isolated frequencies, avoiding costly wireless interference while adapting frequency usage to varying traffic demands.

device examines locally available spectrum before each network connection, and directs its radio to operate on a frequency range that not only matches its traffic demands, but also lies orthogonal to existing transmissions. In addition, devices can grab and release spectrum as necessary without being confined by fixed partitions.

Motivated by these observations, we propose a new distributed access technique that lets flows access spectrum in the frequency domain and adapt their spectrum usage based on traffic demands (shown in Figure 1). We refer to this new access technique as "per-session FDMA," where each session refers to a single continuous flow, and build a basic framework where traffic flows can independently select and adapt their frequency usage. First, by detecting "edges" on observed power spectrum maps, each device can accurately and quickly identify free spectrum in its local area. Second, each device can select an available spectrum range based on its present traffic demands, using classical algorithms such as *best fit*, *worst fit*, and *first fit* [19]. Finally, we propose a distributed coordination procedure to synchronize sender and receiver pairs in their spectrum usage.

Several recent proposals describe systems that adapt spectrum usage based on bandwidth demands [15, 20, 32]. In this context, our work builds an efficient framework that determines how device pairs sense and coordinate their access in open spectrum ranges. Our system is MAC-agnostic: once devices obtain spectrum using our primitives, they can use any MAC.

Spectrum Fragmentation. Efforts to evaluate our basic design reveal another fundamental challenge. Over time, as individual transmissions enter and exit the network or adjust their spectrum usage, available spectrum becomes increasingly divided into a collection of discrete fragments. This "spectrum fragmentation" means that a significant portion of spectrum, while free, is effectively unusable because its fragments do not provide the minimum contiguous spectrum range required by new flows. Our experiments show that this artifact does exist in practice, and leads to significant performance degradation even for networks with very few parallel transmissions.

We propose two distinct, but complementary mechanisms to address this fundamental problem: *online spectrum defragmentation* at the spectrum access layer, and *noncontiguous frequency access* at the physical layer. With online spectrum defragmentation, each pair of communicating devices voluntarily defragment spectrum by moving to alternative frequencies, thereby optimizing spectrum availability for other sessions. These frequency moves occur periodically in a session or as flows adapt to changing spectrum demands. They are nearly instantaneous and transparent to neighboring flows. Given our emphasis on minimizing disruptions, however, this technique cannot completely remove spectrum fragmentation. As a complementary mechanism, we offer non-contiguous frequency access, where a radio can utilize multiple spectrum ranges in a single transmission. This provides support for high-bandwidth transmissions even in the presence of moderate levels of spectrum fragmentation. Our approach implements non-contiguous frequency access using a "distributed OFDMA" mechanism, which differs from prior approaches like SWIFT [24] that rely on CSMA to share spectrum among frequency-agile radios.

These two techniques work best in unison. Non-contiguous frequency access requires "frequency guard bands" between allocated frequency boundaries to eliminate cross frequency interference, similar to guard bands between WiFi channels. Since they are not usable for communication, guard bands represent spectrum overhead that increases as flows make use of more fragmented spectrum ranges. Online spectrum fragmentation, on the other hand, effectively suppresses the level of fragmentation.

The Jello Overlay. Based on these two complementary techniques, we design and implement Jello, a MAC overlay to support high-bandwidth real-time applications. Jello does not require centralized spectrum controllers or dedicated radios for control traffic, making it a low-cost and easily deployed solution. Jello radios sense, identify and occupy usable frequencies based on traffic demands while minimizing spectrum fragmentation. Where low levels of fragmentation remain, devices accommodate high-bandwidth transmissions using non-contiguous frequency access. We deploy a prototype of Jello on a 8-node USRP GNU radio testbed, and evaluate the benefits of online spectrum defragmentation and non-contiguous frequency access, both individually and together. Measurements show that Jello reduces disruptions to applications by as much as a factor of 8.

Our work makes three key contributions. First, we explore spectrum access techniques for real-time wireless applications with low tolerance for traffic disrup-

tions, and propose mechanisms for frequency-agile radios to sense, occupy, and synchronize spectrum usage. Second, we identify the spectrum fragmentation challenge, and propose two complementary solutions to maximize spectrum utilization. Finally, we implement and deploy a prototype of Jello, a complete MAC overlay encompassing our techniques. We evaluate the effectiveness of Jello mechanisms using both detailed measurements of an 8-node GNU-radio testbed and simulated experiments. Jello provides interference-free access to demanding applications while maximizing utilization of available radio spectrum, and can be deployed on hardware available today.

2 A Case for Per-session FDMA

The expected arrival of new wireless spectrum is an opportunity to redesign spectrum access protocols to support a richer set of network applications. In particular, available spectrum can be used to support "soft realtime" applications, *i.e.* applications such as multimedia streaming that have very low tolerance for data loss, delays and jitter.

Given their strong demands on the underlying wireless network, these applications do not perform well on CSMA protocols that require parallel flows to perform per-packet contention. Recent experimental results show that such contention leads to unpredictable network delays and disruptions [25, 27], ultimately resulting in visible disruptions to the application-level user experience. Quality of Service extensions such as IEEE 802.11e can prioritize traffic, but does not prevent contention between multiple flows in the same traffic class, *e.g.* video streams in neighboring houses. An alternative for predictable traffic delivery is to employ Time Division Multiplexing (TDM) to obtain a collision-free transmission schedule. However, this requires fine-grain network-wide time synchronization and scheduling, which are difficult to implement in practice.

Assumptions. Our focus is on supporting demanding wireless media applications. We assume that these applications operate in a dedicated spectrum band, generate continuous traffic with time-varying load, and have strong quality of service requirements. In environments where they must co-exist with legacy systems using best-effort traffic, we envision that local wireless spectrum can be partitioned into two ranges for isolation. One range is dedicated to legacy applications using 802.11 CSMA, and the other is dedicated for media-streaming applications running our proposed protocols.

Frequency-agile Radios. Recent hardware advances have produced "frequency-agile radios," wireless radios capable of operating across a wide range of frequencies and jumping between them in milliseconds. Currently available hardware includes the WARP [30], USRP [21], AirBlue [16] and SORA [28], with more expected in the next few years. With these radios, we can now consider *per-session FDMA*, or Frequency Division Multiplexing Access. In this approach, parallel sessions occupy orthogonal spectrum ranges, thus completely avoiding cross-flow interference. When a media session starts, the two end-devices involved choose a free frequency block to set up packet transmissions. As shown in Figure 1, flows can adapt their frequency usage over time as their bandwidth demands vary, thus using time multiplexing to make the best use of radio spectrum. Recent work [15] shows that adapting spectrum on demand leads to 75% improvement over 802.11b.

Our approach differs from the concept of adapting frequency bandwidth on conventional 802.11 devices [8], where 802.11 channels can change their width to 40, 20, 10 or 5MHz by adjusting clock cycles. Our experiments show that scaling up traffic to fixed channel widths can reduce utilization up to 30% in our application scenarios. In comparison, per-session FDMA operates across wider spectrum ranges at fine granularities to ensure high utilization, completely eliminates CSMA traffic contention. Furthermore, each link now can flexibly combine multiple spectrum ranges to form high bandwidth transmission. The proposed per-session FDMA can work on any of the current frequency-agile radio designs [2, 16, 21, 24, 28, 30]. Since our approach operates directly on frequency bands, and uses frequency selection to avoid access conflicts, we also differ from prior work [15] that uses pseudo-random spreading codes to implement random spectrum access.

Challenges. A practical per-session FDMA system for wide-spread deployment needs to support soft realtime applications without relying on centralized spectrum controllers or costly dedicated radios for control traffic. Such a system must address several key challenges. First, to avoid disrupting ongoing transmissions, devices must be able to accurately and quickly identify free frequencies. Second, each transmission pair needs to select a free spectrum block based on their traffic demand while minimizing spectrum fragmentation. They also must do so without disrupting other ongoing transmissions, and without the help of any control radio. Similarly when a transmission pair needs to change frequency usage to accommodate variations in traffic demand (those cannot be handled by MAC rate adaptation), they also need to make the process transparent to others.

3 Jello Framework

To address these challenges, we propose Jello, a lightweight MAC overlay system that realizes distributed per-

session FDMA. Jello radios sense, identify and occupy usable frequencies to support time-varying traffic demands and to avoid interfering with each other. Each Jello device has a single half-duplex frequency-agile radio for wireless communication, and does not require any central control or dedicated control radio.

3.1 Identifying Usable Spectrum

When accessing spectrum, Jello devices must avoid conflicting with other ongoing sessions. Jello achieves this by performing spectrum sensing to quickly and accurately identify usable spectrum ranges. Unlike the time-domain sensing approach [4], Jello uses a frequency-domain mechanism, benefiting from its radio hardware's frequency-agility. Unlike WiFi devices that sequentially scan channels, a frequency-agile radio can listen to the entire spectrum span, as demonstrated by several available radio platforms [24, 30]. Using the frequency-domain signal, each radio constructs a power spectral density (PSD) map [13] that measures the energy level on each small frequency range.

To identify usable frequency blocks, conventional approaches perform energy detection on the PSD map [10]. For a given threshold Γ_{energy}, each radio treats frequency ranges with energy higher than Γ_{energy} as busy and the rest as unoccupied. The detection accuracy, however, is shown to be highly sensitive to the choice of Γ_{energy} and finding a uniformly optimal Γ_{energy} is unrealistic [24]. Recent work proposes to cross-validate the detection result by "poking" transmissions on "busy" frequency ranges and observing their reactions [24]. Each poking event disrupts existing transmissions, forcing them to move to other frequencies or change their transmission parameters. Thus while this solution works for transmissions that are highly resilient to frequent disruptions, it would cause serious performance issues for the media sessions our system targets.

Sensing via Edge Detection. We exploit a unique property of radio transmissions in the frequency domain for accurate detection. To avoid interference to other transmissions, OFDM based transmitters use filters to limit the radio energy within certain frequency bands. As a result, the PSD profile of each transmission has clear edges on the frequency band boundaries, regardless of energy levels (shown in Figure 2). We can reliably identify usable frequency blocks by identifying these edges.

Our edge detection mechanism works as follows. First, as a pre-processing step, we smooth the PSD map by averaging it over multiple consecutive observations and applying two coarse power thresholds to filter out obvious frequency ranges. Frequency ranges with very high power are treated as busy and very low power ones as occupied. This pre-processing aims to filter out most noises in the PSD map before trying to locate edges. This technique has been sufficient in our experiments without using sophisticated smoothing algorithms like [5].

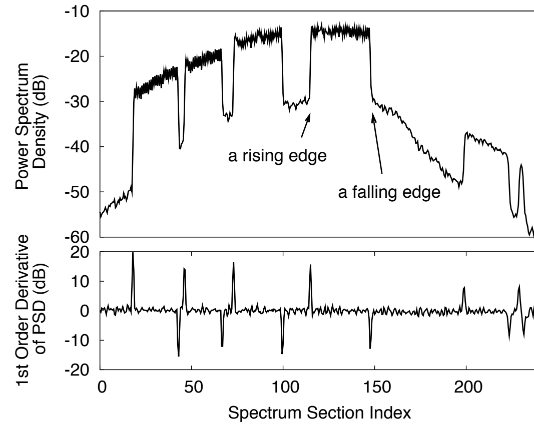


Figure 2: A sample PSD map and its first-order derivative. Jello identifies occupied frequency blocks using edge detection. While the absolute signal strength varies significantly across the frequency, the rising/falling edges are easier to detect.

Second, we apply search-based edge detection [14] and measure the edge strength by the first-order derivative of the PSD map. Let $P(k)$ represent the energy value of a spectrum section with index k, and let $P'(k)$ represent its first-order derivative. To decide whether edges are present, we choose a detection threshold Γ_{edge}. If $P'(k) > \Gamma_{edge}$ then k has a rising edge and if $P'(k) < -\Gamma_{edge}$ then k has a falling edge. A frequency block with a rising edge to its left and a falling edge to its right is declared as busy and the rest as free.

Compared to the energy detector, the edge-detection based sensing is less dependent on the choice of detection threshold. As shown in Figure 2, while the absolute signal strength varies significantly over the frequency, the rising/falling edges are easy to detect. This design works well in OFDM-based systems where the PSD map can capture frequency usage accurately and on-demand. While other forms of interference such as wireless microphones might not display the similar edges in the PSD map, we can incorporate other mechanisms such as feature detection based sensing for improved accuracy [4, 11]. In our target scenario, we focus on a homogeneous setting with radios all using OFDMA and within a short distance, thus our proposed sensing mechanism works well.

Calibrating Sender/Receiver Sensing Results. Each sender/receiver pair must synchronize their sensing results to identify mutually available frequency ranges. While the sender must pause its transmission to sense spectrum, the receiver senses while receiving at no extra

cost. Therefore, the receiver constantly monitors spectrum usage and locates occupied frequency ranges based on its maximum tolerable noise and interference level. When new spectrum blocks become available, it piggybacks the information via data or control packets to signal the sender to sense.

3.2 Choosing Frequency Blocks

After identifying mutually available frequency ranges, a sender/receiver pair needs to choose a frequency block to occupy. Such decisions usually occur when sessions start. It can also happen during a session when traffic changes cannot be handled by MAC rate adaptation. The device pair determines the amount of frequency needed based on estimated traffic demands and estimated MAC transmission rates on available frequency ranges. They can expand/shrink the current frequency usage, or move to a different frequency block. The ultimate goal is to obtain desired spectrum while maximizing system-wide usage efficiency.

The frequency selection problem is analogous to the online task scheduling problem [19]. Due to the unpredictable dynamics of spectrum demands, optimal solutions are hard to find. Similar problems have also been studied extensively in the context of CPU, memory and storage allocations. The most efficient known solutions apply heuristics-based algorithms [19], which have been shown to perform very well in most cases. In particular, we consider the well-known *best fit* strategy that selects the smallest available frequency block that can accept the current spectrum request, the *worst fit* strategy that uses the largest available block, and the *first fit* strategy that uses the first large enough block. When no block is large enough to satisfy a session's demand, we choose the largest block to accept the session partially. We found in our experiments that *best fit* outperforms others.

Propagation-aware frequency selection. In some cases, radio propagation conditions differ significantly across frequency ranges, *i.e.* due to channel fading. Information on received signal and interference strength, if available, can be integrated into Jello's frequency selection algorithm to select high-quality blocks that provide better reliability and higher bandwidth [4,23]. In the current Jello prototype, the propagation quality is flat across the frequency span considered, thus the receivers use the measured interference strength in their selection process.

4 Suppressing Spectrum Fragmentation

Efforts to evaluate our basic Jello design reveal another fundamental challenge. Over time, as individual transmissions enter and exit the network or adjust their spectrum usage, available spectrum becomes increasingly divided into a collection of fragments (Figure 3(a)). This is because each radio must access spectrum contiguously, *i.e.* using a single frequency block. In this case, although a significant portion of spectrum remains unoccupied, it is effectively unusable because no individual fragment is large enough for a new request. A similar fragmentation problem appears in disk and memory allocation [19]. In this section, we examine the severity and impact of spectrum fragmentation, and propose two distinct but complementary techniques to minimize it. We provide high-level descriptions of our proposed techniques and delay the detailed implementation issues to Section 5.

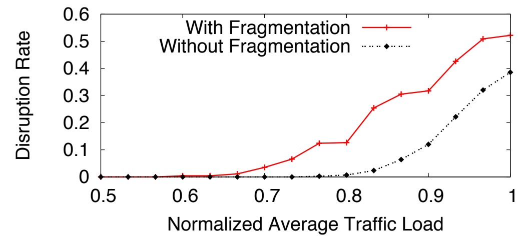


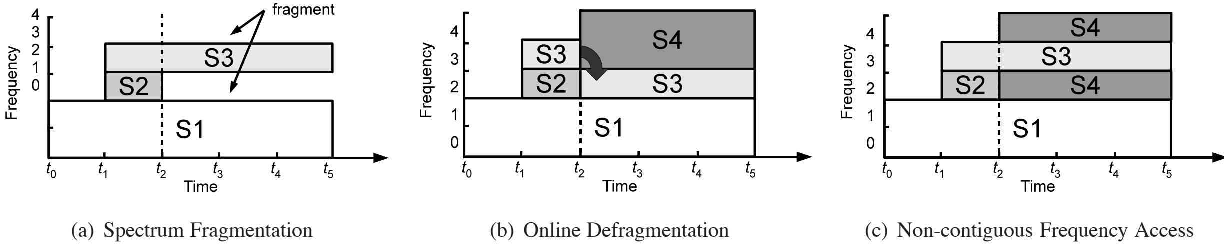
Figure 4: The impact of spectrum fragmentation with 4 streaming media sessions, using VBR video traces from the ASU trace database [3]. We compare the basic Jello system (with fragmentation) to an oracle system that eliminates all fragments.

4.1 Impact of Spectrum Fragmentation

To understand the severity and impact of spectrum fragmentation, we perform a detailed simulation using video traces from an online database [3]. Using a number of frame traces of H.263 video sessions, we simulate a scenario of multiple media sessions within close proximity. We measure the impact of spectrum fragmentation by *application disruption rate*, defined as the percentage of time a session cannot obtain enough spectrum to support $X\%$ of its present traffic demand.

We compare two possible frequency access systems: (1) an oracle system that rearranges sessions' frequency usage to defragment the spectrum completely; and (2) a basic Jello system where sessions "claim" their needed spectrum when they start, and do not change frequencies unless their spectrum demands change.

Figure 4 plots the application disruption rate for 4 variable bit rate (VBR) video sessions for $X = 90\%$. On the x-axis, we show the ratio of the total average traffic load of all 4 videos to the spectrum capacity. Clearly, the oracle system that fully defragments the spectrum performs significantly better when the 4 videos present a significant portion of all available spectrum. To guarantee that the disruption rate never rises above 3%, the basic system can only support traffic equal to 67% of the total



(a) Spectrum Fragmentation (b) Online Defragmentation (c) Non-contiguous Frequency Access

Figure 3: Spectrum fragmentation and ways to mitigate its impact. (a) When sessions share spectrum by accessing contiguous frequency, they can create spectrum fragments. (b) After $S3$'s self defragmentation, the same spectrum can now support more spectrum requests. (c) Session S4 uses two spectrum fragments for a single transmission.

spectrum capacity, while the oracle system can support traffic up to 83% of the spectrum capacity*. This is a significant boost in allowed traffic volume, and underlines the significant impact that fragmentation has on system performance.

4.2 Online Spectrum Defragmentation

The above results motivate us to improve Jello's basic design to suppress spectrum fragmentation. The first and most direct solution is to perform online defragmentation. A naive strawman version is to periodically perform global defragmentation where all sessions pause their transmissions, rearrange their frequency usage so that unoccupied frequency blocks are merged into a large contiguous range. This, however, is infeasible in our problem context because we must minimize disruptions to ongoing traffic flows. There is also no central controller to perform global defragmentation.

Instead, we propose an online, distributed approach to defragmentation: ongoing transmissions periodically consider moving to an alternative spectrum block using the *best-fit algorithm* to optimize overall spectrum availability. Each sender/receiver pair periodically senses local spectrum usage, and if possible, coordinates to switch to a frequency block that better optimizes the overall spectrum availability. For example, Figure 3(b) follows our earlier scenario where a session S_2 terminates and leaves a spectrum fragment. If S_3 voluntarily moves to spectrum block 2–3, the new request S_4 can be fulfilled and the overall spectrum utilization increases. Finally, using spectrum sensing to identify unoccupied frequency ranges, each device pair independently defragments spectrum without coordinating with other pairs.

Cost. The cost of our online defragmentation includes (1) the sensing and coordination overhead spent by device pairs to identify unoccupied spectrum and rearrange their frequency usage, and (2) possible conflicts when two sessions simultaneously defragment and make conflicting frequency adjustments.

Focusing on minimizing disruptions to ongoing sessions, Jello uses the following mechanisms to minimize defragmentation cost:

• *Minimizing Sensing/Coordination Overhead*: To minimize sensing overhead, Jello receivers constantly monitor spectrum to identify possibly opportunities for defragmentation. They signal their senders to perform sensing only after identifying possible opportunities themselves. To minimize coordination delay, each sender/receiver pair uses their present frequency block to exchange handshakes and schedule frequency adjustments. At initialization or during a unlikely event of lost synchronization or link failure, Jello devices enter a SYNC state to recover and resume communications.

• *Avoiding Defragmentation Conflicts*: Multiple devices can simultaneously detect a defragmentation opportunity and make conflicting frequency adjustments. Jello minimizes such conflicts by randomizing defragmentation efforts to avoid simultaneous adjustments.

4.3 Non-contiguous Frequency Access

Our second solution is to enable radios to combine multiple spectrum pieces to form a single transmission. Shown in Figure 3(c), S_4 now combines frequency block 2 and 4 together in a single transmission as if it uses a single frequency block.

Non-contiguous frequency access is now widely used in centralized wireless networks such as WiMAX and cellular LTE systems. It is implemented in the form of Orthogonal Frequency-Division Multiple Access (OFDMA). Existing designs, however, require global synchronization, and fail when applied to distributed networks without global synchronization. A key contribution of Jello is to identify and address the challenges of implementing *distributed OFDMA* and to prototype our design on USRP GNU radios.

Cost. To minimize interference, frequency guard bands

*The oracle cannot support 100% traffic load because the flows are VBR and the peak load occasionally exceeds the spectrum capacity.

must be placed at link boundaries [17]. Guard bands are not usable for transmissions, and are essentially spectrum overhead. Frequency guard bands are not an artifact of non-contiguous frequency access: they are required for contiguous access including 802.11 channels (which use 16% of frequency bandwidth as guard bands). On the other hand, the amount of guard bands increases when links start to use non-contiguous frequency blocks.

4.4 The Case for a Unified Approach

With the above two solutions, we ask the question: "*Is one solution sufficient enough to address spectrum fragmentation?*"

First, consider a scenario where only online spectrum defragmentation is available. While this technique improves spectrum utilization overall, each sender-receiver pair is acting independently, and cannot disturb other ongoing transmissions. Therefore, only a limited level of spectrum defragmentation is possible, and this technique cannot achieve the same effectiveness as a global, synchronized defragmentation approach. Thus a low level of fragmentation might remain.

Next consider a network using noncontiguous frequency access, but no online defragmentation. While this technique allows devices to utilize spectrum fragments as if they were a single contiguous fragment, it comes at the cost of multiple guard bands between link boundaries. Without online defragmentation, spectrum fragmentation will continue to degrade over time. Spectrum lost to guard bands will continue to increase, lowering overall spectrum utilization.

Clearly, neither technique by itself can fully address the challenge of spectrum fragmentation. Together they form a more complete solution. Online defragmentation limits spectrum fragmentation to a low level, and noncontiguous access makes all of the spectrum available without incurring significant overhead to guard bands.

5 Implementing Jello

We have implemented Jello on USRP GNU Radios. Despite having limited frequency bandwidth and large processing delays [12], USRP radios are widely available and fully reconfigurable across various protocol layers. We use the USRP implementation as a "proof-of-concept" evaluation of Jello. We modified GNU radio software to implement spectrum sensing, distributed contiguous and noncontiguous frequency access, online defragmentation, and sender/receiver coordination.

Figure 5 presents a high-level structure of Jello. At the physical layer, each Jello device operates on non-contiguous frequency ranges using distributed OFDMA.

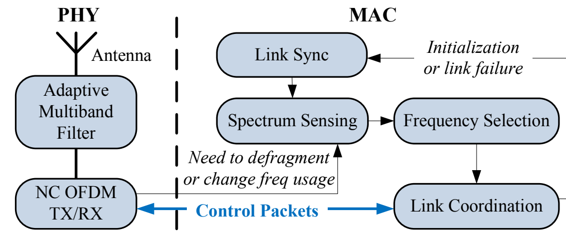


Figure 5: Jello system architecture.

At the MAC layer, Jello devices sense spectrum to identify usable frequency, and adapt their frequency usage when the application demand changes or when an opportunity to defragment appears. We now describe our implementation in detail.

5.1 Physical Layer

At the physical layer, Jello's key contribution is to implement spectrum sensing and distributed frequency access, both contiguous and non-contiguous, on today's common off-the-shelf hardware. Jello implements frequency access using OFDMA, which partitions the spectrum span into many small subcarriers. OFDMA has been widely used in centralized systems such as WiMAX, which divides a 20MHz frequency range into 2048 subcarriers of 10KHz each. Each sender can transmit on any subset of the subcarriers, either contiguously or non-contiguously aligned in frequency. Each receiver can listen to the *entire* set of subcarriers at once. Simultaneous transmissions can occur at different subcarriers without interfering with each other.

Implementing OFDMA on distributed networks, however, is hard. Existing designs in centralized networks rely on global synchronization to maintain subcarrier orthogonality, so that transmissions on isolated subcarriers do not interfere with each other. In distributed networks, where global synchronization is infeasible, OFDMA transmissions fail. To understand the causes, we perform an experiment by configuring 4 links on different frequency subcarriers. Our results show that significant link failures occur. The failures are not caused by the inherent propagation impairments, but by the following two reasons:

(1) *Unable to detect packet preamble*: In many cases, the receivers cannot detect any preamble that marks the beginning of a packet. This is because OFDMA detects preambles using a time-domain "delayed correlation" property from a signal placed at the head of each packet [29]. Because preambles from multiple transmissions are no longer synchronized, the delayed correlation property no longer holds in time-domain signals, preventing any successful preamble detection.

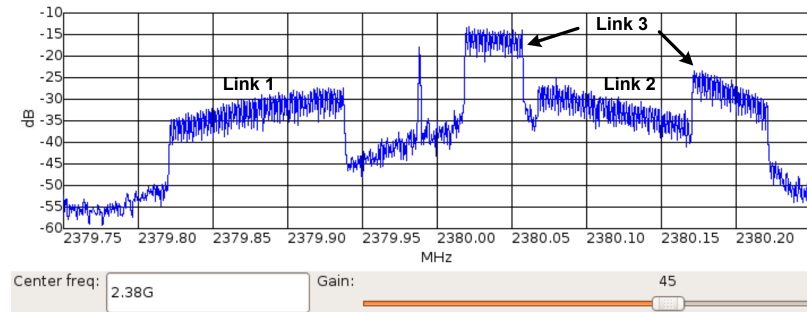


Figure 6: An example of Jello's flexible distributed spectrum access, implemented on USRP GNU radios. Three transmissions access and share radio spectrum in the frequency domain. Among them, link 3 operates on two non-contiguous spectrum blocks to form a single transmission.

(2) *Unable to decode data packet*: Even after fixing the preamble detection, significant losses still occur during packet decoding. This is because while multiple transmissions operate on different subcarriers, they leak energy to adjacent subcarriers, creating inter-carrier interference and destroying the subcarrier orthogonality at receivers. Compared to the preamble, packet data is much more vulnerable to interference because it is sent fewer error protections.

This motivates us to design receivers that "filter" out or minimize unwanted signals to restore the desired transmission properties. With this concept in mind, we propose two new mechanisms on top of the conventional OFDMA design to restore successful transmissions in distributed networks.

Restoring Preamble Detection. To restore the delay correlation property required for preamble detection, we apply an adaptive filter at receivers to remove signals from unwanted subcarriers. To support non-contiguous frequency access, we use a multi-band filter bank. Given the knowledge of the subcarriers used by its transmitter, the receiver first applies a low-pass filter to eliminate signals outside of its lowest and highest indexed subcarriers, and then uses multiple band-stop filters to remove signals from other unwanted subcarriers within the range. This design allows receivers to adapt filter ranges on-the-fly.

Without global synchronization, devices also experience frequency offset [13], defined as the frequency skew between devices' central carrier frequency. The presence of frequency offset could lead to errors in signal filtering. To suppress its impact between sender/receiver pairs, Jello receivers dynamically adjust their carrier frequency and filter width based on the result of preamble detection. At initialization it starts from a loose filter and gradually shrinks the filter to suppress interference. If the filter becomes too tight and fails to detect any preamble in a period, the receiver expands the filter to capture more subcarriers. After each successful preamble decoding, it estimates the frequency offset from its sender and refines the filter parameters.

Restoring Reliable Packet Receptions. While the use of receiver filters significantly improves preamble detection, packet losses can still occur due to out-of-band emissions among transmissions [17]. This work also shows that placing frequency guard bands between transmission boundaries is the most effective solution. To minimize these overheads, Jello devices directly measure interference power levels from the PSD map, and avoid using severely affected frequencies. This technique, combined with the adaptive filtering, allows Jello devices to correctly determine and minimize the usage of guard bands.

GNU Radio Implementation. We implement Jello's distributed OFDMA at 2.38GHz on a spectrum band of 500kHz. We use 256 subcarriers (or frequency sections), each of size 1.953kHz. To carry adequate signals for reliable preamble detection, each transmission must use at least 28 subcarriers, which can be non-contiguously aligned. We implement the receiver filter using the *hamming window* approach [22]. To compensate the frequency offset between sender and receiver, we initially extend the filter by 5 subcarriers and then adjust its central frequency and width on-the-fly. We found in our experiments that adding the receiver filter helps to reduce the amount of guard bands. Overall, placing 2 subcarriers at each link boundary is sufficient to protect all the links in our experiments.

Figure 6 illustrates an example PSD map of a system with three links. In this example, both link 1 and 2 occupy a contiguous block while link 3 utilizes two blocks simultaneously to build a high bandwidth transmission. Small guard bands were placed at link frequency boundaries to minimize cross-link interference.

We implement the spectrum sensing directly over OFDMA. Each device performs the Fast Fourier Transform (FFT) on collected frequency signals, and averages

the results over 50 OFDM symbols[†] to produce a PSD map. It computes the first-order derivative and uses a threshold of Γ_{edge} =5dB to locate edges. We chose these parameters because they work well in our experiments.

5.2 Access Layer

At the access layer, each Jello device will select frequency blocks to set up its communication session. During the session, it adapts its frequency usage when its traffic demand changes or when an opportunity for defragmentation appears. Without any dedicated radio for control, Jello addresses the following challenges: (1) each sender/receiver pair needs to synchronize on their frequency usage to ensure reliable transmissions; (2) to avoid hidden terminal problem, each sender/receiver pair needs to coordinate and choose proper frequency block(s) that are available to both of them; (3) simultaneous transmissions need to avoid using overlapping frequency blocks; and finally (4) devices must be able to quickly recover from failures caused by channel impairments and external interference.

Synchronizing Sender/Receiver. Each Jello sender and receiver pair performs handshaking to synchronize the frequency blocks they use for data transmission. This coordination has low overhead and does not involve any contention among sessions. Because GNU radios have large processing delays [12], our current Jello implementation does not include per-packet acknowledgements. The handshaking process is always initiated by the sender.

To change a session's spectrum usage, the sender performs spectrum sensing to see if there is any opportunity for change. If so, it sends a request (REQ) to its receiver indicating its spectrum sensing results. After receiving a REQ, the receiver selects a proper set of blocks and replies with an acknowledgement (ACK) indicating the selection. It also starts to decode signals from the new blocks. Upon receiving an ACK, the sender configures its transmissions on the new blocks. ACK failures could lead to discrepancy between sender and receiver's frequency usage. Thus, after failing to decode packets for a period of T_{BOFF}, the receiver "switches" back to decoding on the original blocks.

Choosing Frequency Blocks. Each Jello pair first tries to find a contiguous frequency block using the *best-fit* algorithm. If no such block is available, the pair selects multiple frequency blocks following the "noncontiguous best-fit" strategy: select the largest available blocks until the remaining demand is less than the largest remaining available blocks; then use *best-fit* to choose the final block. This approach minimizes the number of blocks required for the session.

Avoiding Conflicts. When an opportunity to defragment spectrum appears, multiple device pairs could react simultaneously, thus leading to frequency adjustments that conflict. To minimize these conflicts, we incorporate a random delay to both the sender's sensing function and receiver's defragmentation triggering. First, upon detecting a defragmentation opportunity, the receiver waits for a random interval T^R_{sense}, and notifies the sender only if the opportunity still exists. Second, a sender always repeats its spectrum measurement after a random delay of T^S_{sense}. A frequency block is considered free only if it is found to be free during both measurements. Random backoffs reduce the probability of simultaneous defragmentation attempts, similar to the CSMA backoffs in 802.11. Finally, devices can configure their backoff windows based on the projected effectiveness of their frequency shifts, giving priority to those that can provide the maximum benefit to the system. For simplicity, Jello uses a uniform random backoff window.

Recovering from Failures. Despite minimizing link failures through careful coordination of spectrum sensing and selection, link failures are sometimes unavoidable. They can occur from external interference or an unlikely conflict scenario where two links simultaneously move to the same frequency block. Redundancy techniques such as error correction codes [24] can improve the robustness of coordination packets, but are ineffective under complete link failures. If a link fails due to interference or conflict, its sender-receiver coordination messages will also fail to reach their destinations.

To address this, Jello introduces a SYNC state that devices enter at initialization or when they detect a coordination failure. A sender enters the SYNC state after failing to receive any ACK after retransmitting a REQ N_S times, and a receiver enters the SYNC state after not receiving any packets for a time period T_{SYNC}. In the SYNC state, devices communicate on the "SYNC Frequency Set" (SCS), a set of frequency blocks dedicated for performing resynchronization. The sender and receiver perform normal handshakes to reestablish synchronization and move to selected frequency block(s). There are several ways to define SCS, in our current implementation we configure it as a preassigned frequency block known to all devices. Devices try to avoid using the SCS for data transmissions except as a last resort, maximizing the probability that the SCS is idle.

GNU Radio Implementation. We implement Jello's access layer as a user-level program. Because USRP radios have a large random processing delay up to 20ms [12, 21], we use relatively large timing parame-

[†]The typical OFDM symbol duration for 802.11 a/g radios is 4μs, so the sensing time is 0.2ms. In GNU radios, the symbol duration is 2ms and the sensing time is 100ms.

ters in our experiments: T^S_{sense} and T^R_{sense} are uniformly distributed in $[0.1s, 1s]$, $T_{BOFF} = 1s$, $N_S = 5$, and $T_{SYNC} = 3s$. Each Jello device tries to defragment the spectrum once every $10s$. We choose the 28 lowest indexed subcarriers (out of 256) as the SCS. Based on our experience with the experimental platform, we found these to be reasonable parameter values.

5.3 Unexpected Hardware Artifacts

We also observe two unexpected hardware artifacts that may affect Jello's testbed performance.

Amplified Impact of Frequency Offsets. The bandwidth limitation of USRP radios magnifies the impact of frequency offsets, because they are now larger than the subcarrier width. Our 20-day measurements show that the frequency offsets can reach 10KHz (≈5 subcarriers) but have relatively smaller variances (<2 subcarriers). To suppress its impact, we manually correct each USRP's central frequency by its measured average, reducing its frequency offset to < 2 subcarriers.

Artificial Signals. Due to imperfect RF shielding, a USRP radio may leak energy to its receiving path, creating a energy peak of random strength near the central frequency (shown in Figure 6 as a spike near 2.38GHz). As a result, a radio could mistake some free subcarriers as being occupied. In our experiments this artifact leads to a small amount (<2%) of spectrum sensing errors. The impact is minor because Jello uses temporal averaged signals in its sensing, reducing the peak's edge strength to that below the detection threshold.

6 Evaluation

We evaluate Jello using both network simulations and GNU radio experiments. We use simulations to evaluate Jello with various design choices and network configurations. We also run experiments on an indoor network of 8 GNU radios in a 12m×7m room (Figure 7), running 4 simultaneous media sessions. We configure each radio's transmit power so that each link maintains 5% or less packet loss when there is no interference present, and all links interfere with each other. Each GNU radio experiment lasts 10 minutes and is repeated 5 times.

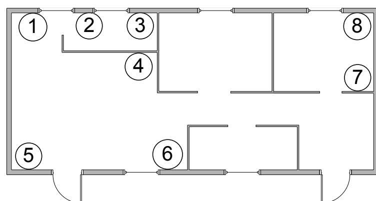


Figure 7: Our Jello testbed: 8 USRP GNU radios are placed in a 12m × 7m room with various walls and furniture.

We use both VBR video traces and synthetic On/Off traffic to generate sessions. We scale the traffic flow as necessary to create a desired load normalized by the frequency bandwidth. Sessions carrying video traffic have similar average loads, and sessions carrying On/Off traffic have different traffic volumes. For video traces, we assume a 10s application buffer so that each session's demand changes every 10s. For synthetic traffic, the On and Off periods are randomly generated from a uniform distribution. Each session determines the amount of frequency required based on its traffic demand and the average data rate achievable on each frequency subcarrier. If the current available frequency cannot fulfill the entire demand, the session will take what is available.

We evaluate Jello by comparing four systems:

- **Static**: partitioning spectrum equally by the number of sessions; each session has a dedicated frequency block.
- **Jello-C**: Jello with contiguous frequency access.
- **Jello-NC**: Jello with non-contiguous access enabled.
- **Optimal**: an "oracle" solution with perfectly accurate sensing that removes fragmentation by assigning spectrum using knowledge of all future requests.

We collect two performance metrics that measure application performance and spectrum usage efficiency:

Application disruption rate: the proportion of time that a session experiences packet losses higher than a maximum threshold X, and thus cannot sustain satisfactory media quality. For example, prior work shows that streaming video sessions can only tolerate up to 10% packet loss [26]. We examined Jello using $X = 5, 10, 20\%$, and arrived at similar conclusions. Due to space limitations, we only show results using $X = 10\%$.

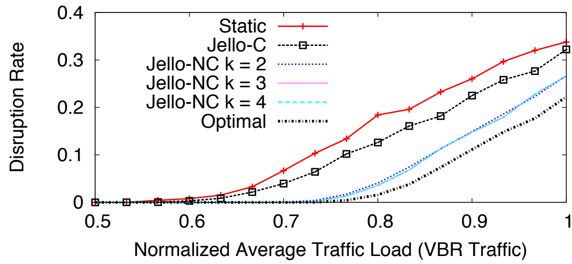
Residual usable spectrum: given a traffic load, the amount of spectrum left for a new media session, averaged over time and normalized by the spectrum capacity.

6.1 Simulation Results

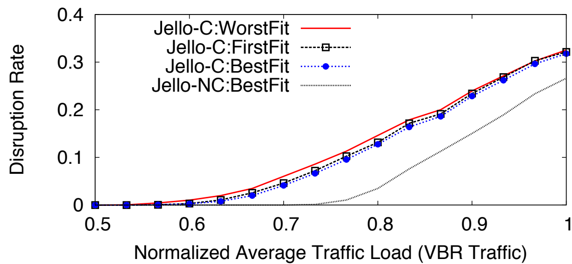
We first simulate Jello under general network configurations. We use these experiments to examine and verify Jello's design concept without any sensing/transmission error or coordination overhead. Figure 8 shows that Jello-NC, by enabling dynamic non-contiguous frequency access, significantly outperforms Jello-C and Static. There is a small distance between Jello-NC and Optimal because Jello-NC uses periodic defragmentation so that a low-level of fragmentation still remains, leading to some loss in spectrum from frequency guard bands.

We also make several key observations:

Impact of k (the maximum # of frequency blocks each radio can use). Since hardware complexity scales with k, it is interesting to understand its impact. In Figure 8



(a) Impact of max. frequency blocks allowed



(b) Impact of frequency selection algorithms

Figure 8: Simulated Jello performance using video traces: (a) when allowing each radio to access $k = 1..4$ frequency blocks; (b) when using different frequency selection algorithms.

	Normalized average traffic load				
	0.6	0.7	0.8	0.9	1
P(1 block)	98.9%	87.7%	70.3%	58.2%	52.1%
P(2 blocks)	1.1%	11.8%	25.1%	31.1%	33%
P(3 blocks)	0	0.4%	4.2%	9.1%	12.2%
P(4 blocks)	0	0.1%	0.4%	1.5%	2.4%

Table 1: Probability distribution of the number of frequency blocks each session uses, using Jello-NC with $k = \infty$.

we examine the application disruption rate of Jello-NC by varying k between 1 and 4. We see that raising k from 1 to 2 leads to a significant performance leap, but after that the benefit of raising k becomes marginal. To further examine this, we list in Table 1 the probability distribution of the number of frequency blocks each session uses when k is unlimited. We see that the need for non-contiguous access does increase with the traffic load, but each session uses no more than 3 blocks with a 97+% probability. We repeated our experiments using different traffic models and network sizes, and arrived at a similar observation. Although inconclusive, this shows that adding 1 or 2 bands to a radio's frequency access capability will significantly boost the overall performance. We also prove this trend analytically in a separate study [6].

Impact of the frequency selection algorithm. Figure 8(b) plots the application disruption rate of Jello-C with *Worst Fit*, *First Fit*, *Best Fit*, and Jello-NC with *Best Fit*. We see that Jello-C with *Best Fit* outperforms Jello-C with the rest but only slightly. In our testbed experiments, we use *Best Fit* for Jello.

Impact of network topology. We examine this impact using a network of 50 sessions. By varying the transmit power we create networks of different conflict conditions, represented by the average conflict degree D. Higher D means each session conflicts with more peers. Table 2 lists the application disruption rate for Jello-C, Jello-NC and Optimal. The same conclusion applies. One interesting observation is that Jello-NC leads to more gains as the conflict level decreases. This is because non-contiguous access provides more opportunity for spatial reuse where non-conflicting sessions can reuse the same frequency blocks.

	Average conflict degree D				
	3.9	5.1	6.3	7.5	8.5
Jello-C	0.025	0.057	0.107	0.155	0.207
Jello-NC	0.005	0.019	0.054	0.095	0.147
Optimal	0.001	0.005	0.018	0.037	0.065

Table 2: Application disruption rate with different conflict degrees using a large network of 50 sessions.

6.2 Testbed Results

We now evaluate Jello using the GNU radio testbed. All the results now include the impact of channel impairments, but those of Jello-C and Jello-NC also include the impact of coordination protocol overhead and spectrum sensing errors. For Jello-NC, we use $k = 3$ in our hardware implementation.

6.2.1 Jello's Overall Performance

Media Quality Measurements. Figure 9 summarizes the application disruption rates using both video and synthetic traffic. Due to channel impairments, all disruption rates are slightly higher than those of simulations. We see that Jello-NC can effectively utilize a large portion of the spectrum (up to 75%) while keeping disruption rates below 5%. It outperforms Static and Jello-C significantly and is within a reasonable distance from Optimal.

Jello-C also outperforms Static, except in the VBR case when the traffic load is lower than 68%. This unexpected degradation comes from Jello's coordination overhead, hardware artifacts (discussed in Section 5.3) and sensing errors (recall that Static has no such overhead). As the traffic load grows, the gain of dynamic spectrum multiplexing overcomes the system overhead. For On/Off traffic, Jello-C consistently outperforms Static. This is because traffic burstiness is higher than that of the VBR traffic, thus dynamic spectrum access leads to significant gains.

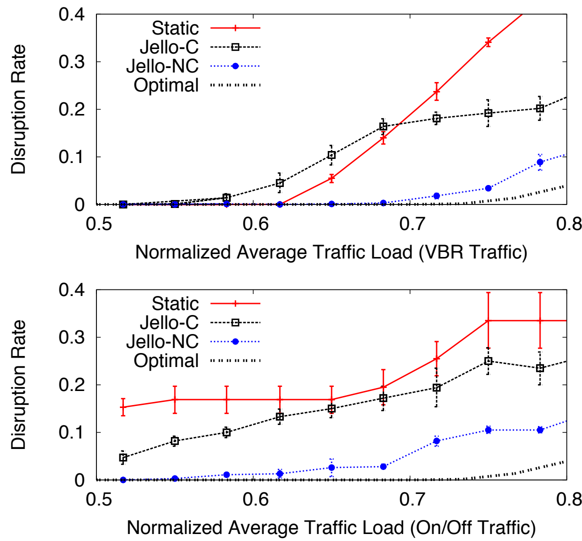


Figure 9: Testbed results: application disruption rate vs. average traffic load. Jello-NC consistently outperforms Jello-C and Static, and is within a small gap from Optimal.

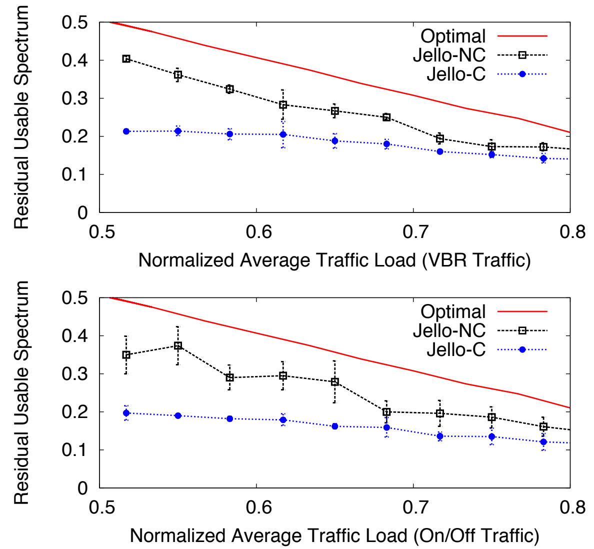


Figure 10: Testbed results: comparing Jello-C, Jello-NC and Optimal in terms of the residual usable spectrum.

Spectrum Usage Efficiency. As another measure of Jello's spectrum usage efficiency, we measure the *residual usable spectrum* as a function of the normalized average traffic load. Figure 10 shows the results for Jello-C, Jello-NC and Optimal. The result of Statis is not shown because the entire spectrum is used by existing sessions.

Compared to Optimal which completely removes all fragments, Jello-NC only sacrifices 10-15% of the total spectrum bandwidth. Among those, 3% comes from the extra guard bands associated with the non-contiguous frequency access (due to infrequent defragmentation), and the rest is from sensing errors and the fact that each new flow can only at most 3 frequency blocks.

For Jello-C, however, the overhead increases to 20-30% of the total spectrum bandwidth. In this case, the impact of residual fragmentations is amplified by the limitation that each new flow can only use 1 frequency block. For the same reason, its residual spectrum is insensitive to variations in traffic loads. An alternative way to interpret the results is that, compared to Jello-C, Jello-NC offers up to 45% more free spectrum to new sessions.

6.2.2 Where Does The Gain Come From?

The improvement of Jello comes from both non-contiguous spectrum access and online defragmentation. In the following, we evaluate their gains separately by comparing the performance of Jello with contiguous and non-contiguous frequency access, and by enabling and disabling online fragmentation.

Benefits from Non-contiguous Frequency Access. For a fair comparison, we assume both access mechanisms use online defragmentation. Figure 9 already shows that allowing non-contiguous access keeps the disruption rate below 10%, while contiguous access may suffer more than 25% disruptions. Another way to interpret the result is that, to keep a 10% or less disruption, non-contiguous access achieves 22–32% improvement in spectrum utilization over contiguous access.

Benefits of Online Defragmentation. Using On/Off traffic, we compare the performance of Jello with and without online spectrum defragmentation. From Figure 11, we see that online defragmentation reduces spectrum disruptions for both contiguous and non-contiguous Jello. For example, with 68% load, defragmentation reduces disruptions from 18% to 15% for contiguous and 5% to 3% for non-contiguous access. Compared to enabling non-contiguous access, online defragmentation has a smaller gain. This is because in our implementation, Jello devices defragment infrequently (at most twice per On period) due to hardware limitations.

6.2.3 Jello's Overhead

Having examined Jello's application-level and spectrum usage performance, we now look into the overhead that separates Jello-NC from Optimal. We quantify the impact of each element that contributes to Jello-NC's application disruption rate. These include: (1) the inherent traffic dynamics where the total spectrum cannot support all the sessions (the same applies to Optimal); (2) the frequency guard band overhead from non-contiguous

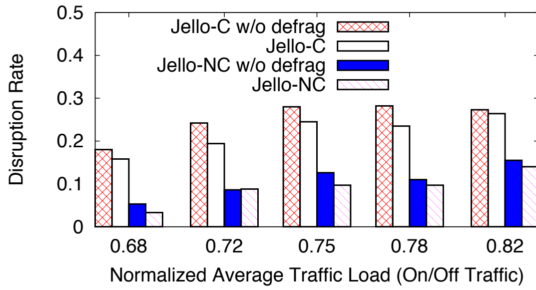


Figure 11: Testbed results: benefits of Jello's online defragmentation using On/Off traffic.

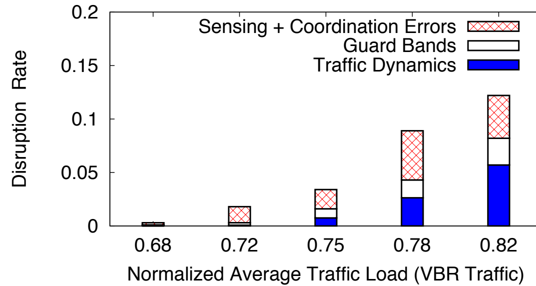


Figure 12: Testbed results: breakdown of contributions in Jello-NC's disruptions, using video traces. The impact of traffic dynamics is unavoidable and also applies to Optimal.

frequency access (for being unable to defragment spectrum completely); (3) the sensing error and coordination overhead caused by channel impairments, and (4) conflicting defragmentation. Results in Figure 12 show that the guard band overhead has a relatively small impact compared to the other two, which confirms that Jello produces a very low-level of spectrum fragmentation. The probability of defragmentation conflicts is 0.5% in our experiments and its impact is absorbed in the coordination errors in Figure 12.

Frequency Guard Bands. In Figure 13(a), we compare Jello-NC and Optimal in terms of their guard band overhead. Without any fragment, Optimal uses a fixed number of guard bands (6 subcarriers out of 240 usable subcarriers), or a 2.5% of overhead. For Jello-NC, the guard band overhead increases with the traffic load because each session uses more frequency blocks to fulfill its demand. However, similar to the results in Table 1, in our experiments more than 85% of time a session uses only 1 or 2 frequency blocks. Thus the overall guard band overhead is less than 5% even at 80% traffic load.

Spectrum Sensing Errors. Figure 12 shows that sensing errors could be a major contributor to the disruptions. In our current implementation, the average false positive (treating available blocks as occupied) and false negative (treating occupied blocks as available) rates are 5–10%. Figure 13(b) shows the results of two sample topologies. These errors are due to the time-varying channel impairments and heterogeneous signal strengths commonly found in indoor environments.

On the other hand, Jello's edge-detection based sensing is much more accurate than the energy-detector, and is relatively insensitive to the choice of detection threshold. To quantify this benefit, we plot in Figure 14 the detection false positive and false negative rates from energy-detection based sensing, using the same topologies in Figure 13(b). We see that energy-detection sensing leads to much higher detection errors, and is highly sensitive to the choice of its detection threshold (-32dB for topology 1, -48dB for topology 2). In addition, it suffers from high false positives (e.g. 40%) in order to maintain a reasonable rate of false negatives (e.g. 10%).

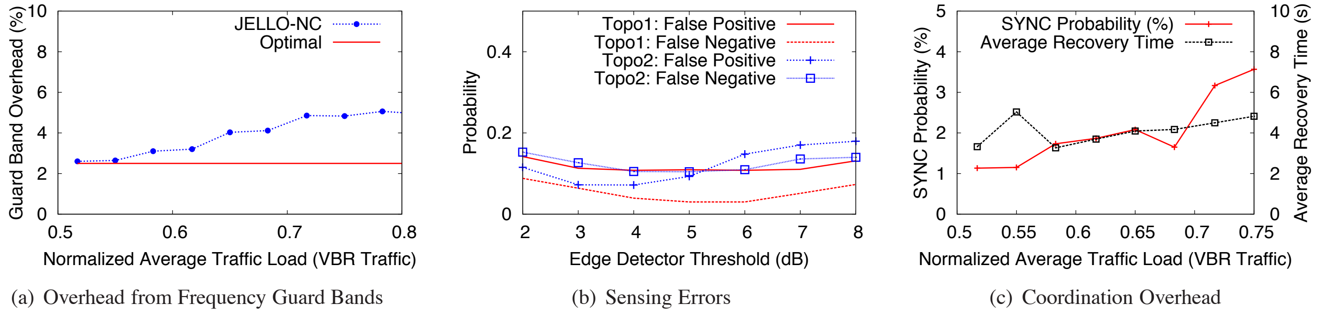
Coordination Overhead. The majority of Jello's coordination overhead is due to links falling back to the SYNC state to resynchronize. In our experiments, these occur from external interference, or an unlikely conflict in frequency adjustments. From Figure 13(c), we see that the probability of entering SYNC is only 2-3%, and the average recovery time is 4-5s. Both the SYNC probability and the recovery time increase with the traffic load because as more sessions start to adapt frequency for additional spectrum, they create slightly more conflicts and more traffic on the SCS. Now a session could wait longer before starting resynchronization. However, because links leave the SCS immediately after locating free spectrum, the SCS utilization stays low.

7 Discussion

We can extend Jello in the following directions.

Integrating with Other MAC Functions. Due to USRP Radios' large processing delay, current Jello implementation does not include several MAC functions. These include (1) rate adaptation (Jello uses BPSK); (2) channel-aware frequency selection (in our experiments the channel quality is flat across the frequency range due to limited bandwidth); (3) power control (we use uniform transmit power across all the subcarriers in use); and (4) packet retransmission. Using powerful radio platforms, Jello can add these functions. A key issue is to investigate the interaction between Jello's frequency selection and these functions and to jointly optimize them together.

Optimizing Frequency Selection. Jello's frequency selection algorithms focus on minimizing network-wide spectrum fragmentation and conflicts. Additional information about each spectrum section such as received signal strength can allow Jello to choose a good set frequency blocks to achieve reliable transmissions match-



(a) Overhead from Frequency Guard Bands (b) Sensing Errors (c) Coordination Overhead

Figure 13: Testbed results: Examining Jello-NC's overhead in terms of the frequency guard band overhead, sensing errors, and coordination delay.

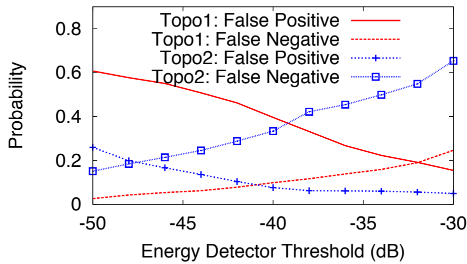


Figure 14: Testbed results: detection reliability of energy detection-based sensing as a function of its detection threshold. Compared to Jello's edge-detector, it leads to much higher detection errors, and is highly sensitive to the choice of its detection threshold (-32dB for topology 1, -48dB for topology 2).

ing its traffic demand and minimize frequency usage. Jello can also use this information to configure a proper amount of guard bands at link boundaries instead of using a uniform configuration. An interesting issue is how to obtain such information reliably and efficiently.

Porting Jello to Other Radios. Jello can be ported onto advanced hardware platforms [16, 21, 24, 28, 30], to benefit from their increased frequency bandwidth and processing speed. For best performance, Jello requires radios that can support fine-grained frequency access, and quickly scan spectrum to identify available ranges.

8 Related Work

We divide the related work into two categories: contiguous and non-contiguous frequency access.

Contiguous Frequency Access. The majority work on dynamic spectrum networks assumes contiguous frequency access [1, 4, 8, 16, 20, 31, 32]. This access pattern has the advantage of being readily implemented on conventional 802.11 devices [8]. In this context, prior works have developed centralized algorithms for load balancing [20], distributed protocols for spectrum contention [32] and for utilizing UHF whitespaces [4].

Jello differs from these works in three aspects. First, Jello's per-session FDMA design is more general in that it operates across wider spectrum ranges at a fine granularity and completely eliminates CSMA traffic contention. Second, unlike [32], which requires a separate control radio to reserve spectrum, Jello devices self-sense spectrum to avoid access conflicts, and defragment spectrum while staying transparent to others. As a result, Jello provides dedicated frequency usage to demanding applications. Finally, Jello's spectrum sensing differs from SIFT [4] which detects any contiguous frequency usage using time-domain signals. Instead, Jello uses wide-band sensing in the frequency domain that can quickly identify multiple active frequency blocks instead of single blocks at a time.

Non-contiguous Frequency Access. Most works in this area assume either centralized control [23] or a dedicated radio for control. Others are limited to simulations [7] without considering practical artifacts such as sensing and guard bands. Jello, on the other hand, implements distributed non-contiguous frequency access and deploys a USRP prototype.

SWIFT [24] is a distributed wideband spectrum access system that can use a large frequency band even when a narrowband signal is present. SWIFT nodes share spectrum in the time domain using CSMA. Jello differs from SWIFT by using per-session FDMA to avoid costly packet contentions and by using an non-intrusive edge-detection based mechanism to identify usable frequency. ODS [15] implements on-demand spectrum access using spread-spectrum codes, focusing on adapting spectrum allocation to bursty traffic. It applies a random policy for selecting codes and uses adaptive receiver feedback to regulate code allocations. Jello differs from ODS by operating in the frequency-domain, using spectrum sensing to avoid access conflicts.

9 Conclusion

Jello provides a new distributed spectrum access technique for demanding wireless applications. High-quality delay-sensitive media sessions can now access and share wireless medium in the frequency domain and adapt their spectrum usage to varying traffic demands. Jello utilizes frequency-agile radios to sense, identify and occupy unused spectrum, allowing multiple sessions to work in parallel on isolated frequencies. To maximize spectrum usage efficiency, Jello devices self-defragment spectrum on-the-fly, and scavenge multiple frequency fragments for use by single, high-speed transmissions. Jello is also MAC-agnostic and does not require any dedicated radio for control. Despite USRP radio's limited bandwidth and large processing delays, our measurements on an 8-node testbed confirm that Jello can provide reliable spectrum access for media applications and significantly improve spectrum usage efficiency.

Acknowledgments

We thank Geoff Voelker, our shepherd Venkat Padmanabhan, and the anonymous reviewers for their helpful suggestions. We also thank Peter Steenkiste and Songwu Lu for their comments on earlier versions of this work. This work is supported in part by NSF Grants CNS-0916307, IIS-0847925, CNS-0832090, CNS-0546216. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] AKYILDIZ, I. F., LEE, W. Y., VURAN, M., AND MOHANTY, S. NeXt generation/dynamic spectrum access/cognitive radio wireless networks: A survey. *Computer Networks Journal (Elsevier)* (2006).

[2] AMINI, P., ET AL. Implementation of a cognitive radio modem. In *Proc. of SDR* (2007).

[3] MPEG-4 and H.263 video traces for network performance evaluation. http://trace.eas.asu.edu/TRACE/trace.html/.

[4] BAHL, P., CHANDRA, R., MOSCIBRODA, T., MURTY, R., AND WELSH, M. White space networking with Wi-Fi like connectivity. In *Proc. of SIGCOMM* (2009).

[5] CANNY, J. A computational approach to edge detection. *IEEE Trans. on Pattern Analysis and Machine Intelligence 8*, 6 (1986), 679–698.

[6] CAO, L., YANG, L., AND ZHENG, H. The impact of frequency-agility on dynamic spectrum sharing. In *Proc. of IEEE DySPAN* (2010).

[7] CAO, L., AND ZHENG, H. Spectrum allocation in ad hoc networks via local bargaining. In *Proc. of SECON* (2005).

[8] CHANDRA, R., MAHAJAN, R., MOSCIBRODA, T., RAGHAVENDRA, R., AND BAHL, P. A case for adapting channel width in wireless networks. In *Proc. of SIGCOMM* (2008).

[9] CRAMTON, P., SKRZYPACZ, A., AND WILSON, R. The 700 MHz spectrum auction: An opportunity to protect competition in a consolidating industry. *Report for Frontline Wireless* (2007).

[10] DIGHAM, F. F., ALOUINI, M.-S., AND SIMON, M. K. On the energy detection of unknown signals over fading channels. *IEEE Transactions on Communications* (2007).

[11] FEHSKE, A., GAEDDERT, J., AND REED, J. A new approach to signal classification using spectral correlation and neural networks. In *Proc. of IEEE DySPAN* (2005).

[12] GE, F., YOUNG, A., BRISEBOIS, T., CHEN, Q., AND BOSTIAN, C. W. Software defined radio execution latency. In *Proc. of SDR* (2008).

[13] GOLDSMITH, A. *Wireless Communications*. Cambridge University Press, New York, NY, USA, 2005.

[14] GONZALEZ, R. C., AND WOODS, R. E. *Digital Image Processing (3rd Edition)*. Prentice-Hall, 2006.

[15] GUMMADI, R., AND BALAKRISHNAN, H. Wireless networks should spread spectrum based on demands. In *HotNets* (2008).

[16] GUMMADI, R., NG, M. C., FLEMING, K., AND BALAKRISHNAN, H. AirBlue: A system for cross-layer wireless protocol development and experimentation. In *MIT Report* (2008).

[17] HOU, W., YANG, L., ZHANG, L., SHAN, X., AND ZHENG, H. Understanding the impact of cross-band interference. In *Proc. of ACM Coronet Workshop* (2009).

[18] JARDOSH, A. P., RAMACHANDRAN, K. N., ALMEROTH, K. C., AND BELDING-ROYER, E. M. Understanding congestion in IEEE 802.11b wireless networks. In *Proc. of IMC* (2005).

[19] KNUTH, D. E. *The Art of Computer Programming, Vol. 1 (3rd ed.): Fundamental Algorithms*. Addison-Wesley, 1973.

[20] MOSCIBRODA, T., CHANDRA, R., WU, Y., SENGUPTA, S., BAHL, P., AND YUAN, Y. Load-aware spectrum distribution in wireless LANs. In *Proc. of ICNP* (2008).

[21] NYCHIS, G., SESHAN, S., STEENKISTE, P., HOTTELIER, T., AND YANG, Z. Enabling MAC protocol implementations on software-defined radios. In *Proc. of NSDI* (2009).

[22] OPPENHEIM, A. V., SCHAFER, R. W., AND BUCK, J. R. *Discrete-time signal processing (2nd ed.)*. Prentice-Hall, 1999.

[23] RAHUL, H., EDALAT, F., KATABI, D., AND SODINI, C. Frequency-aware rate adaptation and MAC protocols. In *Proc. of MobiCom* (2009).

[24] RAHUL, H., KUSHMAN, N., KATABI, D., SODINI, C., AND EDALAT, F. Learning to share: narrowband-friendly wideband networks. In *Proc. of SIGCOMM* (2008).

[25] SINGH, S., ACHARYA, P. A. K., MADHOW, U., AND BELDING-ROYER, E. M. Sticky CSMA/CA: Implicit synchronization and real-time QoS in mesh networks. *Ad Hoc Network 5*, 6 (2007), 744–768.

[26] STOCKHAMMER, T., HANNUKSELA, M. M., AND WIEGAND, T. H.264/AVC in wireless environments. *IEEE Trans. on Circuits and Systems for Video Technology 13*, 7 (2003), 657–673.

[27] SUN, Y., SHERIFF, I., BELDING-ROYER, E. M., AND ALMEROTH, K. C. An experimental study of multimedia traffic performance in mesh networks. In *Proc. of WiTMeMo* (2005).

[28] TAN, K., ET AL. SORA: High performance software radio using general purpose multi-core processors. In *Proc. of NSDI* (2009).

[29] TIMOTHY M., S., AND DONALD C., C. Robust frequency and timing synchronization for OFDM. In *IEEE Transactions on Communications* (1997).

[30] Wireless open-access research platform. http://warp.rice.edu/.

[31] YUAN, Y., BAHL, P., CHANDRA, R., MOSCIBRODA, T., NARLANKA, S., AND WU, Y. Allocating dynamic time-spectrum blocks in cognitive radio networks. In *Proc. of MobiHoc* (2007).

[32] YUAN, Y., ET AL. KNOWS: Kognitiv networking over white spaces. In *Proc. of IEEE DySPAN* (2007).

Contracts: Practical Contribution Incentives for P2P Live Streaming

Michael Piatek* Arvind Krishnamurthy* Arun Venkataramani†
Richard Yang◇ David Zhang× Alexander Jaffe*

Abstract

PPLive is a popular P2P video system used daily by millions of people worldwide. Achieving this level of scalability depends on users making contributions to the system, but currently, these contributions are neither verified nor rewarded. In this paper, we describe the design and implementation of *Contracts*, a new, practical approach to providing contribution incentives in P2P live streaming systems. Using measurements of tens of thousands of PPLive users, we show that widely-used bilateral incentive strategies cannot be effectively applied to the live streaming environment. *Contracts* adopts a different approach: rewarding globally effective contribution with improved robustness. Using a modified PPLive client, we show that *Contracts* both improves performance and strengthens contribution incentives. For example, in our experiments, the fraction of PPLive clients using *Contracts* experiencing loss-free playback is more than 4 times that of native PPLive.

1 Introduction

System collapse due to large-scale reductions in user contributions is a major concern for PPLive, which is one of the most widely deployed live streaming services on the Internet today, serving more than 20 million active users spread across the globe. Using peer-to-peer (P2P) as the core technique, PPLive achieves cost-effective live video distribution by providing a small amount of seed bandwidth to a few participants, with the rest of the distribution being performed by users relaying data. Thus, the availability and scalability of PPLive depends crucially on the contributions of its users.

The current PPLive design neither verifies nor rewards contributions, creating the potential for strategic users to restrict their contribution, degrading robustness. This is particularly true in environments where capacity is limited or priced by usage. Furthermore, when developing an open live video streaming standard, relying on closed systems with proprietary protocols is not feasible.

In this paper, we explore how to provide practical contribution incentives for P2P live streaming, using PPLive as a concrete example. Although incentives have been studied extensively in the case of widely deployed file-sharing systems (e.g., [1, 16, 22]), live streaming presents unique challenges. For instance, clients cannot be rewarded with faster downloads once they are receiving data at the broadcast rate (since additional data has not yet been produced). While some recent proposals have considered contribution incentives in a live streaming setting (e.g., [17]), they do not take into account several practical considerations of deployed systems, such as client heterogeneity and operation under bandwidth constraints. We provide an examination of live streaming incentives grounded in experience with a deployed and widely used live streaming system.

We proceed in two steps. First, we use measurements of tens of thousands of PPLive clients to demonstrate quantitatively the challenges in adapting existing incentive strategies to the live streaming environment. We find that in practice, the majority of system capacity is contributed by a minority of high capacity users. As a result, incentive mechanisms that require balance between consumption and contribution will either exclude many users from participation or underutilize capacity substantially.

More broadly, bilateral exchange mechanisms widely used in bulk data distribution, such as tit-for-tat in BitTorrent [5], are ineffective in the live streaming environment, in part because sequential block availability sharply limits trading opportunities between peers. Imposing topologies that increase bilateral trading opportunities (e.g., [17]) increases the variance in block delivery delay, causing either increased playback deadline misses or increased startup delay. Tit-for-tat, for example, substantially reduces performance when applied to PPLive.

The second part of the paper describes *Contracts*, a new design for providing robust contribution incentives in live streaming P2P services. *Contracts* differs from existing techniques in two principal ways. First, *Contracts* is designed for the live streaming environment. Rather than relying on increased download rates, *Contracts* rewards contributions with increased quality of service when the system is constrained. Second, *Contracts* departs from traditional incentive mechanisms that rigidly constrain client behavior. Instead, we define a default *contract* specifying an agreement between an individual client and the overall system (i.e., PPLive and other clients) as to how its contributions will be evaluated by others. To enable this, we introduce a lightweight protocol that provides verifiable accounting of each client's contributions. But, the contract does not mandate fine-grained behavior, leaving individual clients free to make local optimizations that increase efficiency.

We have integrated *Contracts* with PPLive, and find that our implementation both improves performance and strengthens contribution incentives. For example, in our

*U. of Washington; † U. of Mass; ◇ Yale; × PPLive.

experiments, the fraction of PPLive/*Contracts* clients experiencing loss-free playback is more than 4 times that of native PPLive, and clients that contribute more than others receive consistently higher quality of service.

The remainder of this paper is organized as follows. Section 2 provides an overview of live streaming in PPLive. Sections 3 and 4 describe the challenges of applying incentive strategies based on bilateral exchange to live streaming. These challenges motivate the design and implementation of *Contracts*, which we present in Section 5 and evaluate in Section 6. We discuss related work in Section 7 and conclude in Section 8.

2 PPlive overview

PPLive is a hybrid P2P system for streaming live and on-demand video. Clients are organized into channels, with members of a given channel redistributing video data to one another. Clients rely on two forms of infrastructural support: 1) coordinating *trackers* that provide a rendezvous point for users watching the same channel, and 2) seed bandwidth provided by a group of broadcast servers that source all content. Multiple channels can be managed by a single tracker and sourced by a single broadcaster. Currently, PPLive maintains roughly 600 public live channels daily.

The wire-level details of the PPLive protocol are similar to existing swarming systems like BitTorrent [5]. Clients maintain a large set of directly connected peers (50–100) to which they advertise their local data blocks and issue requests for missing blocks. Each block is 4–16 KB and is discarded shortly after being played.

Trackers maintain state that includes the set of clients in each channel, the properties of clients (e.g., reported bandwidth capacity and NAT status), and the overall health of channels. The health of an individual channel is monitored by its broadcast source. Clients use the source as a peer of last resort. Only when a block cannot be obtained from any other peer is a request sent to the data source. Thus, the load on the broadcaster provides a metric for the health of a given channel relative to others. By shifting capacity from channels demanding less load to those servicing more requests, PPLive allocates infrastructure bandwidth automatically.

Most relevant to our work is PPLive's servicing policy. By default, each client contributes its full available capacity and does not prioritize service for particular peers, i.e., download requests from a peer contributing little to the system and those from a peer making the highest contribution are treated equally.

3 Limits of bilateral exchange

The lack of contribution incentives means that PPLive's scalability depends on clients' good will and the faithful execution of its software. At present, these are largely effective due to flat-rate network pricing and the complexity of PPLive's proprietary implementation. Increasingly, however, the all-you-can-eat pricing model is giving way to the realities of network management [6]. Furthermore, there is continued interest in developing an open live video streaming standard supporting multiple implementations (e.g., IETF 73 PPSP BoF). These trends motivate the explicit consideration of incentives for live streaming systems to reward good (and discourage bad) behavior by coupling performance and contribution.

Why does live streaming necessitate revisiting the incentive design problem? To appreciate this, consider the most widely-used class of incentive strategies in P2P systems today—bilateral exchange—wherein a peer x determines the amount of upload bandwidth to peer y based solely on the amount of bandwidth that client y uploads to x, independent of the total bandwidth that client y uploads to all clients. Servicing policies based on bilateral exchange are compellingly simple. For example, tit-for-tat has been widely applied in bulk data distribution systems (e.g., BitTorrent [5]), and has also been studied extensively [16, 25]. More recently, bilateral exchange has also been proposed as a basis for providing incentives in live streaming systems (e.g., [17]). However, bilateral exchange schemes suffer from fundamental performance limitations in the context of live streaming.

For bilateral exchange to work, peers need to have trading opportunities (i.e., distinct data blocks of mutual interest). When distributing bulk data, trading opportunities are frequent. Each client seeks to acquire the entirety of a large set of blocks. Bulk distribution systems typically use block selection strategies such as local rarest first (e.g., BitTorrent [5]) or network coding of blocks (e.g., Avalanche [9]) to ensure that all blocks having roughly equal trading value over time. Ideally, once a new client has received just a few random blocks, it is bootstrapped into the trading system.

Live streaming differs radically from bulk data distribution in ways that significantly reduce the effectiveness of bilateral exchange. We consider four key challenges in live streaming that inform the design of *Contracts*.

1) Heterogeneity: Capacity heterogeneity poses a fundamental challenge to the efficiency of balanced exchange schemes. Live streaming offers a common download rate—the stream playback rate—to all peers regardless of their upload capacity. In practice however, peer capacities can vary by an order of magnitude. We verify this by measuring the capacity distribution of PPLive clients using logs of the reported bandwidth capacity of 99,184 clients. The distribution is highly skewed with a mean capacity (142 KBps) that is more than double the median (65 KBps). As a result of the skew, the majority of total aggregate capacity is provided by a minority of high capacity peers. The top 10% of clients account for

58% of total capacity.

Capacity heterogeneity implies a discouraging tradeoff between efficiency and balance. Insisting on near-perfect balance will either exclude many users that cannot support the stream rate or significantly underutilize capacity. Concretely, streaming at the average capacity of 142 KBps (the maximum possible) would would exclude 86% of PPLive clients in our trace when requiring balanced contribution and consumption. On the other hand, providing service to 95% of PPLive clients (with balanced exchanges) requires restricting the stream data rate to the 5th percentile of capacity at 21 KBps, which corresponds to an overall utilization of just 15%.

The fundamental tradeoff between efficiency and balance under skew can be quantified as follows. Let μ and σ respectively denote the mean and variance of the upload capacity distribution. We define the *skew*[1] S as σ/μ. For a stream rate of r, the *efficiency* E is r/μ, where a feasible broadcast implies $\mu \geq r$. We define the *imbalance* I as the deviation of peer upload rates with respect to the stream rate normalized by the mean, i.e., $\frac{1}{\mu}[\frac{\sum_i (x_i - r)^2}{N}]^{1/2}$, where x_i is peer i's upload rate (less than or equal to its capacity) and the sum indexes over all N peers. Note that all three of skew, efficiency, and balance lie between 0 and 1. The theorem below captures the stated tradeoff, the proof of which is available in a technical report [24].

THEOREM **1** *High efficiency and high skew imply high imbalance. Specifically,* **(a)** *If peers upload at a rate proportional to their capacity, $I = E \cdot S$.* **(b)** *For any feasible set of upload rates, I is bounded from below by a function that monotonically increases from* 0 *to S as E increases from* 0 *to* 1.

2) Limited bandwidth needs: In bulk data distribution using bilateral exchange, the incentive to increase upload rate is a corresponding increase in download rate. In live streaming, however, once a client is downloading data at the rate of production, a further increase in download rate is not possible (as additional data does not yet exist). Although one may consider rewarding increased contribution with improved video quality (e.g., at higher resolutions using layered coding), PPLive avoids such a scheme due to its increased complexity, reduced video coding efficiency, and the need for substantially higher bandwidth to produce visually-discernible quality differences. Thus, a challenge to incentivize users to contribute capacity in excess of their demands is to create a compelling reward with nonzero marginal utility.

The above points do not rule out bilateral exchange schemes that are not balanced, which we consider below.

3) Limited trading opportunities: Bilateral exchange depends on the existence of mutually beneficial trading opportunities to evaluate peers. Unfortunately, live streaming provides clients with limited opportunities for mutually beneficial trading. The key difference is that unlike bulk data distribution, where blocks have roughly equal value over time and among clients, *the value of blocks in live streaming varies over time and client*. A block has little value at a client if it is received after the playback point at the client. Thus, the data useful to an individual client is limited to a narrow range between the production point and the local playback point, i.e., the *lag*. The smaller the lag that a system targets, the fewer the trading opportunities. Furthermore, blocks in live streaming emerge at the data source one at a time at the production rate unlike bulk data distribution where data becomes available all at once. As a result, clients closer to the data source in the topology have inherent advantages in receiving rare (new) blocks first, creating a perpetual trade imbalance with clients further from the source. Although trade imbalance does not necessarily rule out bilateral exchange schemes, it makes evaluating peers significantly more challenging.

[1] unlike the more standard definition based on the third moment.

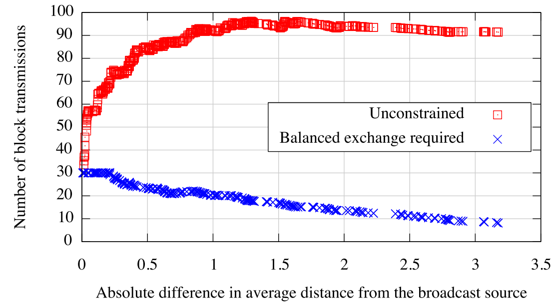


Figure 1: The impact of distance from the broadcast source on bilateral exchange. Requiring balanced exchange significantly limits trading opportunities as does distance from the source.

To make this concrete, we compare the number of trading opportunities for clients in a PPLive broadcast with 100 clients running on the Emulab testbed. Each client uses random block selection to maximize trading opportunities unless a block is near its playback deadline. Each client simultaneously joins a test stream and periodically logs its buffer state during playback. Figure 1 summarizes the trading opportunities among pairs of peers taken from a snapshot of buffer states collected several minutes into the broadcast. Each individual client's average distance to the broadcast source is the average number of overlay hops traversed by all of its received blocks, which correlates with lag. The number of trading opportunities is shown in terms of the absolute difference in these averages for pairs of peers (x-axis) without constraints and when a balanced number of mutually beneficial trades is required. These results show that the greatest opportunity for bilateral exchange is between peers that are at a similar distance from the broadcast source. But, such pairs of peers are in the minority. Most

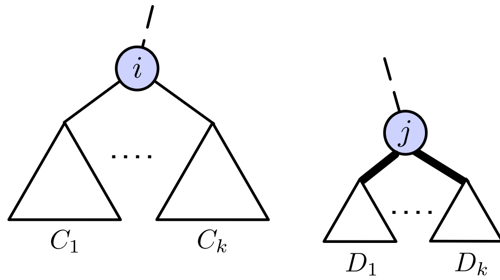


Figure 2: Illustration for Theorem 2: Node j has higher upload capacity than node i but has fewer descendants.

block transfer opportunities are between pairs of peers that have an imbalance of useful data.

4) Delay sensitivity: Live streaming requires low distribution delay so as to reduce lag (i.e., improve liveness) or, equivalently, reduce the playback miss rate for a given lag. Minimizing block dissemination delays imposes some structural requirements on the steady-state block distribution topology. We formalize this claim as follows. The dissemination topology traversed by a single block must be a tree as peers only request blocks they do not already have. Consider the dissemination tree T for a single block. Note that different blocks may have different dissemination trees, so a node may be at different distances from the source across blocks. We assume that in steady-state, the system can sustain the stream rate such that a block is never queued at a node behind another block.[2]

THEOREM **2** *Any topology in which a peer i has lower bandwidth than peer j but i has more descendants than j has higher average block delay than the topology obtained by swapping i and j if one of the following two conditions hold: (a) the topology is a balanced tree, or (b) i is an ancestor of j.*

Figure 2 illustrates the condition in the theorem above. The proof shows that, if either T is balanced or i is an ancestor of j, T can be transformed to a topology T' with lower delay by simply swapping i and j.

The structural requirement for low delays presents a design conflict for bilateral exchange schemes. Being closer to the source, a high capacity peer A is likely to receive newer blocks before a lower capacity child B, so A is unlikely to benefit from B. However, in bilateral exchange, A evaluates B solely by B's uploads to A, forcing B to try to upload to A even though that is detrimental to the average block delay. Note that bulk distribution does not face this predicament as it does not require individual block delays to be low, a crucial consideration in live streaming.

[2]The technical report [24] describes a procedure to construct such a dissemination tree packing.

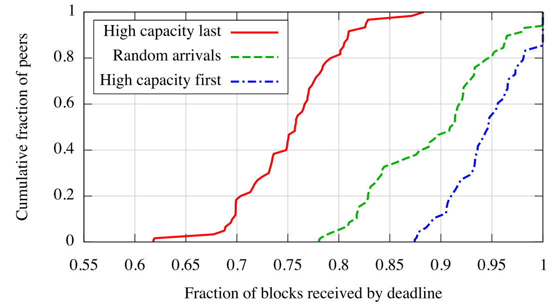


Figure 3: Cumulative fraction of clients with a given block delivery rate for different topologies. Placing high capacity clients near the source improves quality.

4 Structuring for performance and incentives

The limitations of bilateral exchange lead us to pursue a fundamentally different approach to providing incentives in P2P live streaming systems. Instead of rewarding higher upload rates with higher download rates, we craft a mechanism that incentivizes higher upload rates with robust playback quality; i.e., fewer missed playback deadlines, despite churn and capacity constraints. As observed in measurement studies [20], in a streaming system at a given channel rate, robust playback quality is the key determiner of user satisfaction. Our design of the incentive mechanism is enabled by a pleasant coincidence that aligns performance and incentive objectives: high capacity peers must be close to the source to keep block delays low, and peers closer to the source experience lower and more predictable block delays yielding better playback quality.

Topology: To maximize utilization, high capacity clients need to be placed near the data source so that they can quickly replicate useful data. To demonstrate the impact of topology and capacity heterogeneity on playback quality, we compare the block delivery rate for 120 instrumented PPLive clients running on Emulab under three scenarios: 1) clients joining in a random order, 2) high capacity clients joining first, followed by low capacity clients, and 3) low capacity clients preceding high capacity. In each scenario, the over-provisioning of capacity relative to demand is two. 50% of clients are assigned an upload capacity equal to the stream data rate (low capacity) with the remaining 50% having capacity 3× the stream rate (high capacity). The results are summarized in Figure 3. Playback quality is best when high capacity peers join first, and are therefore closer to the data source. When high capacity peers join last, the quality degrades significantly. With no change in total system capacity, the median delivery rate drops from 0.95 to 0.75. In practice, the order in which clients join is likely to be random with respect to capacity, yielding playback quality in between the two extremes.

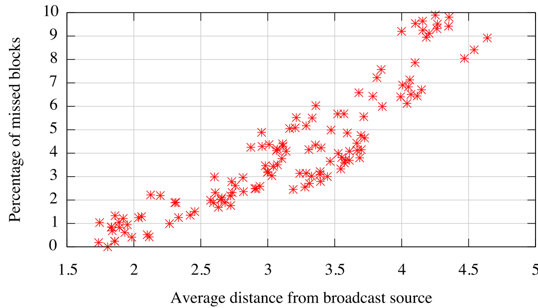


Figure 4: The fraction of blocks missing playback deadlines as a function of distance from the broadcast source. Playback quality is best for clients nearest to the source.

Buffering: When blocks miss playback deadlines, PPLive takes one of two actions: 1) if only a few blocks are missing, they are skipped; 2) but, if several blocks miss their playback deadline, PPLive pauses while waiting for downloads to complete. This buffering policy is designed to handle temporary degradation in quality of service. A single missed block has limited impact on video quality, and rebuffering suffices to recover from more significant fluctuations. But, if a client chronically experiences misses, it will eventually fall so far behind its directly connected peers that required blocks are no longer available. In this case, users need to manually rejoin the broadcast. Restarting is a simple recovery mechanism and requiring it is an explicit design choice in PPLive that is consistent with typical user behavior.

The buffering policy implies that the effects of quality degradation cascade. When a client near the data source stalls, more distant clients to which it forwards data also experience service disruption. Although the PPLive mesh contains significant path redundancy, failover is not instantaneous and may require rebuffering. We quantify the quality of service in terms of distance from the broadcast source in Figure 4. In this experiment, 127 clients with equal capacities (twice the stream rate) participate in a broadcast with one client joining every 10 seconds. Statistics are computed after all clients have been in the system for at least 20 minutes. As in previous experiments, we define the average distance of a client from the source to be the average number of hops traversed by all blocks received by the client. The results show that service quality degrades with distance from the source even in a static setting. Introducing churn will further degrade service quality for clients that are further away from the source.

5 Contracts

We now describe *Contracts*, a new scheme to provide contribution incentives in P2P live streaming systems. Our scheme is based on two key design choices that are motivated by the considerations unique to live streaming.

Contracts rather than bilateral reciprocation: Recognizing that bilateral exchanges are ineffective for live streaming, we develop a scheme that rewards each peer according to its global effectiveness. We borrow from economic theory, in particular the *principal-agent problem*, the idea of *contracts*—a method of structuring incentives in asymmetric or non-bilateral settings [15]. In *Contracts*, a data provider grants a level of service proportional to the consumer's ability to replicate the data further, as opposed to basing service simply on reciprocal contributions. The contract is thus designed to motivate consumers to contribute their bandwidth and also to hold them accountable for their respective servicing decisions. Further, since a provider is also a consumer in a P2P setting, it should be in the provider's self-interest to enforce the contract to obtain good service from its own providers. Put simply, a node's incentives as a provider should be aligned with its incentives as a consumer.

Global topology optimization: Instead of operating with an unstructured mesh, *Contracts* structures the overlay topology globally to account for the heterogeneity of peer capacities. Specifically, we introduce mechanisms that allow clients to identify their positions relative to the stream source and reorganize themselves, with high capacity peers percolating towards the source. Peers with disparate capacities develop asymmetric yet mutually beneficial relationships: low capacity peers benefit from the replication capabilities of high capacity peers, while high capacity peers are rewarded with better quality of service for their contributions.

In this section, we outline the following: (1) the single global contract for evaluating both the quantity and quality of each peer's contributions, (2) a default policy for updating the overlay topology, and (3) a wire-level accounting protocol for verifying contributions.

5.1 Contribution contracts

In *Contracts*, each peer is evaluated for both 1) the amount that it contributes to directly connected peers, and 2) the amounts those peers contribute in turn; i.e., the quality of its selections. Peers with higher valuations will have a greater likelihood of being added to the peer lists of high capacity nodes and also enjoy prompt service when requesting individual blocks from those nodes. We next describe the details of how peer evaluations are computed.

Performance metrics: For two peers x and y, we denote the contribution rate from x to y by $B(x \rightarrow y)$, and compute this using a weighted moving average. $B(x)$ represents the total bandwidth contributed by node x. Each of these values is mapped to discrete classes of contribution—the deciles of the observed capacity distribution from PPlive. We label the bandwidth class of a node x as $BC(x)$.

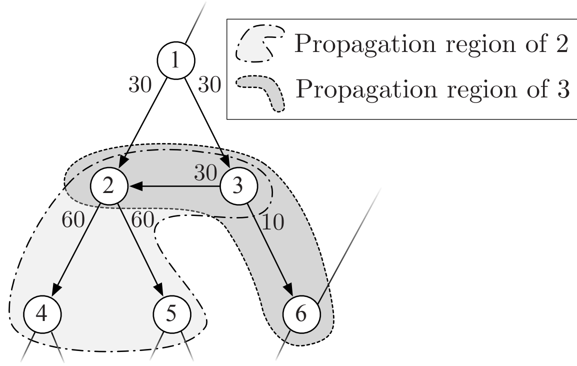


Figure 5: Evaluating I for client 1. Contribution from $1 \rightarrow 2$ is weighted by the rates from $2 \rightarrow 3$, 4, 5. Contribution of $1 \rightarrow 3$ is weighted by the rates from $3 \rightarrow 2$, 6.

To measure the effectiveness of contributions made by a client, we define $I(x)$ to be the one-hop propagation of x's contributions, calculating this as follows:

$$I(x) = \sum_{p \in \text{peers}(x)} B(x \rightarrow p) \times D_{\text{BW}}(BC(p)) \tag{1}$$

where $D_{\text{BW}} \in [0,1]$ is a weight specified by the cumulative distribution function of peer upload capacities.

As an example computation of I, consider Figure 5. In this case, the effectiveness of contributions from node 1 are being evaluated. The total contribution rates of peers 2 and 3 are 120 and 40, respectively. Mapping the values 120 and 40 to their bandwidth classes and looking them up in our measured capacity distribution yields: $D_{\text{BW}}(BC(3)) = 0.1$, $D_{\text{BW}}(BC(2)) = 0.8$. Substituting these values allows us to compute $I(1) = 30 \times 0.1 + 30 \times 0.8 = 27$.

Taken together, measures of contribution (BC) and effectiveness (I) constitute the global evaluation function, $V(x)$, which we define as the tuple $[BC(x), I(x)]$ with the following comparison operator:

$$V(x) > V(y) \iff BC(x) > BC(y) \text{ or } BC(x) = BC(y) \wedge I(x) > I(y).$$

In other words, peers are compared by their bandwidth class first, and peers within a class are compared according to the effectiveness of their contributions.

Servicing policy: The metrics defined above are used by clients to identify which peers are selected to receive service, the priority of that service, and which potential peers to prefer. We distinguish between *connection* and *selection* in our discussion of service policy. Connection is a prerequisite for being selected to receive service, and only connected peers exchange the control traffic necessary to compute $V(\cdot)$.

- *Peer selection:* Each node periodically rank-orders its peers by their corresponding $V(\cdot)$ values, selects the top k of these, where k is a configurable parameter, and notifies each that block requests will be accepted.
- *Block request servicing:* Among peers with outstanding requests, each client prioritizes the request from the peer with the maximum $B(\cdot)$ value.

In Section 5.3, we describe how each client reliably ascertains the performance metrics of its peers. Before doing so, we first analyze the incentive structure arising from this servicing policy.

What are the incentives provided by the system? Contracts provides strong contribution incentives by linking quality of service to effective contribution. A peer increases its chances of being *selected* for service and its priority by increasing its upload contribution. It might appear that contribution incentives are weakened by the use of bandwidth classes, as a peer p can lower its contribution while still remaining in its class. However, doing so reduces its service priority for block requests, which is determined by $B(p)$ among peers in its bandwidth class.

Contracts also rewards peers for making globally beneficial contributions. A peer that transmits blocks to higher capacity peers will achieve a higher evaluation under $I(\cdot)$ (and hence $V(\cdot)$), increasing the likelihood of being *selected* by others.

Will a provider enforce contracts? One possible deviation is for the provider p to ignore the rank ordering of $V(\cdot)$ values when choosing peers. In this case, a deviating provider *selects* a node y rather than x even though $V(x) > V(y)$. This ordering implies either $BC(x) > BC(y)$ or $BC(x) = BC(y) \wedge I(x) > I(y)$. In the former case, the provider's deviation lowers its own $I(\cdot)$ value since its contributions to y will be weighted less than its contributions to x. Hence, the deviation is not in its self-interest. In the latter case, $I(p)$ is unchanged by enforcing the contract, and hence p does not benefit from deviating.

Another possible deviation is for the provider to not provide prioritized service to higher capacity peers. For instance, a provider could transmit a block to y instead of x even though $B(x) > B(y)$. Again, this is not in the provider's self-interest since it reduces the provider's evaluation under $I(\cdot)$.

Why not other incentive structures? Initially, our definition of $V(\cdot)$ may seem somewhat arbitrary. Why not make effectiveness (I) fully recursive? That is, by including the contributions of peers, peers of peers, and so on. And, why use bandwidth classes rather than bandwidth itself? Or, why not use bandwidth only and ignore effectiveness? We tackle each of these questions in turn to provide additional intuition behind the development of our global contract.

We avoid a fully recursive definition of effectiveness for scalability. Accounting for the propagation of contri-

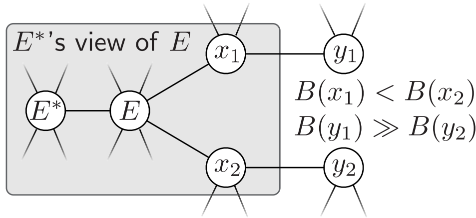


Figure 6: Under an alternate evaluation function $V'(\cdot)$, E has an incentive to unilaterally deviate when $B(x_2) > B(x_1)$ despite $V'(x_1) < V'(x_2)$.

butions globally creates significant overhead, both computationally and from increased control traffic. To reduce overhead, we limit propagation to one hop, with nodes percolating to their globally appropriate position through repeated cycles of evaluation and topology updates.

Unfortunately, limiting the propagation of accounting information creates an incentive to ignore the effectiveness of a peer's contributions, a crucial consideration when structuring the topology. As an example, consider a simpler evaluation function that uses bandwidth directly rather than bandwidth classes: $V'(x) = \sum_{p \in \text{peers}}(B(x \to p) \times D_{BW}(B(p))$. Under this evaluation function, a strategic client has an incentive to deviate. Consider the topology shown in Figure 6. In this case, the system would benefit from E contributing to x_1 since y_1 has much higher capacity than y_2, thus $V'(x_1) > V'(x_2)$. But, a client E^* evaluating E considers only the effectiveness of E, which is determined by the bandwidths of its peers only. Thus, when $B(x_2) > B(x_1)$, a rational E would contribute to x_2 rather than x_1, despite the greater redistribution capacity of x_1. This problem arises for any evaluation function with a limited view.

Bandwidth classes mitigate this problem by making peers with similar capacities incomparable at evaluators. When using V rather than V' to evaluate peers in Figure 6, E^* treats contributions to x_1 and x_2 equally (if they are in the same bandwidth class), eliminating the incentive for E to deviate. Our assumption is that clients are rational, but not Byzantine. Since deviating does not offer a local benefit, clients will split ties using contribution effectiveness, improving overall efficiency. We consider colluding and Byzantine peers in Section 5.4.

Finally, we incorporate effectiveness into our default contract rather than bandwidth alone in order to provide incentive-aware gossip, the topic we describe next.

5.2 Topology updating policy

Incentive-aware gossip: In addition to specifying how clients should make local servicing decisions, the contract also influences how the overlay topology is to be updated globally. To achieve a distribution structure where high capacity peers are closer to the source, *Contracts* uses peer gossip informed by $V(\cdot)$ as well as structural information provided by a hop count field in block messages. By maintaining the average hop count of blocks received from peers, each client can compare the average distance of its peers to the source, and we use this information to speed convergence when evaluating potential peers for *connection*.

Each client is aware of the capacities of its one-hop neighborhood of peers, and each client attempts to *connect* to the peers in this set with highest capacities. Universally applied, this results in the highest capacity peers percolating to the source, and lower capacity nodes being pushed to the periphery of the mesh through attrition.

Specifically, each client c sorts $p \in (\text{Peers} \circ \text{Peers})(c)$ by $BC(p)$, *connecting* to new peers in descending order. To split ties within a bandwidth class, c orders each potential peer by its average block hop count; i.e., a measure of the distance to the source, preferring the most distant of these. The intuition behind this policy is that the most distant peers within a bandwidth class are likely to be poorly clustered with respect to capacity and thus more likely to have outstanding block requests.[3] Preferential *connection* with misplaced peers in a bandwidth class speeds convergence of the topology. On the receiver's side, clients accept incoming *connections* optimistically, pruning those that have neither provided data nor warranted service in the recent past. Recall that connectivity does not imply that a peer will be *selected* for service. Exploring new nodes serves to expand the set of peers for which a given client can compute $V(\cdot)$.

This gossip strategy is *incentive-aware*; it incorporates the interests of clients seeking to maximize their V-value by contributing to the highest capacity peers. Consider the example topology in Figure 6. To compute $V(\cdot)$ for each of its peers, node E knows the bandwidth capacities of all labeled nodes (provided by our account mechanism). Suppose $BC(y_1) > BC(x_1)$. In this case, E would increase its V-value by sending to y_1 directly rather than through x_1, and so *connects* to y_1. Although x_1 might prefer to avoid this, revealing the bandwidth capacity of y_1 to E is required to demonstrate the effectiveness of its own contributions.

Bootstrapping new clients: For a newly joined, high capacity client to demonstrate its capability, it needs to receive stream data early enough to replicate that data widely. But, since a recently added client is typically placed far from the data source, the client might be overlooked simply because it could not receive enough useful data to replicate.

To address this problem, *Contracts* clients advertise an optional *bootstrapping block* comprised of random data. Advertising the bootstrapping block serves to inform directly connected peers that a client has excess capacity

[3] Preferring distant peers is a heuristic to increase trading opportunities. Recall that in a mesh with a capacity surplus, clients must compete to satisfy requests.

that can be verified through direct transfer. Each transferred bootstrapping block is worth half that of a normal block in terms of contribution value, although this value need not be precise. In light of the significantly skewed bandwidth distribution, our goal is to encourage meaningful contribution whenever possible while still allowing high capacity peers to demonstrate their capabilities.

Contracts's use of bootstrapping blocks exploits a key characteristic of the P2P environment, bandwidth asymmetry, to make a tradeoff beneficial to the overlay. Because many home broadband connections have significantly greater download capacity than upload capacity, identifying high capacity clients by downloading random data trades a reduction in abundant download capacity for an increase in upload capacity, the scarce resource. Of course, if a live broadcast has significantly more capacity than demand, bootstrapping blocks need not be transferred. To limit overhead, a *Contracts* client requests bootstrapping blocks only when it has excess capacity and with a probability that decreases as the number of its peers with excess capacity increases. This probability is given by: $1 - \frac{\text{Peers with excess capacity}}{\text{Total peers}}$.

5.3 Verifying contributions

The preceding topology update policies depend on the peers and the tracker obtaining truthful values of the global contributions of the peers (e.g., the calculation of Equation (1)). *Contracts* introduces *verifiable contributions* to support this task.

Each client P using *Contracts* completes a one-time registration step with the streaming infrastructure during which it is provided with a unique public/private key pair K_P. It should be clear in the context whether K_P represents the public key or the private key. The key pair serves as the client's identity and is persistent. Afterwards, clients are provided with two additional pieces of information when connecting to a channel: 1) a peer-specific nonce value and 2) the public key of the tracker of the channel. The nonce is used by several *Contracts* protocol messages to prevent replay attacks, and the tracker's public key is used to authenticate messages from the tracker that are forwarded in the overlay. We use $\langle M \rangle_K$ to denote a message M signed by key K.

When *Contracts* clients connect to one another, they exchange their respective public keys and nonce values.[4] Afterwards, data is exchanged normally. Periodically during data transfer, each *Contracts* client mints a signed receipt message for each of its peers. Each receipt accounts for the most recent contributions of that client and is sent to the remote endpoint. For example, if a client P with key pair K_P has received V blocks of data from a peer Q since it last sent a receipt to Q, P sends Q a receipt containing $\langle N_Q, K_P \rightarrow K_Q : V \rangle_{K_P}$. This includes a nonce (N_Q), the sender and receiver identities, and the number of blocks, signed by P.[5] Receipts are sent when a threshold on the volume of sent data is reached. This threshold is set by the tracker to control load and overhead.

Receipts serve as the foundation for verified contributions in *Contracts*, and we describe both distributed and centralized methods for using them to evolve the overlay topology. Distributed verification reduces load on the tracker, increasing scalability. Centralized verification speeds topology updates and reduces total network overhead, while also precluding several attacks from Byzantine users. These methods are not mutually exclusive; either (or both) can be used during a broadcast, and clients can switch between them freely depending on the level of contention for tracker resources. We describe each in turn.

Distributed verification: When using distributed verification, the tracker bootstraps new peers by providing a random subset of candidates to each client, and timestamps are used as nonce values. Each client forwards all receipts it receives due to contributions to its directly connected peers, including receipts collected from its one-hop neighborhood. Unlike the tracker, which generates the keys for all valid users in the broadcast, ordinary peers that receive receipts cannot distinguish valid identities from those generated by a strategic user. If any receipt is accepted, such users can manufacture an arbitrary number of receipts and claim any level of contribution. Thus, the challenge for verification in the distributed case is identifying valid receipts.

To do this, the tracker issues each user a small valid user message when the user first joins a given broadcast. This message is signed by the tracker and includes a timestamp, channel identifier, and the public key of the recipient. When peers connect to one another, they exchange and verify valid user messages, demonstrating to one another that they are a valid peer for the given broadcast.

Centralized verification: Although conceptually straightforward, distributed verification and contract enforcement assumes rational clients. A large number of Byzantine clients may undercut the convergence of our topology structuring algorithm, degrading performance. To address this, *Contracts* supports evaluating peers and enforcing topology updates at the tracker if necessary. Centralized topology updates also enable rapid and/or fine-grained adjustments to the topology during challenging workloads, e.g., flash crowds. The primary challenge to centralizing these functions is ensuring that the tracker is not overwhelmed with network traffic

[4] Man-in-the-middle attacks can be precluded by bundling peer keys with the peer list returned by the tracker. This increases overhead and is optional.

[5] For brevity, we omit the broadcast identifier, also included.

Computing the digest of receipts at peer Q

```
1: Digest ← ∅
2: Hash ← 0
3: Sort receipts by sender key
4: for each receipt R : {N, K_P → K_Q : V}
5:     from client P to Q with block count V; do
6:   Increment counter for K_P in Digest
7:   Hash ← SHA-1(Hash * SHA-1(R))
8: done
9: Send {Digest, Hash} to coordinator
```

Figure 7: Construction of the receipt digest message at a client Q. The $*$ operator indicates concatenation.

or computational demands, and the remainder of this section describes the mechanisms *Contracts* uses to achieve this.

Periodically, each client contacts the tracker to report its continuing participation in the broadcast and requests an updated set of peers. In the current PPLive implementation, this message also includes the client's maximum upload rate as measured by the client. *Contracts* piggybacks on this message, replacing the self-reported upload rate with a verifiable accounting of blocks contributed to specific peers during the previous update interval. Since public keys (and hence individual receipts) are lengthy, the naïve approach of simply forwarding all receipts to the tracker would amount to a de-facto DDoS attack. Instead, *Contracts* clients report a compact, plain-text *digest* of receipts.

The algorithm for constructing the receipt digest message is given in Figure 7. The key underlying technique is to trade optional computation at the tracker for a substantial reduction in network traffic. Instead of transmitting full receipts, each digest contains *claims* about receipts received and a verification hash. Claims are a plain-text list of contributions that allows the tracker to reconstruct the original contribution receipts by recomputing them. A digest contains claims for each receipt received since the last digest was sent (line 4). Each claim contains the first n bits of the public key of the receiver specified in the full receipt (line 6). Each truncated key serves as an index, allowing the tracker to map an identifier to a public/private key pair it previously generated for a particular user. Finally, a hash chain is computed over the original receipts (line 7) sorted by receiver identifier (line 3). This can be used by the tracker to verify that claims correspond to valid receipts.

A list of claims informs the tracker as to which receivers generated receipts, but to recompute those original receipts and verify the hash chain, the tracker also needs to know the number of blocks received and the receipt nonce. Both of these are set by the tracker when clients initially connect. The block threshold for dispatching receipts, V, is set to control overhead both at the tracker and among clients. Each client's nonce is selected at random by the tracker and incremented by clients per-peer for each receipt received. For example, if a client's initial nonce is 5 and it receives 2 receipts from peer A and 3 from peer B in a given reporting interval, subsequent receipts minted by A and B to this client will be stamped with nonce values of 7 and 8, respectively. The tracker verifies increments to nonce values to prevent replay attacks, and nonce values are maintained on a per-peer basis to prevent concurrent data transfers from producing receipts with the identical nonce values.

At the tracker, ranking clients based on the plain-text claims in digests requires little overhead relative to the existing processing already done by the tracker; table lookups provide the required information to compute Equation (1) (where the sum of contributed block claims per update interval provides contribution rates). Although *processing* is straightforward, *verification* is computationally intensive, requiring the tracker to regenerate and hash each signed receipt. But, since only the plain-text content of digests is needed to rank clients, the tracker can shed load at any time. While sampling digests may increase susceptibility to cheating, our evaluation shows that verifying all digests on the fly is feasible given PPLive's current infrastructure provisioning.

5.4 Collusion resistance

Contracts includes both centralized and distributed verification of receipts to allow the tracker to manage the tradeoff between protocol overhead and robustness to malicious behavior. In the absence of Byzantine behavior, distributed verification effectively rewards contribution without relying on centralized accounting. With isolated Byzantine agents, coordinating topology updates at the tracker enables convergence even while some nodes deviate from our default contract. This increases overhead, but as we show in our evaluation, not prohibitively.

In the remainder of this section, we describe the techniques used by a tracker to use its global perspective to mitigate security attacks, in particular, the well-known P2P attack: collusion, in which a group of participants work collectively to subvert our accounting mechanism. The collusion participants we consider may include both real users with interest in receiving stream data, as well as synthetic identities created strictly for collusion.

Limited identity creation: The tracker appeals to standard techniques used by other P2P proposals for inhibiting the creation of arbitrarily many synthetic identities, the so-called Sybil attack [7]. In particular, the tracker limits the creation of new identities on the basis of durable identifiers, e.g., cell phone number via SMS.

Flow integrity check: When a new client joins a broadcast, the tracker evaluates its maximum upload capacity. Although a client may choose to upload at a lower rate, it cannot exceed the capacity. This restricts potential false

claims on $BC(\cdot)$. In addition, live streaming imposes a known incoming rate bound on each client's long-term incoming data rate, which is the streaming rate. When verifying receipts, the tracker validates the upload capacity and incoming rate bounds. Such verification limits the collusion of a set of broadcast participants to issue fraudulent receipts. No group of colluders can form a loop and arbitrarily boost a colluder's contribution value. Specifically, consider a client x with the support of a total of K colluders. Assume that x is an actual broadcast participant that needs to receive actual data from non-colluders. Then x cannot issue any fraudulent receipts, as it needs to issue receipts for actual data. The capability of the colluders to help x is also limited. The value $B(x \rightarrow p)$, where p is a colluder, is limited by the streaming rate r due to incoming rate bound on p. Thus, with K colluders generating fraudulent receipts, x can claim at most $K \cdot r$ fraudulent contributions to these colluders. But, $K \cdot r$ cannot exceed the upload rate of x measured by the tracker for WAN traffic. Further, if a given colluder p helps x to claim contribution rate r, then $B(p' \rightarrow p)$ should be zero for any other client p', otherwise p violate its incoming rate bound. Thus, if a collusion scheme is to let $B(x \rightarrow p)$ be r for all K colluders, then $B(p)$ has to be zero for all of the colluders. This substantially limits $I(x)$.

Global and diversity weighting: In spite of the preceding checks, some clients might still be able to collude and/or acquire several synthetic identities to increase the overall value of $V(\cdot)$ of a client. To address this, the tracker detects a cluster of linked colluders. Also, *Contracts* can optionally weight the overall value of $V(\cdot)$ by the network-level address diversity of the peers to which a client contributes. As a consequence of registering for a broadcast, the tracker knows each client's IP address and port. For identities within the same IP prefix (/24), *Contracts* dampens the value of contributions when using centralized verification. For identities registered at the same address (e.g., users behind a NAT), contributions are further dampened. This policy restricts collusion by exploiting the scarcity of IP addresses.

Note that we do not adopt a universal notion of client utility, and we do not claim that *Contracts* is strategy-proof, even given these defenses. An alternative approach to mitigating collusion and strategic behavior is to restrict each client's choice in peer selection. As shown in previous work [17, 18], limiting peer selection is a powerful tool for enabling formal analysis of gossip protocols since the potential for protocol deviations is restricted. But, such restrictions may limit the potential for grouping peers based on locality or bandwidth, e.g., high bandwidth, local exchange between peers on the same LAN. In practice, flexible peering significantly increases distribution efficiency in PPLive, leading us to eschew restrictions which may aid in formal analysis, leaving open these issues for future work.

6 Evaluation

Our evaluation of *Contracts* answers two main questions. First, is applying *Contracts* to streaming systems feasible? We find that it is; *Contracts* adds modest overhead but does not fundamentally limit scalability. Second, is *Contracts* effective? To confirm this, we report measurements of a modified PPLive client to demonstrate the performance improvement of *Contracts* relative to other systems and incentive strategies. Specifically:

- *Contracts* improves performance relative to unmodified PPLive. In experiments with heterogeneous capacities and churn, *Contracts* increases the number of clients with uninterrupted playback from 13% to 62%, an increase of more than 4×.
- *Contracts* provides robust contribution incentives. Experiments in bandwidth constrained environments show that quality of service improves with contribution. Moreover, *Contracts* provides a substantial and consistent improvement in quality of service relative to tit-for-tat.
- *Contracts* is scalable, even when using centralized verification. Using our default parameters, a single *Contracts* tracker can support the computational and network overhead of more than 90,000 concurrent clients.
- Clients are quickly integrated into the mesh. After only a few rounds of peer exchanges, newly joined clients percolate to their intended locations in the overlay with bandwidth clustered peers.

6.1 Performance and incentives

We first evaluate the performance of our *Contracts* implementation, which is built from modifications to the reference PPLive client. We show two main results: 1) PPLive with *Contracts* significantly outperforms both unmodified PPLive and one modified to support tit-for-tat (TFT). 2) *Contracts* provides our intended contribution incentives; when the system is bandwidth constrained, increasing contribution improves performance.

Performance: We define performance as the fraction of data blocks received by their playback deadlines, and compare performance for PPLive, PPLive using *Contracts*, PPLive with tit-for-tat, and FlightPath [17]. For each of these techniques, we measure the performance of 100 clients participating in a test broadcast on Emulab.[6] Each client initially joins the system separated by a ten-second interval. To evaluate *Contracts* under churn, each client disconnects and rejoins after participating for 20 minutes. All clients continue this process for two hours. To compare performance under realistic bandwidth constraints, client upload capacities are drawn from our mea-

[6]Emulab allows us to execute Windows binaries.

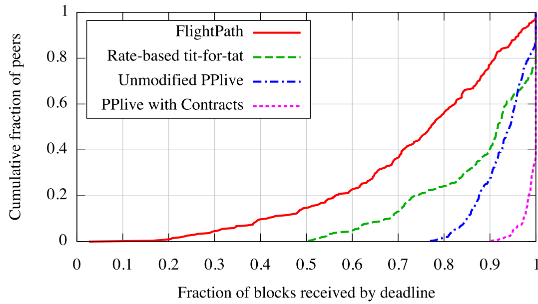


Figure 8: Performance comparison of unmodified FlightPath, PPLive, rate-based tit-for-tat, and *Contracts*.

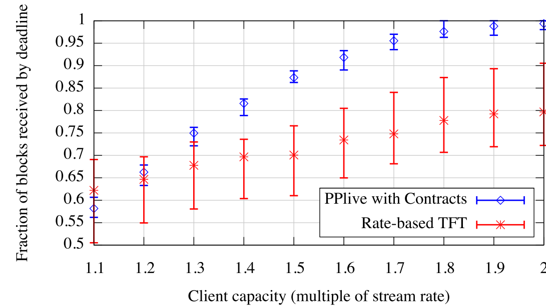


Figure 9: Delivery rate as a function of contribution.

sured capacity distribution of PPLive clients, normalized to provide an over-provisioning factor of 2; i.e., the sum of peer capacities is twice the aggregate demand. Crucially, however, many peers have capacity less than the stream data rate—a common occurrence in practice. Both TFT and *Contracts* clients actively exchange data with 10 directly connected peers and reevaluate these decisions every 10 seconds using the statistics of previous 30 seconds. For FlightPath trails, we use default configuration parameters described by Li, et al. [17].

Figure 8 shows our results. *Contracts* significantly improves performance relative to unmodified PPLive and FlightPath; 62% of *Contracts* clients experience loss-free playback compared with just 13% when using unmodified PPLive or 3% when using FlightPath. In other words, the fraction of PPLive/*Contracts* clients experiencing loss-free playback is more than 4 times that of unmodified PPLive. For clients that do miss playback deadlines, a larger fraction of blocks arrive in time when using *Contracts*. Relative to unmodified PPLive, tit-for-tat degrades performance for the majority of clients. This is consistent with our analysis in Section 3. Tit-for-tat benefits high capacity clients when they happen to be placed near the broadcast source ($y > 0.96$). But, more distant clients cannot collect enough useful data with which to trade. Even high capacity clients cannot prove their capabilities when far from the source, decreasing overall utilization and average performance.

Incentives: *Contracts* rewards contribution with increased robustness. We evaluate this by comparing the performance of PPLive using *Contracts* with that of PPLive using tit-for-tat. In both cases, the system is bandwidth constrained. We use 100 clients with capacities uniformly distributed between 1–2× the stream rate (over-provisioning factor 1.5) to connect to a test stream, participating in the broadcast for 10 minutes. We repeat this experiment 10 times.

Figure 9 shows the results. Averages are shown with error bars giving the full range of block delivery rates for clients with a given capacity. While tit-for-tat does provide some correlation between contribution and performance, the amount of improvement varies significantly because tit-for-tat does not update the topology. In contrast, *Contracts* combines both topology updates and local servicing rate decisions to provide a consistent improvement in performance, strengthening incentives.

6.2 Overhead

In this section, we describe implementation details and overhead related to verifying contributions, including: 1) state maintained by the tracker and clients, 2) computation required to verify receipts, and 3) network control traffic. We discuss each of these in turn.

State: When using centralized verification, the PPLive tracker maintains soft state including bandwidth capacity, client version, etc. of active clients. To these, *Contracts* adds a last digest update field which records the timestamp and content of the most recently received receipt digest message. This is used to compute contribution rates when new digests are received, and its size varies depending on content. We estimate the likely size of receipt digest messages when computing network overhead (described below).

Trackers also maintain hard state: the key pairs of registered clients. For cryptographic operations, *Contracts* uses SHA-1 with RSA, DER-encoded PKCS#1 and 1024 bit keys. Maintaining one million key pairs requires less than a gigabyte of storage on disk. A lookup table mapping truncated identifiers to keys easily fits in memory on modern servers. For distributed verification, clients associate public keys and nonces with connections and maintain counters of verified receipts received from each directly connected peer.

Network traffic: Exchanging receipt and receipt digest messages is the main source of network overhead in *Contracts*. Three related parameters influence this. The tracker specifies a *digest interval* indicating how often digests are reported by clients. A lengthy interval reduces the number of such messages at the cost of delayed topology updates or delayed detection of cheating clients. The *receipt volume* specifies how much data each receipt acknowledges. Finally, the *stream data rate* controls how many receipts are exchanged among peers. To make our analysis independent of stream data rate, we define receipt volume in terms of how many seconds of

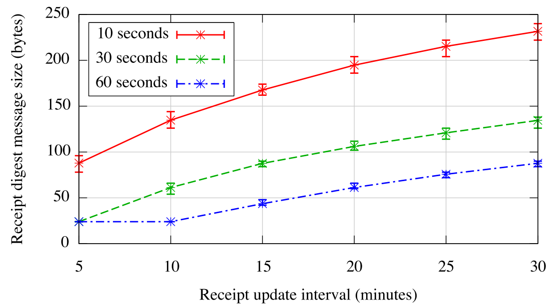


Figure 10: The size of receipt digest messages as a function of the digest update interval.

video data each receipt acknowledges. Currently, *Contracts* uses a digest update interval of 15 minutes and a receipt volume that acknowledges 30 seconds of stream data. In the remainder of this section, we examine the tradeoffs underlying these choices.

For a video stream of moderate quality (500 Kbit), sending a receipt acknowledging every 30 seconds of video data imposes less than 0.1% overhead relative to data transfer among peers when verifying contributions at the tracker. Distributed verification requires forwarding additional receipts from each peer's one hop neighborhood. This increases average network overhead to 1.2%, trading an increase in traffic among peers for a reduction in traffic at the tracker, which we consider next.

Network overhead at the tracker is determined by the number of receipt digest messages received. Each receipt digest message contains a 24 byte header and 6 byte tuples specifying a peer (4 byte truncated public key) and a receipt count (2 bytes). In the worst case, each digest would include an entry for every directly connected peer.

In practice, only a fraction of connected peers are included in a single digest update. To compute this, we measured the amount of data uploaded to directly connected peers by an instrumented PPLive client that participated in popular broadcasts for 10 minutes apiece. Contribution is highly skewed; for each client, the top 10% of its peers receive 60% of its contributed data, meaning that there are fewer entries in each digest.

Combining our measurements of skew with the typical number of directly connected peers allows us to compute the size of receipt digest messages. Figure 10 shows this data for several receipt volume values. Each line shows the growth in the size of a digest message as a function of the update interval. Each data point is averaged over 10,000 randomly generated digest messages using samples from measured distributions to specify the directly connected peers and capacity skew. To compute aggregate traffic at the tracker, we multiply the average receipt digest size by the total number of clients. For instance, processing digests for 100,000 clients with our default parameters requires 10 KBps of tracker overhead.

Computation: Computational requirements at the clients are dominated by the demands of video playback. At the tracker, the computational overhead of *Contracts* is dominated by receipt verification. Verification requires regenerating the receipt messages specified by receipt digest messages and computing the SHA-1 hash chain for the generated receipts to verify the hash specified in the digest (Figure 7). Thus, the computational overhead of verification depends on the number of receipts, which is determined by the stream data rate and receipt volume.

The total number of receipts per second generated by a channel is simply the ratio of data rate and receipt volume multiplied by the population. A micro-benchmark on a single commodity server using our current implementation can verify 3,200 receipts per second, and receipt verification is embarrassingly parallel. If receipts encapsulate 30 seconds worth of video data, our current implementation can verify receipts for more than 90,000 simultaneous clients in real-time using a single server. In practice, management of so large a broadcast is already distributed across several servers in PPLive, meaning that receipt verification with *Contracts* does not dominate resource usage when scaling the coordination infrastructure. As with network overhead, *Contracts* allows the tracker to shed computational load when required. receipt digest messages that are not cryptographically verified can still be used to evolve the topology and (optionally) stored for later verification. This increases the window of vulnerability to a cheating client but does not degrade the efficiency of distribution.

6.3 Convergence

We next consider the integration of new clients into the mesh. Convergence of clients to their intended location in the topology is determined by many factors. We consider two explicitly: 1) the capacity of a newly joined client, and 2) the number of newly joining clients; e.g., integrating a flash crowd may require additional peer exchanges relative to integrating a single client into a stable mesh. We measure convergence in terms of update rounds; i.e., the interval between peer gossip connections. To understand convergence at scale, we use trace-driven simulation of *Contracts* using default parameters.

We first evaluate convergence as a function of a newly joined client's bandwidth capacity. For each capacity, the new client joins a 10,000 user channel with stable membership, and we record the number of topology updates required for the newly joined client to reach a stable position. We consider a client to have reached a stable position when the average capacity of its net contributors (i.e., those that provide more blocks than they receive) is within 5% of the average capacity of net consumers. The vast majority of peers ($>80\%$) reach a stable position in four update rounds or less. Broadly, the results are consistent with the variation in observed bandwidth capacity.

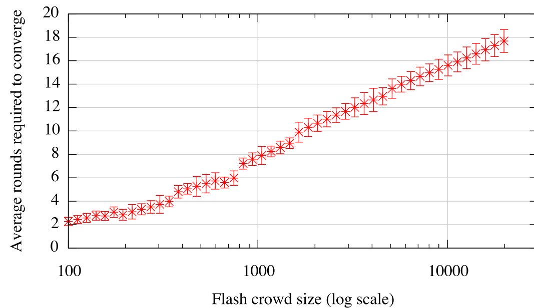


Figure 11: The number of peer exchanges required for a set of newly joined clients to reach stable matchings as a function of the number of arrivals in a flash crowd.

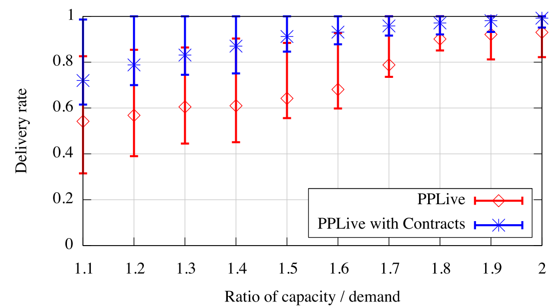


Figure 12: The impact of over-provisioning on PPLive's performance. Data points show the average fraction of blocks received by their playback deadlines.

Low capacity peers can quickly discover a stable set of similarly low capacity peers, while high capacity peers need several rounds to stabilize.

Next, we consider topology convergence for flash-crowd arrivals. In this case, we simulate a channel with 1,000 initial participants and vary the number of joining clients. Each new client is assigned a capacity drawn from the same distribution as the existing clients, providing a constant amount of resources in the system. Figure 11 shows the number of rounds required to achieve stability for the last newly joined client in the crowd. The number of rounds required increases logarithmically with the number of joining peers.

6.4 Over-provisioning

Our evaluation thus far has focused on the performance of *Contracts* in settings with a specific amount of over-provisioning; i.e., capacity in excess of total demand. We now evaluate over-provisioning directly, measuring the performance of PPLive and *Contracts* while scaling our measured capacity distribution to vary the ratio of capacity to demand. We measure the block delivery rate of 100 static PPLive clients running on Emulab. As in previous experiments, we record each client's block delivery rate. Figure 12 summarizes the results, with error bars showing the 5th and 95th percentiles of delivery rate across all clients. In each trial, the average delivery rate PPLive using *Contracts* exceeds that of unmodified PPLive. When capacity is limited, low capacity clients are penalized by *Contracts*, contributing to variations in performance. As capacity increases, however, *Contracts* delivers consistently higher quality of service for all peers. Taken together, these results show that *Contracts* achieves consistently higher performance for a range of operating conditions, and delivers on our overall goals of improving efficiency and providing contribution incentives. When the system is capacity rich, *Contracts* improves distribution efficiency, improving delivery rate for all peers. But, during periods of resource contention, high capacity peers receive better quality of service.

7 Related work

Our work builds on a large body of prior work focused on live streaming, P2P data distribution, and incentives.

The notion of a P2P approach to data streaming was pioneered by Narada, Overcast and Yoid [4, 14, 8]. These designs tried to approximate multicast support using a tree structured overlay. SplitStream, a subsequent design, addressed the limited utilization of leaf nodes in tree-based schemes [2].

Subsequent work has applied swarming designs, borrowed from BitTorrent-like systems, to video-on-demand and live streaming. Coolstreaming/DONet applies a mesh-based network structure to live streaming [28]. Annapureddy, et al. argue that high quality video on demand is feasible using a P2P architecture, a point reinforced by recent work describing PPLive's video-on-demand P2P implementation [13] as well as other publicly available commercial streaming systems (e.g., PPStream, SopCast, TVAnts, and UUSee).

More recent work has studied incentives in bulk data distribution in widely deployed systems, particularly BitTorrent. Qiu and Srikant studied BitTorrent formally, finding that it achieves a Nash equilibrium under certain conditions [25], although more recent work has shown practical mechanisms for subverting BitTorrent's incentive strategy [22]. These advancements in understanding the subtlety of bilateral exchange motivated our consideration of its application to live streaming. In [1], Aperjis, et al. extend bilateral exchange to multilateral exchange by introducing prices. They compare their scheme with BitTorrent and show improvements in efficiency and robustness. One hop reputations [23] use limited propagation of contribution information to improve incentives in BitTorrent; we apply similar ideas to live streaming.

Most related to our work are systems that address incentives in live streaming (e.g., [10, 12, 18, 19, 21, 26, 27]). Sung, et al. describe a live streaming design that rewards contribution but depends on honest capacity reporting by peers [27]. SecureStream introduces proto-

col mechanisms to defend against several attacks (e.g., forged data and denial of service [12]). These techniques are largely complementary to our work, which focuses on verifiably rewarding contribution. BAR Gossip analyzes incentives in streaming formally and describes a protocol designed to induce contributions from rational users [18]. FlightPath relaxes several constraints of BAR Gossip (e.g., by allowing dynamic membership), but still requires the long-term balance of contribution and consumption to provide contribution incentives. Our experience applying rate-based tit-for-tat is consistent with that of Pianese, et al. who apply TFT to live streaming and experimentally confirm the need for significant altruism to achieve robustness [21].

Motivated by the practical challenges of client heterogeneity, we take a different approach in the design of *Contracts*, providing incentives via a global contract and including explicit topology restructuring in our algorithm design. Habib, et al. propose providing high capacity clients with additional peers to improve their service quality [10], but such improvements are not assured in environments with high levels of bandwidth heterogeneity.

Several live-streaming systems focus on providing robustness by enforcing contribution amounts. Chu, et al. propose mandatory, centrally enforced taxation in the context of multi-tree live streaming [3]. Haridasan, et al. provide a two-level auditing scheme for live streaming that ensures that peers contribute more than a threshold amount of data [11]. Local auditing and gossip provide an immediate but partial check on user's contribution, while global audit ensures that a misbehaving node is caught. Rather than punishing nodes that do not contribute a sufficient amount, *Contracts* rewards nodes for voluntarily contributing as much as possible.

8 Conclusion

We have examined performance and contribution incentives for live streaming systems. The unique features of the P2P live streaming environment limit the effectiveness of many widely-used incentive strategies based on balanced or bilateral exchange. These challenges motivate the design of *Contracts*, a new incentive strategy that rewards contribution with quality of service by evolving the overlay topology. Building on a protocol that provides verifiable contributions, we have shown that the use of *Contracts* both improves performance relative to PPLive and strengthens contribution incentives relative to existing approaches without curtailing scalability.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Lorenzo Alvisi, for their valuable feedback.

References

[1] C. Aperjis, M. Freedman, and R. Johari. Peer-assisted content distribution with prices. In *CoNEXT*, 2008.

[2] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: high-bandwidth multicast in cooperative environments. In *SOSP*, 2003.

[3] Y. Chu, J. Chuang, and H. Zhang. A case for taxation in peer-to-peer streaming broadcast. In *SIGCOMM PINS*, 2004.

[4] Y. Chu, S. Rao, and H. Zhang. A case for end system multicast. In *SIGMETRICS*, 2000.

[5] B. Cohen. Incentives build robustness in BitTorrent. In *P2P-ECON*, 2003.

[6] Comcast limits download volume. `http://online.wsj.com/article/SB122004003325884079.html`.

[7] J. R. Douceur. The Sybil attack. In *IPTPS*, 2002.

[8] P. Francis. Yoid: Extending the internet multicast architecture. Available at `http://www.icir.org/yoid/docs/`, 2000.

[9] C. Gkantsidis, J. Miller, and P. Rodriguez. Comprehensive view of a live network coding P2P system. In *IMC*, 2006.

[10] A. Habib and J. Chuang. Service differentiated peer selection: an incentive mechanism for peer-to-peer media. In *IEEE Trans. Multimedia*, 2006.

[11] M. Haridasan, I. Jansch-Porto, and R. van Renesse. Enforcing fairness in a live-streaming system. In *MMCN*, 2008.

[12] M. Haridasan and R. van Renesse. SecureStream: An intrusion-tolerant protocol for live-streaming dissemination. *Computer Communication*, 31(3):563–575, 2008.

[13] Y. Huang, T. Fu, D. Chiu, J. Lui, and C. Huang. Challenges, design and analysis of a large-scale P2P-VoD system. In *SIGCOMM*, 2008.

[14] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole. Overcast: Reliable multicasting with on overlay network. In *OSDI*, 2000.

[15] J.-J. Laffont and D. Martumort. *The Theory of Incentives: The Principal-agent model*. Princeton University Press, 2002.

[16] D. Levin, K. LaCurts, N. Spring, and B. Bhattacharjee. BitTorrent is an auction: analyzing and improving BitTorrent's incentives. In *SIGCOMM*, 2008.

[17] H. Li, A. Clement, M. Marchetti, M. Kapritsos, L. Robinson, L. Alvisi, and M. Dahlin. FlightPath: Obedience vs choice in cooperative services. In *OSDI*, Dec 2008.

[18] H. Li, A. Clement, E. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin. BAR gossip. In *OSDI*, Nov. 2006.

[19] Z. Liu, Y. Shen, S. Panwar, K. Ross, and Y. Wang. Using layered video to provide incentives in P2P live streaming. In *P2P-TV*, 2007.

[20] Z. Liu, C. Wu, B. Li, and S. Zhao. Distilling superior peers in large-scale P2P streaming systems. In *INFOCOMM*, 2009.

[21] F. Pianese, D. Perino, J. Keller, and E. W. Biersack. PULSE: An adaptive, incentive-based, unstructured P2P live streaming system. *IEEE Transactions on Multimedia*, 2007.

[22] M. Piatek, T. Isdal, T. Anderson, A. Krishnamurthy, and A. Venkataramani. Do incentives build robustness in BitTorrent? In *NSDI*, 2007.

[23] M. Piatek, T. Isdal, A. Krishnamurthy, and T. Anderson. One hop reputations for P2P file sharing workloads. In *NSDI*, 2008.

[24] M. Piatek, A. Krishnamurthy, A. Venkataramani, R. Yang, and D. Zhang. Contracts: Practical contribution incentives for P2P live streaming. Technical Report, UW CSE, 2010.

[25] D. Qiu and R. Srikant. Modeling and performance analysis of BitTorrent-like peer-to-peer networks. In *SIGCOMM*, 2004.

[26] T. Silverston, O. Fourmaux, and J. Crowcroft. Towards an incentive mechanism for peer-to-peer multimedia live streaming systems. In *International Conference on Peer-to-Peer Computing*, 2008.

[27] Y.-W. Sung, M. Bishop, and S. Rao. Enabling contribution awareness in an overlay broadcasting system. *SIGCOMM*, 2006.

[28] X. Zhang, J. Liu, B. Li, and T.-S. P. Yun. CoolStreaming/DONet: A data-driven overlay network for live media streaming. In *INFOCOMM*, 2005.

Experiences with CoralCDN: A Five-Year Operational View

Michael J. Freedman
Princeton University

Abstract

CoralCDN is a self-organizing web content distribution network (CDN). Publishing through CoralCDN is as simple as making a small change to a URL's hostname; a decentralized DNS layer transparently directs browsers to nearby participating cache nodes, which in turn cooperate to minimize load on the origin webserver. CoralCDN has been publicly available on PlanetLab since March 2004, accounting for the majority of its bandwidth and serving requests for several million users (client IPs) per day. This paper describes CoralCDN's usage scenarios and a number of experiences drawn from its multi-year deployment. These lessons range from the specific to the general, touching on the Web (APIs, naming, and security), CDNs (robustness and resource management), and virtualized hosting (visibility and control). We identify design aspects and changes that helped CoralCDN succeed, yet also those that proved wrong for its current environment.

1 Introduction

The goal of CoralCDN was to make desired web content available to everybody, regardless of the publisher's own resources or dedicated hosting services. To do so, CoralCDN provides an open, self-organizing web content distribution network (CDN) that any publisher is free to use, without any prior registration, authorization, or special configuration. Publishing through CoralCDN is as simple as appending a suffix to a URL's hostname, *e.g.*, `http://example.com.`*`nyud.net`*`/`. This URL modification may be done by clients, origin servers, or third parties that link to these domains. Clients accessing such *Coralized* URLs are transparently directed by CoralCDN's network of DNS servers to nearby participating proxies. These proxies, in turn, coordinate to serve content and thus minimize load on origin servers.

CoralCDN was designed to automatically and scalably handle sudden spikes in traffic for new content [14]. It can efficiently discover cached content anywhere in its network, and it dynamically replicates content in proportion to its popularity. Both techniques help minimize origin requests and satisfy changing traffic demands.

While originally designed for decentralized and unmanaged settings, CoralCDN was deployed on the PlanetLab research network [27] in March 2004, given PlanetLab's convenience and availability. CoralCDN has since remained publicly available for more than five years at hundreds of PlanetLab sites world-wide. Accounting for a majority of public PlanetLab traffic and users, CoralCDN typically serves several terabytes of data per day, in response to tens of millions of HTTP requests from around two million users (unique client IP addresses).

Over the course of its deployment, we have come to acknowledge several realities. On a positive note, CoralCDN's notably simple interface led to widespread and innovative uses. Sites began using CoralCDN as an elastic infrastructure, dynamically redirecting traffic to CoralCDN at times of high resource contention and pulling back as traffic levels abated. On the flip side, fundamental parts of CoralCDN's design were ill-suited for its deployment and the majority of its use. If one were to consider the various reasons for its use—for resurrecting long-unavailable sites, supporting random surfing, distributing popular content, and mitigating flash crowds—CoralCDN's design is insufficient for the first, unnecessary for the second, and overkill for the third, at least given its current deployment. But diverse and unanticipated use is unavoidable for an open system, yet openness is a necessary design choice for handling the final flash-crowd scenario.

This paper provides a retrospective of our experience building and operating CoralCDN over the past five years. Our purpose is threefold. First, after summarizing CoralCDN's published design [14] in Section §2, we present data collected over the system's production deployment and consider its implications. Second, we discuss various deployment challenges we encountered and describe our preferred solutions. Some of these changes we have implemented and incorporated into CoralCDN; others require adoption by third-parties. Third, given these insights, we revisit the problem of building a secure, open, and scalable content distribution network. More specifically, this paper addresses the following topics:

- *The success of CoralCDN's design given observed usage patterns (§3).* Our verdict is mixed: A large majority of its traffic does not require any cooperative caching at all, yet its handling of flash crowds relies on such cooperation.
- *Web security implications of CoralCDN's open API (§4).* Through its open API, sites began leveraging CoralCDN as an elastic resource for content distri-

bution. Yet this very openness exposed a number of web security challenges. Many can be attributed to a lack of explicitness for specifying appropriate protection domains, and they arise due to violations of traditional security principles (such as least privilege, complete mediation, and fail-safe defaults [33]).

- *Resource management in CDNs (§5).* CoralCDN commonly faced the challenge of interacting with oversubscribed and ill-behaved resources, both remote origin servers and its own deployment platform. Various aspects of its design react conservatively to change and perform admission control for resources.
- *Desired properties for deployment platforms (§6).* Application deployments could benefit from greater visibility into and control over lower layers of their platforms. Some challenges are again confounded when information and policies cannot be expressed explicitly between layers.
- *Directions for building large-scale, cooperative CDNs (§7).* While using decentralized algorithms, CoralCDN currently operates on a centrally-administered, smaller-scale testbed of trusted servers. We revisit the challenge of escaping this setting.

Rather than focus on CoralCDN's self-organizing algorithms, the majority of this paper analyzes CoralCDN as an example of an open web service on a virtualized platform. As such, the experiences we detail may have implications to a wider audience, including those developing distributed hash tables (DHTs) for key-value storage, CDNs or web services for elastic provisioning, virtualized network facilities for programmable networks, or cloud computing platforms for virtualized hosting. While many of the observations we report are neither new nor surprising in hindsight, many relate to mistakes, oversights, or limitations of CoralCDN's original design that only became apparent to us from its deployment.

We next review CoralCDN's architecture and protocols; a more complete description can be found in [14]. All system details presented after §2 were developed subsequent to that publication. We discuss related work throughout the paper as we touch on different aspects of CoralCDN.

2 Original CoralCDN Design

The Coral Content Distribution Network is composed of three main parts: (1) a network of cooperative HTTP proxies that handle client requests from users, (2) a network of DNS nameservers for `nyud.net` that map clients to nearby CoralCDN HTTP proxies, and (3) the underlying Coral indexing infrastructure and clustering machinery on which the first two applications are built. This paper consistently refers to the system's *indexing* layer as Coral, and the entire content distribution system as CoralCDN.

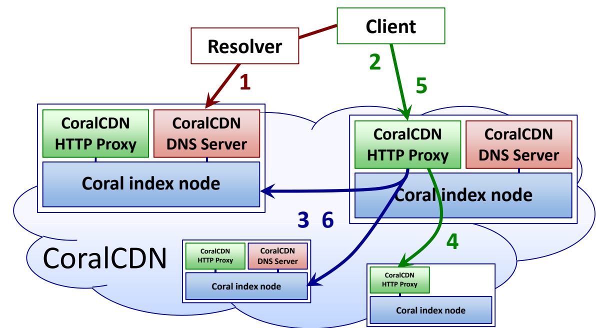


Figure 1: **The steps involved in serving a Coralized URL.**

2.1 System overview

At a high level, the following steps occur when a client issues a request to CoralCDN, as shown in Figure 1.

1. ***Resolving DNS.*** A client resolves a "Coralized" domain name (*e.g.*, of the form `example.com.nyud.net`) using CoralCDN nameservers. A CoralCDN nameserver probes the client to determine its round-trip-time and uses this information to determine appropriate nameservers and proxies to return.
2. ***Processing HTTP client requests.*** The client sends an HTTP request for a Coralized URL to one of the returned proxies. If the proxy is caching the web object locally, it returns the object and the client is finished. Otherwise, the proxy attempts to find the object on another CoralCDN proxy.
3. ***Discovering cooperative-cached content.*** The proxy looks up the object's URL in the Coral indexing layer.
4. ***Retrieving content.*** If Coral returns the address of a node caching the object, the proxy fetches the object from this node. Otherwise, the proxy downloads the object from the origin server `example.com`.
5. ***Serving content to clients.*** The proxy stores the web object to disk and returns it to the client browser.
6. ***Announcing cached content.*** The proxy stores a reference to itself in Coral, recording the fact that is now caching the URL.

This section reviews the design of the Coral indexing layer and the CDN's proxies, as proposed in [14].

2.2 Coral indexing layer

The Coral indexing layer is closely related to the structure and organization of distributed hash tables like Chord [34] and Kademlia [23], with the latter serving as the basis for its underlying algorithm. The system maps opaque keys onto nodes by hashing their value onto a flat, semantic-free identifier (ID) space; nodes are assigned identifiers in the same ID space. It allows scalable key lookup (in $O(\log(n))$ overlay hops for n-node systems), reorganizes itself upon network membership changes, and provides robust behavior against failure.

Compared to "traditional" DHTs, Coral introduced a few novel techniques that were well-suited for its particular application [13]. Its key-value indexing layer was designed with weaker consistency requirements in mind, and its lookup structure self-organized into a locality-optimized hierarchy of clusters of peers. After all, a client need not discover all proxies caching a particular file, it only needs to find several such proxies, preferably ones nearby. Like most DHTs, Coral exposes *put* and *get* operations, to announce one's address as caching a web object, and to discover other proxies caching the object associated with a particular URL, respectively. Inserted addresses are soft-state mappings with a time-to-live (TTL) value.

Coral's *put* and *get* operations are designed to spread load, both within the DHT and across CoralCDN proxies. To *get* the proxy addresses associated with a key *k*, a node traverses the ID space with iterative RPCs, and it stops upon finding any remote peer storing values for *k*. This peer need not be the one closest to *k* (in terms of DHT identifier space distance). To *put* a key/value pair, Coral routes to nodes successively closer to *k* and stops when finding either (1) the nodes closest to *k* or (2) one that is experiencing high request rates for *k* and already is caching several corresponding values (with longer-lived TTLs). It stores the pair at the node closest to *k* that it managed to reach. These processes prevent tree saturation in the DHT.

To improve locality, these routing operations are not initially performed across the entire global overlay. Instead, each Coral node belongs to several distinct routing structures called *clusters*. Each cluster is characterized by a maximum desired network round-trip-time (RTT). The system is parameterized by a fixed hierarchy of clusters with different expected RTT thresholds. Coral's deployment uses a three-level hierarchy, with level-0 denoting the global cluster and level-2 the most local one. Coral employs distributed algorithms to form localized, stable clusters, which we briefly return to in §5.3.

Every node belongs to one cluster at each level, as in Figure 2. Coral queries nodes in fast clusters before those in slower clusters. This both reduces lookup latency and increases the chance of returning values stored at nearby nodes, which correspond to addresses of nearby proxies.

2.3 The CoralCDN HTTP proxy

CoralCDN seeks to aggressively minimize load on origin servers. This section summarizes how its proxies use Coral for inter-proxy cooperation and adaptation to flash crowds.

2.3.1 Locality-optimized inter-proxy transfers

Each CoralCDN proxy keeps a local cache from which it can immediately fulfill client requests. When a client requests a non-resident URL, CoralCDN proxies attempt to fetch web content from each other, using the Coral indexing layer for discovery. A proxy only contacts a URL's origin server after the Coral indexing layer provides no referrals or none of its referrals return the data.

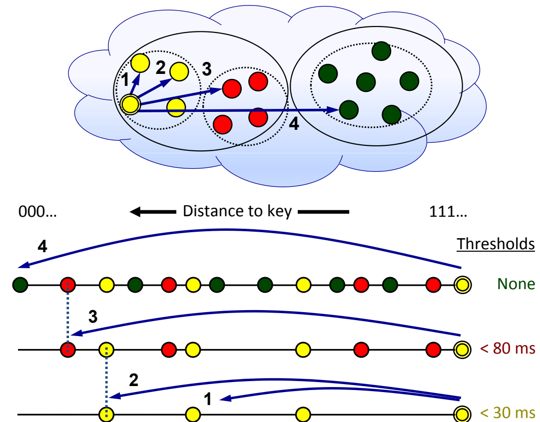


Figure 2: ***Coral's three-level hierarchical overlay structure.*** **A node first queries others in its level-2 cluster (the dotted rings), where pointers reference other caching proxies within the same cluster. If a node finds a mapping in its local cluster (after step 2), its *get* finishes. Otherwise, it continues among its level-1 cluster (the solid rings), and finally, if needed, to any node within the global level-0 system.**

CoralCDN's inter-proxy transfers are optimized for locality, both from their use of parallel connections to other proxies and by the order in which neighboring proxies are contacted. The properties of Coral's hierarchical indexing ensures that the list of proxies returned by *get* will be sorted based on their cluster distance to the request initiator. Thus, proxies will attempt to contact level-2 neighbors before level-1 and level-0 proxies, respectively.

2.3.2 Rapid adaptation to flash crowds

Unlike many web proxies, CoralCDN is explicitly designed for flash-crowd scenarios. If a flash crowd suddenly arrives for a web object, proxies self-organize into a form of multicast tree for retrieving the object. Data streams from the proxies that started to fetch the object from the origin server to those arriving later. This limits concurrent object requests to the origin server upon a flash crowd.

CoralCDN provides such behavior by cut-through routing and optimistic references. First, CoralCDN's use of *cut-through* routing at each proxy helps reduce transmission time for larger files. That is, a proxy will upload portions of a object as soon as they are downloaded, not waiting until it receives the entire object. Second, proxies optimistically announce themselves as sources of content. As soon as a CoralCDN proxy begins receiving the first bytes of a web object—either from the origin or another proxy—it inserts a reference to itself into Coral with a short TTL (30 seconds). It continually renews this short-lived reference until either it completes the download (at which time it inserts a longer-lived reference[1]) or the download fails.

[1]The deployed system uses 2-hour TTLs for successful results (status codes of 200, 301, 302, etc.), and 15-minute TTLs for 403, 404, and other unsuccessful, non-transient results.

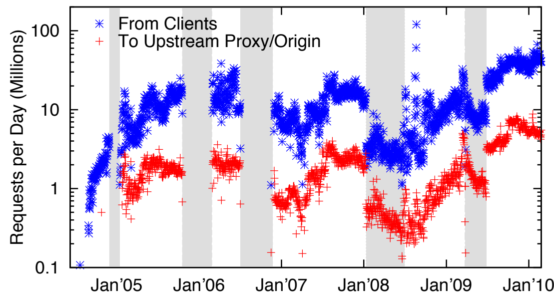


Figure 3: **Total HTTP requests per day during CoralCDN's deployment. Grayed regions correspond to missing or incomplete data.**

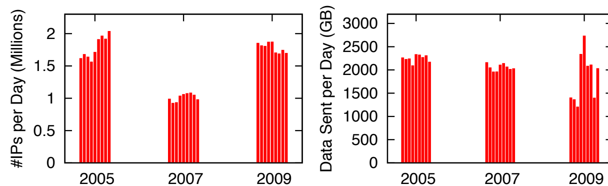


Figure 4: **CoralCDN usage: number of unique clients (left) and upload volume (right) for each day during August 9–18.**

2.4 Implementation and deployment

CoralCDN is composed of three stand-alone applications. The Coral daemon provides the distributed indexing layer, accessed over UNIX domain sockets from a simple client library linked into applications such as CoralCDN's HTTP proxy and DNS server. All three are written from scratch. Coral network communication uses Sun RPC over UDP, while CoralCDN proxies transfer content via standard HTTP connections. At initial publication [14], the Coral daemon was about 14,000 lines of C++, the DNS server 2,000 LOC, and the proxy 4,000 LOC. CoralCDN's implementation has since grown to around 50,000 LOC. The changes we later discuss help account for this increase.

CoralCDN typically runs on 300–400 PlanetLab servers (about 70–100 of which run its DNS server), spread over 100-200 sites worldwide. It avoids Internet2-only and commercial sites, the latter due to policy decisions that restrict their use for open services. CoralCDN uses no special knowledge of these machines' locations or connectivity (*e.g.*, GPS coordinates, routing information, etc.). Even though CoralCDN runs on a centrally-managed testbed, its mechanisms remain decentralized and self-organizing. The only use of centralization is for managing software and configuration updates and for controlling run status.

3 Analyzing CoralCDN's Usage

This section presents some HTTP-level data from CoralCDN's deployment and considers its implications.

3.1 System traces and traffic patterns

To understand some of the HTTP traffic patterns that CoralCDN sees, we analyzed several datasets in increasing

Year	Unique domains	Unique URLs	% URLs with 1 req	Reqs to most popular URL
2005	7881	577K	54%	697K
2007	21555	588K	59%	410K
2009	20680	1787K	77%	1578K

Figure 5: **CoralCDN traffic statistics for an arbitrary day (Aug 9).**

depth. Figure 3 plots the total number of HTTP requests that the system received each day from mid-2004 through early 2010, showing both the number of HTTP requests from clients, as well as the number of requests issued to upstream CoralCDN peers or origin sites. The traces show common request rates for much of CoralCDN's deployment between 5 and 20 million HTTP requests per day, with more recent rates of 40–50 million daily requests.[2]

We examined three time periods from these logs in more depth, each consisting of HTTP traffic over the same nine-day period (August 9–18) in 2005, 2007, and 2009. CoralCDN received 15–25M requests during each day of these periods. Figure 4 plots the total number of unique client IP addresses from which these requests originated (left) and the aggregate amount of bandwidth uploaded (right). The traces showed 1–2 million clients per day, resulting in a few terabytes of content transferred. We will primarily use the 2009 trace, consisting of 209M requests, in later analysis. Figure 5 provides more information about the traffic patterns, focusing on the first day of each trace.

Figure 6 plots the distribution of requests per unique URL. We see that the number of requests per URL follows a Zipf-like distribution, as common among web caching and proxy networks [5]. Certain URLs are very popular—the so-called "head" of the distribution—such as the most popular one in the Aug-9-2009 trace, which received almost 1.6M requests itself. A large number of URLs—the distribution's "heavy tail"—receive only a single request.

The datasets also show stability in the most popular URLs and domains over time. In all three datasets, the most popular URL retained that ranking across all nine days. In fact, this URL in the 2007 and 2009 traces belonged to the same domain: a site that uses CoralCDN to distribute rule-set updates for the popular Firefox AdBlock browser extension. Exploring this further, Figure 7 uses the 2009 trace to plot the request rate per day for the most popular domains (taking the union of each day's most popular five domains resulted in nine unique domains). We see that six of the nine domains had stable traffic patterns—they were long-term CoralCDN "customers"—while three varied between two and six orders of magnitude per day. The traffic patterns that we see in these two figures have design implications, which we discuss next.

[2]The peak of 120M requests on August 21, 2008 corresponds to a short-lived experiment of an academic research project using CoralCDN as a key-value store [15].

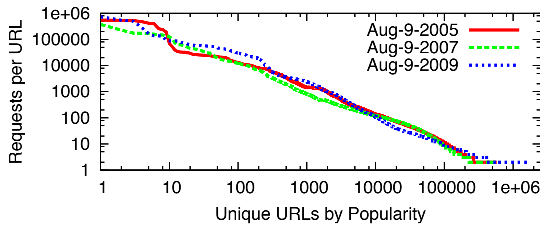


Figure 6: **Total requests per unique URL.**

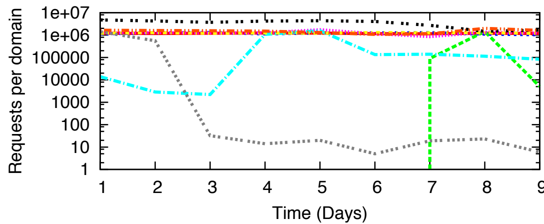


Figure 7: **Requests per top-5 domain over time (Aug 9-18, 2009).**

3.2 Implications of usage scenarios

For CoralCDN to help under-provisioned websites survive unexpected traffic spikes, it does not require any prior registration or authorization. Yet while such openness is necessary to enable even unmanaged websites to survive flash crowds, it comes at a cost: CoralCDN is used in a variety of ways that differ from this more narrow goal. This section considers how well CoralCDN's design is suited for its four main usage scenarios:

1. ***Resurrecting old content:*** Anecdotally, some clients attempt to use CoralCDN for long-term durability. One can download browser plugins that link to both CoralCDN and `archive.org` as potential sources of content when origin servers are unavailable.
2. ***Accessing unpopular content:*** CoralCDN's request distribution shows a heavy tail of unpopular URLs. Servers may Coralize URLs that few visit. And some clients use CoralCDN as a more traditional proxy, for (presumed) anonymity, censorship or filtering circumvention [32], or automated crawling.
3. ***Serving long-term popular content:*** Most requests are for a small set of popular objects. These objects, already widely cached across the network, belong to the stable set of customer domains that effectively use CoralCDN as a free, long-term CDN provider.
4. ***Surviving flash crowds to content:*** Finally, CoralCDN is used for its stated goal of enabling underprovisioned websites to withstand transient load spikes. Popular portals regularly link to Coralized URLs, and users post links in comments. Some sites even adopt dynamic and programmatic mechanisms to redirect requests to CoralCDN, based on observed load and request referrers. We discuss this further in §4.1.

Unfortunately, CoralCDN's design is not well-suited for the first three use cases.

Top URLs	Total Size (MB)	% of Total Reqs
0.01%	14	49.1%
0.1%	157	71.8%
1%	3744	84.8%
10%	28734	92.2%

Figure 8: **CoralCDN's working set size for its most popular URLs on Aug 9, 2009: A small percentage of URLs account for a large fraction of requests, yet they require relatively little storage to cache.**

Insufficient for resurrecting old content. CoralCDN is not designed for archival storage. Proxies do not proactively replicate content for durability, and unpopular content is evicted from proxy caches over time. Further, if content has an expiry time (default is 12 hours), a proxy will serve expired content for at most 24 hours after the origin fails. Still, some clients attempt to use CoralCDN for this purpose. This underscores a design tradeoff: In stressing support for flash crowds rather than long-term durability, CoralCDN devotes its resources to provide availability for content being actively requested. On the other hand, by serving expired content for a limited duration, CoralCDN can mask the *temporary* unavailability of an origin, at least for content already cached in its network.

Unnecessary for unpopular content. While proxies can discover even rare cached content, CoralCDN does not provide any benefit by serving such unpopular content: It does not reduce servers' load meaningfully, and it often results in higher client latency. As such, clients that use CoralCDN to avoid local filtering, circumvent geographic restrictions, or provide (minimal) anonymity may be better served by standard open proxies (that vanilla browsers can be configured to use) or through specialized tools such as Tor [12]. Yet, this type of usage persists—the long tail of Figure 6—and CoralCDN might then be better served with a different design for such traffic, *i.e.*, one that doesn't require a multi-hop, wide-area DHT lookup to complete before fetching content from the origin. For example, for its modest deployment on PlanetLab, each Coral node could maintain connectivity to all others and simply use consistent hashing for a global, one-hop DHT [17, 37]. Alternatively, Coral could only maintain connections with regional peers and eschew global lookups, a design which we evaluate further in §7.

Overkill for stably popular content, so far. For most of CoralCDN's traffic, cooperation is not needed: Figure 6 shows that a small number of URLs accounts for a large fraction of requests. We now measure their working set size in Figure 8, in order to determine how much storage is required to handle this traffic. We find that the most popular 0.01% of URLs account for more than 49% of the total requests to CoralCDN, yet require only 14 MB of storage. Each proxy has a 3.0 GB disk cache, managed using an LRU eviction policy. This is sufficient for serving nearly 85% of all requests from local cache.

70.4% hit in local cache	
12.6% returned 4xx or 5xx error code	
9.9% fetched from origin site	
7.1% fetched from other CoralCDN proxy	
|→ 1.7%	from level-0 cluster (global)
|→ 1.9%	from level-1 cluster (regional)
|→ 3.6%	from level-2 cluster (local)

Figure 9: **CoralCDN access ratios for content during Aug 9, 2009.**

These workload distributions support one aspect of CoralCDN's design: Content should be locally cached by the "forward" CoralCDN proxy directly serving end-clients, given that small to moderate size caches in these proxies can serve a very large fraction of requests. This differs from the traditional DHT approach of just storing data on a small number of globally-selected proxies, so-called "server surrogates" [8, 37].

If CoralCDN's working set can be fully cached by each node, we should understand how much cooperation is actually needed. Figure 9 summarizes the extent to which proxies cooperate when handling requests. 70% of requests to proxies are satisfied locally, while only 7% result in cooperative transfers. (The high rate of error messages is due to admission control as a means of bandwidth management, which we discuss in §5.2.) In short, at least for its current workload and environment, only a small fraction of CoralCDN's traffic uses its cooperation mechanisms.

A related result about the limits of cooperative caching had been observed earlier [38], but from the perspective of limited improvements in client-side hit rates. This is a significantly different goal from reducing server-side request rates, however: Non-cooperating groups of nodes would each individually request content from the origin.

This design trade-off comes down to the question of how much traffic is too much for origin servers. For moderately-provisioned origins, such as the customers of commercial CDNs, a caching system might only rely on local or regional cooperation. In fact, Akamai's network is designed precisely so: Nodes *within* each of its approximately 1000 clusters cooperate, but each cluster typically fetches content independently from origin sites [22]. To replicate such scenarios, Coral's clustering algorithms could be used to self-organize a network into local or regional clusters. It could thus avoid the manual configuration of Harvest [7] or colocated deployments of Akamai.

On the other hand, while cooperation is not needed for most traffic, CoralCDN's ability to react quickly to flash crowds—to offload traffic from a failing or oversubscribed origin—is precisely the scenario for which it was designed (and commercial CDNs are not). We consider these next.

Useful for mitigating flash crowds. CoralCDN's traces regularly show spikes in requests to different URLs. We find, however, that these flash crowds grow in popularity on the order of minutes, not seconds. There is a sufficiently long leading edge before traffic rises by several orders of magnitude, which has interesting implications.

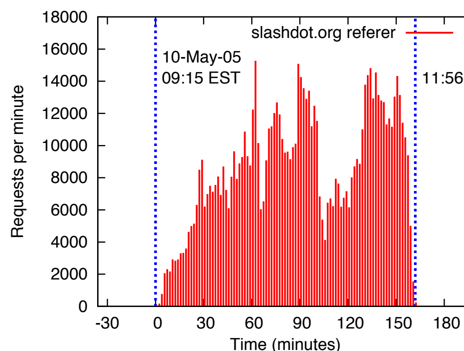


Figure 10: **Flash crowd to a Coralized URL linked to by Slashdot.**

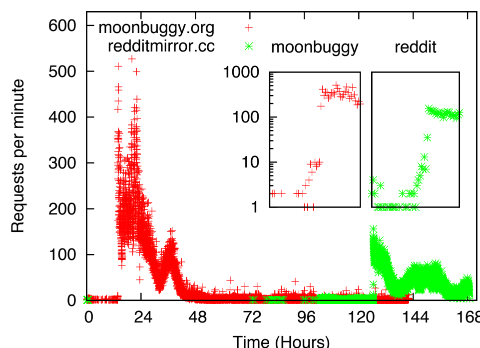


Figure 11: **Mini-flash crowds during August 2009 trace. Each datapoint represents a one-minute duration; embedded subfigures show request rates for the tens of minutes around the onset of flash crowds.**

Figures 10 and 11 show the request patterns of several flash crowds that CoralCDN experienced. The former was to a site linked to in a Slashdot article in May 2005. After rising, the Slashdot flash crowd lasted less than three hours in duration and came to an abrupt conclusion (perhaps as the story dropped off the website's main page). The latter, covering our August 2009 trace, shows spikes to the image cache of a less popular portal (`moonbuggy.org`), as well as to a well-publicized mirror for the collaboratively-filtered `reddit.com`, with another attenuated spike 24 hours later. The embedded graphs in Figure 11 depict the request rates around the onset of the traffic spike for a narrower range of time. All three flash crowds show that the initial rise took minutes.

For a more quantitative analysis of the frequency of flash crowds, we examined the prevalence of domains that experience a large increase in their request rates from one time period to the next. In particular, Figure 12 considers all five-second periods across the August 2009 ten-day trace. The left graph plots a complementary cumulative distribution function (CCDF) of the percentage of domains requested in each period that experience a 10- or 100-fold rate increase. The right graph plots the percentage of requests accounted for by these domains that experience orders-of-magnitude (OOM) increases. Sudden

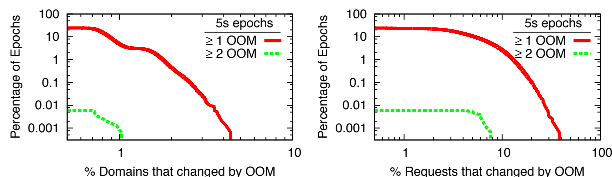


Figure 12: **CCDF of extent of flash-crowd dynamics in August 2009 trace. *Left* graph shows percentage of domains experiencing orders of magnitude (OOM) changes in request rates across five-second epochs. *Right* shows % requests for which these domains account.**

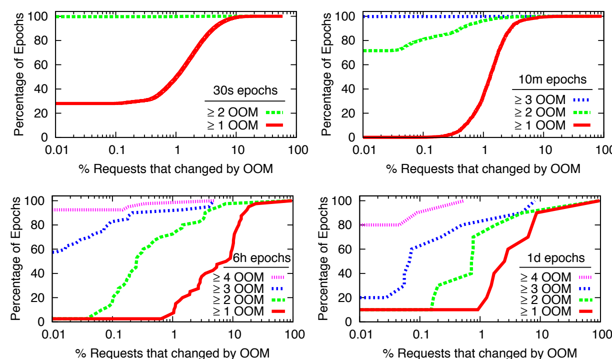


Figure 13: **CDFs of percentage of requests accounted for by domains experiencing order(s)-of-magnitude rate increases. Rate increases computed across epochs of 30 seconds (*top left*), 10 minutes (*top right*), six hours (*bottom left*), and one day (*bottom right*). Plots start on the y-axis with zero domains having such an increase, *e.g.*, 28% of 30s epochs have no domains with a ≥ 1 OOM rate increase.**

increases do exist, but they are rare. In 76% of 5s epochs, no domains experienced any 10-fold increase, while in 1% of epochs, 1.7% of domains (accounting for 12.9% of requests) increased by one order-of-magnitude. Larger dynamism was even more rare: only in 0.006% of epochs did there exist a domain that experienced a 100-fold increase in request rate. No three OOM increase occurred.

To further understand the precipitousness of "flash" crowds, Figure 13 extends this analysis across longer durations.[3] Among 30s epochs, 50% of epochs have at most 0.4% of domains experience a 10-fold increase in their rates (not shown), which account for a total of 1.0% of requests (top left). Only 0.29% of 30s epochs have *any* domains with more than a 100-fold rate increase. At 10-minute epochs, 28% of epochs have at least one domain that experiences a two OOM rate increase, while 0.21% have a domain with a three OOM increase. Still, these flash crowds account for a small fraction of total requests: Domains experiencing 100-fold increases accounted for at least 1% of all requests in only 3.8% of 10m epochs, and 10% of requests in 0.05% of epochs.

[3]To avoid overcounting unpopular domains, we do not count changes when the absolute number of requests to a domain in a given time period is less than some minimum amount, *i.e.*, 10 requests for 5s, 30s, and 10m periods, and 100 requests for 6h and 1d periods.

In short, this data shows that (1) only a small fraction of CoralCDN's domains experience large rate increases within short time periods, (2) those domains' traffic accounts for a small fraction of the total requests, and (3) any rate increases very rarely occur on the order of seconds.

This moderate adoption rate avoids the need to introduce even more aggressive content discovery algorithms. Simulated workloads in early experiments (Figure 4 of [14]) showed that under high concurrency, CoralCDN might issue several redundant fetches to an origin server due to a race-like condition in its lookup protocol. If multiple nodes concurrently *get* the same key which does not yet exist in the index, all concurrent lookups can fail and multiple nodes can contact the origin. This race condition is shared by most applications which use a distributed hash table (both peer-to-peer and datacenter services). But because these traces show that the arrival of user requests happens over a much longer time-scale than a DHT lookup, this race condition does not pose a significant problem.

Note that it is possible to mitigate this condition. While designing a network file system for PlanetLab that supported cooperative caching [2]—meant to quickly distribute a file in preparation for a new experiment—we sought to minimize redundant fetches to the file server. We extended Coral's insert operation to provide return status information, like test-and-set in shared-memory systems. A single *put+get* both returns the first values it encountered in the DHT, as well as inserts its own values at an appropriate location (for a new key, this would be at its closest node). This optimization comes at a subtle cost, however, as it now optimistically inserts a node's identity even before that proxy begins downloading the file! If the origin fetch fails—a greater possibility in CoralCDN's environment than with a managed file server—then the use of these index entries degrades performance. Thus, after using this *put+get* protocol in CoralCDN for several months during 2005, we discontinued its use.

CoralCDN's openness permits users to quickly leverage its resources under load, and its more complex coordination helps mitigate these flash crowds and mask temporary server unavailability. Yet this very openness led to varied usage, the majority of which does not require CoralCDN's more complex design. As we will see, this openness also introduces other problems.

4 Lessons for the Web

CoralCDN's naming technique provides an open API for CDN services that can transparently work for almost any website. Over the course of its deployment, clients and servers have used this API to adopt CoralCDN as an *elastic resource for content distribution.* Through completely automated means, work can be dynamically expanded out to use CoralCDN when websites require additional band-

width resources, and it can be contracted back when flash crowds abate. In doing so, its use presaged the notion of "surge computing" with public cloud platforms. But these naming techniques and CoralCDN's open design introduce a number of web security problems, many of which are engendered by a *lack of explicitness for specifying protection domains*. We discuss these issues here.

4.1 An API for elastic CDN services

We believe that the central reason for CoralCDN's adoption has been its simple user interface and open design.

Interface design. While superficially obvious, CoralCDN's interface design achieves several important goals:

- ***Transparency:*** Work with *unmodified*, *unconfigured*, and *unaware* web clients and webservers.
- ***Deep caching:*** Retrieve embedded images or links automatically through CoralCDN when appropriate.
- ***Server control:*** Not interfere with sites' ability to perform usage logging or otherwise control how their content is served (*e.g.*, via CoralCDN or directly).
- ***Ad-friendly:*** Not interfere with third-party advertising, analytics, or other tools incorporated into a site.
- ***Forward compatible:*** Be amenable to future end-to-end security mechanisms for content integrity or other end-host deployed mechanisms.

Consider an alternative and even simpler interface design [11, 25, 29], in which one embeds origin URLs into the HTTP path, *e.g.*, `http://nyud.net/example.com/`. Not only is HTTP parsing simpler, but nameservers would not need to synthesize DNS records on the fly (unlike our DNS servers for `*.nyud.net`). Unfortunately, while this interface can be used to distribute individual objects, it fails on entire webpages. Any relative links would lack the `example.com` prefix that a proxy needs to identify its origin. One alternative might be to try to rewrite pages to add such links, although active content such as javascript makes this notoriously difficult. Further, such active rewriting impedes a site's control over its content, and it can interfere with analytics and advertisements.

CoralCDN's approach, however, interprets relative links with respect to a page's Coralized hostname, and thus transparently requests these objects through it as well. But all absolute URLs continue to point to their origin sites, and third-party advertisements and analytics remain largely unaffected. Further, as CoralCDN does not modify content, content also may be amenable to verification through end-to-end content signatures [30, 35].

In short, it was important for adoption that *site owners retain sufficient control over how their content is displayed and accessed*. In fact, our predicted usage scenario of sites publishing Coralized URLs proved to be less popular than that of dynamic redirection (which we did not foresee).

An API for dynamic adoption. CoralCDN was envisioned with manual URL manipulation in mind, whether by publishers editing HTML, users typing Coralized URLs, or third-parties posting links. After deployment, however, users soon began treating CoralCDN's interface as an API for accessing CDN services.

On the client side, these techniques included simple browser extensions that offer "right-click" options to Coralize links or that provide a link when a page appears unavailable. They ranged to more complex integration into frameworks like Firefox's Greasemonkey [21]. Greasemonkey allows third-party developers to write site-specific javascript code that, once installed by users, manipulates a site's HTML content (usually through the DOM interface) whenever the user accesses it. Greasemonkey scripts for CoralCDN include those that automatically rewrite links on popular portals, or modify articles to include tooltips or additional links to Coralized URLs. CoralCDN also has been integrated directly into a number of client-side software packages for podcasting.

The more interesting cases of CoralCDN integration are on the server-side. One common strategy is for the origin to receive the initial request, but respond with a 302 redirect to a Coralized URL. This can work well even for flash crowds, as the overhead of generating redirects is modest compared to that of actually serving the content.

Generating such redirects can be done by installing a server plugin and writing a few lines of configuration code. For example, the complete dynamic redirection rule using Apache's `mod_rewrite` plugin is as follows.

```
RewriteEngine on
RewriteCond %{HTTP_USER_AGENT} !^CoralWebPrx
RewriteCond %{QUERY_STRING} !(^|&)coral-no-serve$
RewriteRule ^(.*)$ http://%{HTTP_HOST}.nyud.net
                             %{REQUEST_URI} [R,L]
```

Still, redirection rules must be crafted carefully. In this example, the second line checks whether the client is a CoralCDN proxy and thus should be served directly. Otherwise, a redirection loop potentially could be formed (although proxies prevent this from happening by checking for potential loops and returning errors if one is found).

Amusingly, some early users during CoralCDN's deployment caused recursion in a different way—and a form of amplification attack—by submitting URLs with a long string of `nyud.net`'s appended to a domain. Before proxies checked for such conditions, this single request caused a proxy to issue a number of requests, stripping the last instance of `nyud.net` off in each iteration.

While the above rewriting rule applies for all requests, other sites incorporate redirection in more inventive ways, such as only redirecting clients arriving from particular high-traffic referrers:

```
RewriteCond %{HTTP_REFERER} slashdot\.org [NC,OR]
RewriteCond %{HTTP_REFERER} digg\.com [NC,OR]
RewriteCond %{HTTP_REFERER} blogspot\.com [NC]
```

And most interestingly, some sites have even combined such tools with server plugins that monitor server load and bandwidth use, so that their servers only start rewriting requests *under high load conditions*.

Websites therefore used CoralCDN's naming technique to leverage its CDN resources in an elastic fashion. Based on feedback from users, we expanded this "API" to give sites some simple control over how CoralCDN should handle their requests. For example, webservers can include `X-Coral-Control` response headers, which are saved as cache meta-data, to specify whether CoralCDN proxies should "redirect home" domains that exceed their bandwidth limits (per §5.2) or just return an error as is standard.

4.2 Security and resource protection

A number of security mechanisms curtailed the misuse of CoralCDN. We highlight the design principle for each.

4.2.1 Limiting functionality

CoralCDN proxies have only ever supported GET and HEAD requests. Many of the attacks for which "open" proxies are infamous [24] are simply not feasible. For example, clients cannot use CoralCDN to POST passwords for brute-force cracking. Proxies do not support CONNECT requests, and thus they cannot be used to send spam as SMTP relays or to forge "From" addresses in web mail. Proxies do not support HTTPS and they delete all HTTP cookies sent in headers. These proxies thus provide *minimal application functionality* needed to achieve their goals, which is cooperatively serving cacheable content.

CoralCDN's design had several unexpected consequences. Perhaps most interestingly, given CoralCDN's multi-layer caching architecture, attempting to crawl or brute-force attack a website via CoralCDN is quite slow. New or randomly-selected URLs first require a DHT lookup to fail, which serves to delay requests against an origin website, in much the same way that `ssh` "tarpits" delay responses to failed login attempts. In addition, because CoralCDN only handles explicit Coralized URLs, it cannot be used by simply configuring a vanilla browser's proxy settings. Further, CoralCDN cannot be used to anonymously launch attacks, as it eschews anonymity. Proxies use unique `User-Agent` strings ("`CoralWebPrx`") and include their identity in `Via` headers, and they report an instigating client's IP address to the origin server (in an `X-Forwarded-For` request header). We can only surmise whether the combination of these properties played some role, but CoralCDN has seen little abuse as a platform for proxying server attacks.

4.2.2 Curtailing excessive resource use

CoralCDN's major limiting resource is aggregate bandwidth. The system employs fair-sharing mechanisms to balance bandwidth consumption between origin domains, which we discuss further in §5.2. In addition to monitoring server-side consumption, proxies keep a sliding window of client-side usage. Not only do we seek to prevent excessive bandwidth consumption by clients, but also an excessive number of (even small) requests. These are caused typically by server misconfigurations that result in HTTP redirection loops (per §4.1) or by "bot" misuse as part of a brute-force attack. While CoralCDN's limited functionality mitigates such attacks, one notable brute-force login attempt took advantage of poor security at a top-5 website, which used cleartext passwords over GET requests.

Given both its storage and bandwidth limitations, CoralCDN enforces a maximum file size of 50 MB. This has generally prevented clients from using CoralCDN for video distribution, a pragmatic goal when deploying proxies on university-hosted PlanetLab servers. We found that sites attempted to circumvent these limits by omitting `Content-Length` headers (on connections marked as persistent and without chunked encoding). To ensure compliance, proxies now monitor ongoing transfers and halt (and blacklist) any ones that exceed their limits. This skepticism is needed as proxies interact with potentially untrusted servers, and thus must enforce *complete mediation* [33] to their resources (in this case, bandwidth).

4.2.3 Blacklisting domains and offloading security

We maintain a global blacklist for blocking access to specified origin domain names. Each proxy regularly fetches and reloads the blacklist. This is a practical, but not fundamental, necessity, employed to prevent CoralCDN's deployment sites from restricting its use. Parties that request blacklisting typically cite one of the following reasons.

Suspected phishing. Websites have been concerned that CoralCDN is—or will be confused with—a phishing site. After all, both appear to be "scraping" content and publish a simulacrum under an alternate domain. The difference, of course, is that CoralCDN is serving the site's content unmodified, yet the web lacks any protocol to authenticate the integrity of content (as in S-HTTP [30]) in order to verify this. As SSL only authenticates identity, *websites must typically include CDNs in their trusted computing base.*

Potential copyright violation. Typically following a DMCA take-down notice, third-parties report that copyrighted material may be found on a Coralized domain and want it blocked. This scenario is mitigated by CoralCDN's explicit naming—which preserves the name of the actual origin in question—and by its caching design. Once content is removed from an origin server, it is evicted automatically from CoralCDN in at most 24 hours. This is a natural implication of its goal of handling flash crowds, rather than providing long-term availability.

Circumventing access-control restrictions. Some domains mediate access to their website via IP-based authen-

tication, whereby requests from particular IP prefixes are granted access. This practice is especially common for online academic journals, in order to provide easy access for university subscribers. But open proxies within whitelisted prefixes would enable external clients to circumvent these access-control restrictions.

By offloading policing to their customers, sites *unnecessarily enlarge their security perimeter to include their customer's networks*. This scenario is common yet unnecessary. Recall that CoralCDN proxies do not hide their identities, and they include the originating client's IP address in standard request headers. Thus, origin sites can retain IP-based authentication while verifying that a request does not originate from outside allowed prefixes.[4] Sites are just not making use of this information, and thus fail to properly mediate access to their protected resources.[5]

We did encounter some interesting attacks on our *domain*-based blacklists, akin to fast-flux networks. An adversary created dynamic DNS records for a random domain that pointed to the IP address of a target domain (an online academic journal). The random domain naturally was not blacklisted by CoralCDN, and the content was successfully downloaded from the origin target. Such a circumvention technique would not have worked if the origin site checked either proxy headers (as above) or even just the `Host` field of the HTTP request. The `Host` corresponded to the fast-flux attack domain, not that of the journal. Again, this security hole demonstrates a lack of explicit verification and fail-safe defaults [33].

4.3 Security and naming conflation

We argued that CoralCDN's naming provided a powerful API for accessing CDN services. Unfortunately, its technique has serious implications as the Web's Same Origin Policy (SOP) *conflates naming with security*.

Browsers use domain names for three purposes. (1) Domains specify *where* to retrieve content after they are resolved to IP addresses, precisely how CoralCDN enacts its layer of indirection. (2) Domains provide a human-readable name for *what administrative entity* a client is interacting with (*e.g.*, the "common name" identified in SSL server certificates). (3) Domains specify *what security policies* to enforce on web objects and their interactions.

The Same Origin Policy specifies how scripts and instructions from an origin domain can access and modify browser state. This policy applies to manipulating cookies, browser windows, frames, and documents, as well as to accessing other URLs via an XmlHttpRequest. At its simplest level, all of these behaviors are only allowed between resources that belong to an identical origin domain. This provides security against sites accessing each others' private information kept in cookies, for example. It also prevents websites that run advertisements (such as Google's AdSense) from easily performing click fraud to pay themselves advertising dollars by programmatically "clicking" on their site's advertisements.[6]

One caveat to the strict definition of an identical origin [18] is that it provides an exception for domains that share the same `domain.tld` suffix, in that `www.example.com` can read and set cookies for `example.com`. This has bad implications for CoralCDN's naming strategy. When `example.com` is accessed via CoralCDN, it can manipulate all `nyud.net` cookies, not just those restricted to `example.com.nyud.net`.[7] Concerned with the potential privacy violations from this scenario, CoralCDN deletes all cookies from headers.

Unfortunately, many websites now manage cookies via javascript, so cookie information can still "leak" between Coralized domains on the browser. This happens often without a site's knowledge, as sites commonly use a URL's `domain.tld` without verifying its name. Thus, if the Coralized `example.com` writes `nyud.net` cookies, these will be sent to `evil.com.nyud.net` if the client visits that webpage. Honest CoralCDN proxies will delete these cookies in transit, but attackers can still circumvent this problem. For example, when a client visits `evil.com.nyud.net`, javascript from that page can access `nyud.net` cookies, then issue a XmlHttpRequest back to `evil.com.nyud.net` with cookie information embedded in the URL. Similar attacks are possible against other uses of the SOP, especially as it relates to the ability to access and manipulate the DOM. Note that these attack vectors exist even while CoralCDN operates on fully-trusted nodes, let alone more peer-to-peer environments!

Rather than conclude that CoralCDN's domain manipulation is fundamentally flawed, we argue that better adherence to security principles is needed. Websites are partially at fault because they default access to `domain.tld` suffixes too readily, as opposed to stripping the minimal number of domain prefixes: a violation of the principle of least information. An alternative solution that embraces least

[4] This does not address the corner case in which the original request comes from an IP address within that prefix, while subsequent ones that access the then-cached content do not. This can be handled typically by marking content as not cacheable, or by having a proxy include headers that explicitly specify its client population (*i.e.*, as "open" or by IP prefix).

[5] One might argue that sites use a pure IP-based filtering approach given its ability to be implemented in layer-3 front-end load balancers. But this is not a simple firewall problem, as sites also permit access for individual users that login with the appropriate credentials. The sites with which we communicated implemented such authorization logic either directly in webservers or in complex, layer-7 front-end appliances.

[6] This is prevented because advertisements like AdSense load in an `iframe` that the parent document—the third-party website that stands to gain revenue—cannot access, as the frame belongs to a different domain.

[7] Commercial CDNs like Akamai are typically not susceptible to such attacks, as they generally use a separate top-level domains for each customer, as opposed to CoralCDN's suffix-based approach. Unlike CoralCDN's zero configuration, however, such designs require that origins preestablish an operational relationship with their CDN provider and point their domain to the CDN service (*e.g.*, by aliasing their domain to the CDN through CNAME records in DNS).

privilege (and has much better incremental deployability) would be to *allow sources of content to explicitly constrain default security policies*. As one simple example, when serving content for some `origin.tld`, proxies could use HTTP response headers to specify that the most permissive domain should be `origin.tld.domain.tld`, not their own `domain.tld`. Interestingly, HTML 5, Flash, and various javascript hacks [6] are all exploring methods to *expand* explicit cross-domain communication.[8] Both proposals avow that the SOP is insufficient and should be adapted to support more flexible control through explicit rules; ours just views its corner cases as too permissive, while the other views its implications as too restrictive.

5 Lessons for CDNs

Unlike most commercial counterparts, CoralCDN is designed to interact with overloaded or poorly-behaving origin servers. Further, while commercial systems will grow their networks based on expected use (and hence revenue), the CoralCDN deployment is comprised of volunteer sites with fixed, limited bandwidth. This section describes how we adapted CoralCDN to satisfy these realities.

5.1 Designing for faulty origins

Given its design goals, CoralCDN needs to react to non-crash failures at origin servers as the rule, not the exception. Thus, one design philosophy that has come to govern CoralCDN's behavior is that *proxies should accept content conservatively and serve results liberally*.

Consider the following, fairly common, situation. A portal runs a story that links to a third-party website, driving a sudden influx of readers to this previously unpopular site. A user then posts a Coralized link to the third-party site as a "comment" to the portal's story, providing an alternate means to fetch the content.

Several scenarios are possible. (1) The website's origin server becomes unavailable before any proxy downloads its content. (2) CoralCDN already has a copy of the content, but requests arrive to it after the content's *expiry* time has passed. Unfortunately, subsequent HTTP requests to the origin webserver result in failures or errors. (3) Or, CoralCDN's content is again expired, but subsequent requests to the origin yield only partial transfers. CoralCDN employs different mechanisms to handle these failures.

Cache negative service results (#1). CoralCDN may be hit with a flood of requests for an inaccessible URL, *e.g.*, DNS resolution fails, TCP connections timeout, etc. For these situations, proxies maintain a local negative result cache about repeated failures. Otherwise, both proxies and their local DNS resolvers have experienced resource exhaustion, given flash crowds to apparently dead sites. (While negative result caching has also long been part of some DNS implementations [19], it is not universal and does not extend to TCP or application-level failures.) While more a usability issue, CoralCDN still receives requests for some Coralized URLs several years after their origins became unavailable.

Serve stale content if origin faulty (#2). CoralCDN seeks to avoid replacing good content with bad. As its proxies mostly obey content expiry times specified in HTTP headers,[9] if cached content expires, proxies perform a conditional request (`If-Modified-Since`) to revalidate or update expired content. Overloaded origin servers might fail to respond or might return some temporary error condition (data in §7 shows this to occur in about 0.5% of origin requests). Rather than retransmit this error, CoralCDN proxies return the stale content and continue to retain it for future use (for up to 24 hours after it expires).

Prevent truncations through whole-file overwrites (#3). Rather than not responding or returning an error, what if a revalidation yields a truncated transfer? This is not uncommon during a flash crowd, as a CoralCDN proxy will be competing for a webserver's resources. Rather than have proxies lose stale yet complete versions of objects, proxies implement *whole-file overwrites* in the spirit of AFS [16]. Namely, if a valid web object is already cached, the new version is written to a temporary file. Only after the new version completes downloading and appears valid (based on `Content-Length`) will a proxy replace the old one.

These approaches are not fail-proof, limited by both semantic ambiguity in status directives and inaccuracies with their use. In terms of ambiguity, does a 403 (Forbidden) response code signify that a publisher seeks to make the content unavailable (permanent), or is it caused by a website surpassing its daily bandwidth limits and having requests rejected (temporary)? Does a 404 (File Not Found) code indicate whether the condition is permanent (due to a DMCA take-down notice) or temporary (from a PHP or database error)? On the other hand, the application of status directives can be flawed. We often found websites to report human-readable errors in HTML body content, but with an HTTP status code of 200 (Success). This scenario leads CoralCDN to replace valid content with less useful information. We hypothesize that bad defaults in scripting languages such as PHP are partially to blame. Instead of being fail-safe, the response code defaults to success.

Even if transient errors were properly identified, for how long should CoralCDN serve expired content? HTTP lacks

[8] This is in reaction to the common practice of inserting third-party objects into a document's namespace via `<script>`—and thus sacrificing security protections—as the SOP does not permit a middle ground.

[9] Proxies in our deployment are configured with a *minimum* expiry time of some duration (five minutes), and thus do not recognize `No-Cache` directives as such. Because CoralCDN does not support cookies, SSL bridging, or POSTs, however, many of the privacy concerns associated with caching such content are alleviated.

the ability to specify explicit policy for handling expired content. Akamai defaults to a fail-safe scenario by not returning stale content [22], while CoralCDN seeks to balance this goal with availability under server failures. As opposed to only using the system-wide default of 24 hours, CoralCDN recently enabled its users to explicitly specify their policy through `max-stale` response headers.[10]

These examples all point to another lesson that governs CoralCDN's proxy design: *Maintain the status quo unless improvements are possible.*

Decoupling service dependencies. A similar theme of only improving the status quo governs CoralCDN's management system. CoralCDN servers query a centralized management point for a number of tasks: to update their overall run status, to start or stop individual service components (HTTP, DNS, DHT), to reinstall or update to a new software version, or to learn shared secrets that provide admission control to its DHT. Although designed for intermittent connectivity, one of CoralCDN's significant outages came when the management server began misbehaving and returning unexpected information. In response, we adopted what one might call *fail-same behavior* that accepts updates conservatively, an application of decoupling techniques from fault-tolerant systems. Management information is stored durably on servers, maintaining their status-quo operation (even across local crashes) until well-formed new instructions are received.

5.2 Managing oversubscribed bandwidth

While commercial CDNs and computing platforms often respond to oversubscription by acquiring more capacity, CoralCDN's deployment on PlanetLab does not have that luxury. Instead, the service must manage its bandwidth consumption within prescribed limits. This adoption of bandwidth limits was spurred on by administrative demands from its deployment sites. Following the Asian tsunami of December 2004, and with YouTube yet to be created, CoralCDN distributed large quantities of amateur videos of the natural disaster. With no bandwidth restrictions on PlanetLab at the time, CoralCDN's network traffic to the public Internet quickly spiked. PlanetLab sites threatened to pull their servers off the network if such use could not be curtailed. It was agreed that CoralCDN should restrict its usage to approximately 10 GB per day per server (*i.e.*, per PlanetLab sliver).

Several design options exist for limiting bandwidth consumption. A proxy could simply shut down after exceeding a configured daily capacity (as supported by Tor [12]). Or it could rate-limit its traffic to prevent transient congestion (as done by BitTorrent and Tor). But as CoralCDN

[10]HTTP/1.1 supports max-stale *request* headers, although we are not aware of their use by any HTTP clients. Further, as proxies often evict expired content from their caches, it is unclear whether such request directives can be typically satisfied.

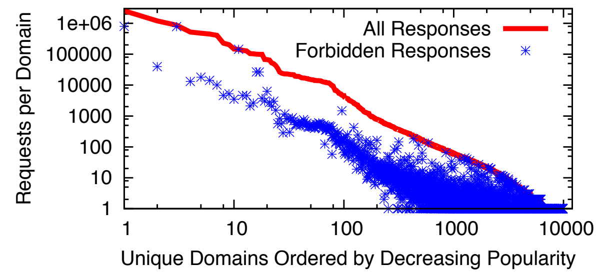


Figure 14: **Requests per domain and number of 403 rejections.**

primarily provides a service for websites, as opposed to clients, we chose to allocate its limited bandwidth in a way that both preserves some notion of *fairness* across its customer domains and maintains its central goal of handling flash crowds. The technique we developed is more broadly applicable than just PlanetLab and federated testbeds: to P2P deployments where users run peers within resource containers, to multi-tenant datacenters sharing resources between their own services, or to commercial hosting environments using billing models such as 95th-%ile usage.

Providing per-domain fairness might be resource intensive or difficult in the general case, given that CoralCDN interacts with 10,000s of domains each day, but our highly-skewed workloads greatly simplify the necessary accounting. Figure 14 shows the total number of requests per domain that CoralCDN received over one day (the solid top line). The distribution clearly has some very popular domains—the most popular one (a Tamil clone of YouTube) received 2.6M requests—while the remaining distribution fell off in a Zipf-like manner. (Note that Figure 6 was in terms of unique URLs, not unique domains.) Given that CoralCDN's traffic is dominated by a limited number of domains, its mechanisms can serve mainly to reject requests for (*i.e.*, perform admission control on) these bandwidth hogs. Still, CoralCDN should differentiate between peak limits and steady-state behavior to allow for flash crowds or changing traffic patterns.

To achieve these aims, each CoralCDN proxy implements an algorithm that attempts to simultaneously (1) provide a hard-upper limit on peak traffic per hour (configured to 1000 MB per hour per proxy), (2) bound the expected total traffic per epoch in steady state (400 MB per hour per proxy), and (3) bound the steady-state limit per domain. As setting this last limit statically—such as $1/k$-th of the total traffic if there are k popular domains—would lead to good fairness but poor utilization (given the non-uniform distribution across domains), we dynamically adjust this last traffic limit to balance this trade-off.

During each hour-long epoch, a proxy records the total number of bytes transmitted for each domain. It also calculates domains' average bandwidth as an exponentially-weighted moving average (attenuated over one week), as well as the total average consumption across all domains. This long attenuation period provides long-term fairness—

and most consumption is long-term, as shown in Figure 7—but also emphasizes support for short-term flash crowds. Across epochs, bandwidth usage is only tracked, and durably stored, for the top-100 domains. If a domain is not currently one of the top-100 bandwidth consumers, its historical average bandwidth is set to zero (providing additional leeway to sites experiencing flash crowds).

When a requested domain is over its hourly budget (case 3 above), CoralCDN proxies respond with 403 (Forbidden) messages. If instead the proxy is over its peak or steady-state limit calculated over all domains (cases 1 or 2 above), then the proxy redirects the client back to the origin site, and the proxy temporarily makes itself unavailable for new client requests, which would be rejected anyway.[11]

By applying these mechanisms, CoralCDN reduces its bandwidth consumption to manageable levels. While its demand sometimes exceeds 10 TBs per day (aggregate across all proxies), its actual HTTP traffic remains steady at about 2 TB per day after rejecting a significant number of requests. The scatter plot in Figure 14 shows the number of requests resulting in 403 responses per domain, most due to these admission control mechanisms. We see how variances in domains' object sizes yield different rejection rates. The second-most popular domain serves mostly images smaller than 10 KB and experiences a rejection rate of 3.3%. Yet the videos of the third-most popular domain—user-contributed screensavers of fractal flames—are typically 5 MB in size, leading to an 89% rejection rate.

Note that we could significantly curtail the use of CoralCDN as a long-term CDN provider (see §3.2) through simple changes to these configuration settings. A low steady-state limit per domain, coupled with a greater weight on a domain's historic averages, devotes resources to flash-crowd relief at the exclusion of long-term consumption.

Admittedly, CoralCDN's approach penalizes an origin site with more regional access patterns. Bandwidth accounting and admission control is performed independently on each node, reflecting CoralCDN's lack of centralization. By not sharing information between nodes (provided that DNS resolution preserves locality), a site with regional interest can be throttled before it reaches its fair share of global capacity. While this does not pose an operational problem for CoralCDN, it is an interesting research problem to perform (approximate) accounting across the network that is both decentralized and scalable. Distributed Rate Limiting [28] considered a related problem, but focused on instantaneous limits (*e.g.*, Mbps) instead of long-term aggregate volumes and gossiped state that is linear in both the number of domains and nodes.

[11] If clients are redirected back to the origin, a proxy appends the query-string `coral-no-serve` on the location URL returned to the client. Origins that use redirection scripts with CoralCDN check for this string to prevent loops, per §4.1. Although not the default, operators of some sites preferred this redirection home even if their domain was to blame (a policy they can specify through a `X-Coral-Control` response header).

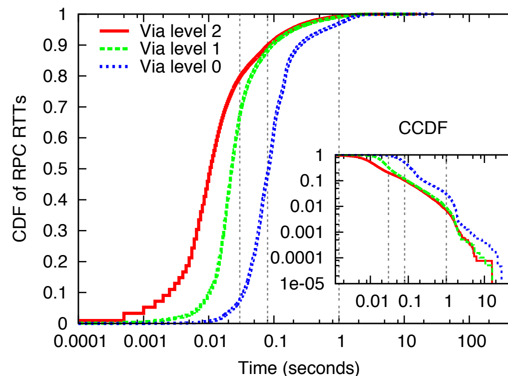


Figure 15: **RPC RTTs to various levels of Coral's DHT hierarchy.**

5.3 Managing performance jitter

Running on an oversubscribed deployment platform, CoralCDN developed several techniques to better handle latency variations. With PlanetLab services facing high disk, memory, and CPU contention, and sometimes additional traffic shaping in the kernel, applications can face both performance jitter and prolonged delays. These performance variations are not unique to PlanetLab, and they have been well documented across a variety of settings. For example, Google's MapReduce [10] took runtime adaption of cluster query processing [3] to the large-scale, where performance variations even among homogeneous components required speculative re-execution of work. More recently, studies of a MapReduce clone on Amazon's EC2 underscored how shared and virtualized platforms provide new performance challenges [39].

CoralCDN saw the implications of performance variations most strikingly with its latency-sensitive self-organization. For example, Coral's DHT hierarchy was based on nodes clustering by network RTTs. A node would join a cluster provided some minimum fraction (85%) of its members were below the specified threshold (30 ms for level 2, 80 ms for level 1). Figure 15 shows the RTTs for RPC between Coral nodes, broken down by levels (with vertical lines added at 30ms, 80ms, and 1s). While the clustering algorithms achieve their goals and local clusters have lower RTTs, the heavy tail in all CDFs is rather striking. Fully 1% of RPCs took longer than 1 second, even within local clusters. Coral's use of concurrent RPCs during DHT operations helped mask this effect.

Another lesson from CoralCDN's deployment was the need for *stability in the face of performance variations*. This translated to the following rule in Coral. A node would switch to a smaller (and hence less attractive) cluster only if fewer than 70% of a cluster's members now satisfy its threshold, and form a singleton only if fewer than 50% of neighbors are satisfactory. In other words, the barrier to enter a cluster is high (85%), but once a member, it's easier to remain. Before leveraging this form of hysteresis, cluster oscillations were much more common, which led

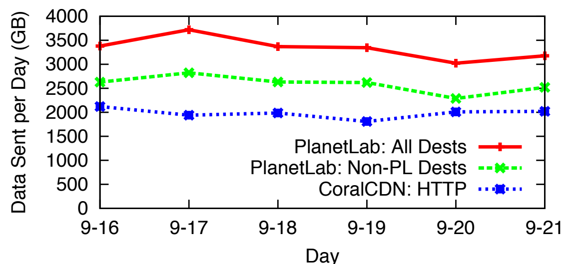


Figure 16: **Comparison of PlanetLab's accounting of all upstream traffic, PlanetLab's count to non-PlanetLab destinations, and CoralCDN's accounting through HTTP logs.**

to many stale DHT references. A related use of hysteresis within self-organizing systems helped improve virtual network coordinate systems for both PlanetLab [26] and Azureus [20], as well as failure recovery in Bamboo [31].

6 Lessons for Platforms

With the growth of virtualized hosting and cloud deployments, Internet services are increasingly running on third-party infrastructure. Motivated by CoralCDN's deployment on PlanetLab, we discuss some benefits from improving an application's visibility into and control over its lower layers. We first revisit CoralCDN's bandwidth management from the perspective of fine-grained service differentiation, then describe tackling its fault-tolerance challenge with adequate network support.

6.1 Exposing information and expressing preferences across layers

We described CoralCDN's bandwidth management as self-regulating, which works well in trusted environments. But many resource providers would rather *enforce* restrictions than assume applications behave well. Indeed, in 2006, PlanetLab began enforcing average daily bandwidth limits per node per service (*i.e.*, per PlanetLab "sliver"). When a sliver hits 80% of its limit—17.2 GB/day from each server to the public Internet—the kernel begins enforcing bandwidth caps (using Linux's Hierarchical Token Bucket scheduler) as calculated over five-minute epochs.

We now have the possibility of two levels of bandwidth management: admission control by CoralCDN proxies and rate limiting by the underlying hosting platform. Interestingly, even though CoralCDN uses a relatively conservative limit for itself (10 GB/day per sliver), it still surpasses the 80% mark (13.8 GB) on 5–10 servers per day (out of its 300-400 servers). The main cause of this overage is that, while CoralCDN counts only successful HTTP responses, its hosting platform accounts for all traffic—HTTP, DNS, DHT RPCs, log transfers, packet headers, retransmissions, etc.—generated by its sliver. Figure 16 shows the difference in these recorded values for the week of Sept 16, 2009. We see that kernel statistics were 50%-90% higher than CoralCDN's accounting. This problem of accurate accounting is a general one, as it is difficult or expensive to collect such data in user-space.[12] And even accurate information does not prevent CoralCDN's managed HTTP traffic from competing for network resources with the rest of its sliver's unmanaged traffic.

We argue that hosting platforms should provide better visibility and control. First, these platforms should export greater information to higher levels, such as their current measured resource consumption in a machine-readable format and in real time. Second, these platforms should allow applications to push policies into lower levels, *i.e.*, an application's explicit preferences for handling different classes of resources. For the specific case of network resources, the platform kernel could apply priorities on a granularity finer that just per-sliver, akin to a form of end-host DiffServ; CoralCDN would prioritize DNS and DHT traffic over HTTP traffic, in turn over log maintenance.

Note that we are concerned with a different type of resource management than that provided by VM hypervisors or kernel resource containers [4]. Those systems focus on *short-term* resource isolation or prioritized scheduling between applications, and typically reason about *coarse-grain* VM-level resources. Our focus instead is on *long-term* resource accounting. PlanetLab is not unique here; commercial cloud-computing providers such as Amazon and Rackspace use long-term resource accounting for billing purposes. (In fact, Amazon just launched its CloudWatch service in June 2009 to expose real-time resource monitoring on a coarser-grain per-VM basis [1].) Thus, providing greater visibility and control would be useful not only for deploying applications on platforms with hard constraints (*e.g.*, PlanetLab), but also for managing applications on commercial platforms so as to minimize costs (*e.g.*, in both metered and 95th-%ile billing scenarios).

6.2 Providing support for fault-tolerance

A central reliability issue in CoralCDN is due to its bootstrapping problem: To initially resolve a Coralized URL with no prior knowledge of system participants, a client's resolver must contact one of only 10–12 CoralCDN nameservers registered with the `.net` gTLD servers. If one of these nameservers fails—each IP address represents a static PlanetLab server—clients experience long DNS timeouts. Thus, while CoralCDN *internally* detects and reacts quickly to failure, the same rapid recovery is not enjoyed by its primary nameservers registered *externally*. And once legacy clients bind to a particular proxy's IP address—*e.g.*, web browsers cache name-to-IP mapping to prevent certain types of "rebinding" attacks on the

[12] In fact, even Akamai servers only use an estimate of bandwidth consumption (their so-called "fully-weighted bits") when calculating server load [22]. Only more recently did PlanetLab expose kernel accounting.

Same Origin Policy [9]—CoralCDN cannot recover for this client if that proxy fails.

While certainly observed before, CoralCDN's reliability challenge underscores the limits of purely application-layer recovery, especially as it relates to bootstrapping. In the context of DNS-based bootstrapping, several possibilities exist, including (1) dynamically updating root nameservers to reflect changes, *e.g.*, via the rarely-supported RFC2136 [36], (2) announcing IP anycast addresses via BGP or OSPF, or (3) using transparent network-layer failover between colocated nameservers (*e.g.*, ARP spoofing or VIP/DIP load balancers). IP-level recovery between proxies has its own solutions, but most commonly rely on colocated servers in LAN environments. None of these suggestions are new ones, but they still present a higher barrier to entry; PlanetLab did not have any available to it.

Deployment platforms should strive to provide or expose such network functionality to their services. Amazon EC2's launch of Elastic IP Addresses in March 2008, for example, hid the complexity of ARP spoofing for VM environments. The further development of such support should be an explicit goal for future deployment platforms.

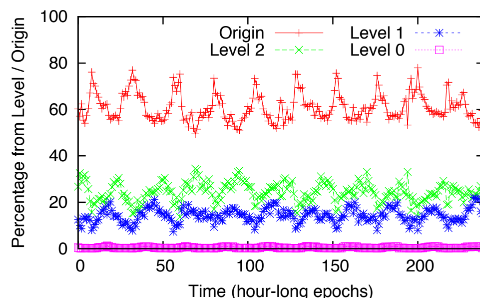


Figure 17: **Percentage of a proxy's upstream requests satisfied by origin and by peers at various clustering levels when *regional cooperation* is used, *i.e.*, level-0 peers only serve as a failover from a faulty origin. Dataset covers 10-day period from December 9–19, 2009.**

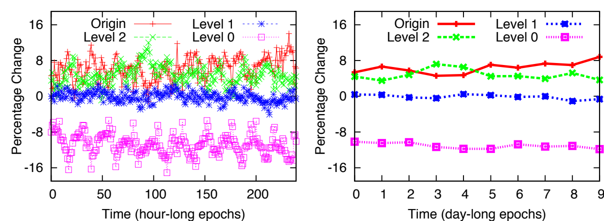


Figure 18: **Change in percentage between regional cooperation policy (Figure 17) and CoralCDN's traditional global peering. Positive values correspond to increased hit rates in regional peering.**

7 Conclusions and Looking Forward

Our retrospective on CoralCDN's deployment has a rather mixed message. We view the adoption of CoralCDN as a successful proof-of-concept of how users can and will leverage open APIs for CDN services. But many of its architectural features were over-designed for its current environment and with its current workload: A much simpler design could have sufficed with probably better performance to boot.

That said, it is a entirely different question as to whether CoralCDN provides a good basis for designing an Internet-scale cooperative CDN. The service remained tied to PlanetLab because we desired a solution that was backwards compatible with both unmodified clients and servers. Running on untrusted nodes seemed imprudent at best given our inability to provide end-to-end security checks. We have shown, however, that even running CoralCDN on fully trusted nodes introduces some security concerns. So, if we dropped the goal of full backwards compatibility, what minimal changes could better support more open, flexible infrastructure?

Naming. CoralCDN's naming provided a layer of indirection for composing two loosely-coupled Internet services. In fact, one could compose longer series of services that each offer different functionality by simply chaining together their domain names. While this technique would not be safe under today's Same Origin Policy, we showed in §4.3 how a trusted proxy could constrain the default security policy. For a participating origin server with an untrusted CDN, the origin should specify (and sign) its minimally required domain suffix of `origin.tld.*`.

Content Integrity. Today's CDNs are full-fledged members of a website's trusted computing base. They have free reign to return modified content. Often, they can even programmatically read and modify any content served *directly* from a customer website to its clients (either by serving embedded `<script>`'s or by playing SOP tricks while masquerading as their customer behind a DNS alias). To provide content delivery via untrusted nodes, the natural solution is an HTTP protocol that supports end-to-end signatures for content integrity [30]. In fact, even a browser extension would suffice to deploy such security [35].

Fine-Grain Origin Control. A tension in this paper is between client latency and server load, underscored by our varied usage scenarios. An appropriate strategy for interacting with a well-provisioned server is a minimal attempt at cooperation before contacting the origin. Yet, an oversubscribed server wants its clients to make a maximal effort at cooperation. So far, proxies have used a "one-size-fits-all" approach, treating all origins as if they were oversubscribed. Instead, much as they have adopted dynamic URL rewriting, origin domains can signal a CoralCDN proxy about their desired policy in-band. At a high-level, this argues for a richer API for elastic CDN services.

To explore the effect of *regional cooperation*, we changed the default lookup policy on about half the deployed CoralCDN proxies since September 2009. If re-

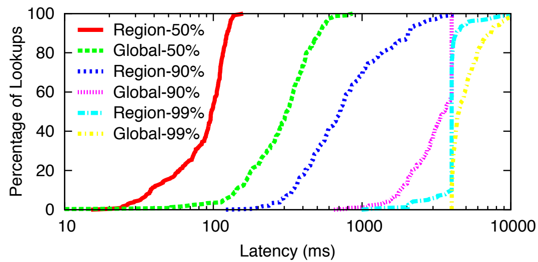


Figure 19: **CDF of median, 90th percentile, and 99th percentile lookup latency (over all hour-long epochs of Dec 9–19, 2009), comparing regional and global cooperation policies. Individual lookups were configured with a five-second timeout.**

quested content is not already cached locally, these proxies only perform lookups within local and regional clusters (level 2 and 1) before contacting the origin. For proxies operating under such a policy, Figure 17 shows the percentage of upstream requests that were satisfied by the origin and at different levels of clusters. Figure 18 depicts the *change* in behavior compared to the traditional global lookup strategy, showing that the 10–12% of requests that had been satisfied by level-0 proxies shifted to higher hit rates at both the origin and local proxies.[13] This change was associated with an order-of-magnitude latency improvement for the Coral lookup, shown in Figure 19. The global index still provides some benefit to the system, however, as per Figure 17, it satisfies an average of 0.56% of requests (stddev 0.51%) that failed over from origin servers. In summary, system architectures like CoralCDN can support different policies that trade-off server load for latency, yet still mask temporary failures at origins.

While perhaps imperfectly suited for a smaller-scale platform like PlanetLab, CoralCDN's architecture provides interesting self-organizational and hierarchical properties. This paper discussed many of the challenges—in security, availability, fault-tolerance, robustness, and, perhaps most significantly, resource management—that we needed to address during its five-year deployment. We believe that its lessons may have wider and more lasting implications for other systems as well.

Acknowledgments. We are grateful to David Mazières for his significant contributions and support during the design and operation of CoralCDN. We also thank Larry Peterson and the entire PlanetLab team for providing a deployment platform for CoralCDN. CoralCDN was originally funded as part of Project IRIS (supported by the NSF under Coop. Agreement #ANI-0225660) and recently under NSF Award #0904860. Freedman was also supported by an NDSEG Fellowship. More information about CoralCDN can be found at `www.coralcdn.org`.

[13]These graphs also show interesting diurnal patterns, related to a default expiry time of 12 hours for content.

References

[1] Amazon CloudWatch. http://aws.amazon.com/cloudwatch/, 2009.

[2] S. Annapureddy, M. J. Freedman, and D. Mazières. Shark: Scaling file servers via cooperative caching. In *NSDI*, 2005.

[3] R. H. Arpaci-Dusseau. Run-time adaptation in river. *ACM Trans. Computer Systems*, 21(1), 2003.

[4] G. Banga, P. Druschel, and J. Mogul. Resource containers: A new facility for resource management in server systems. In *OSDI*, 1999.

[5] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *INFOCOM*, 1999.

[6] J. Burke. Cross domain frame communication with fragment identifiers. http://tagneto.blogspot.com/, June 6, 2006.

[7] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, and K. Worrell. A hierarchical Internet object cache. In *USENIX Annual*, 1996.

[8] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *SOSP*, 2001.

[9] D. Dean, E. W. Felten, and D. S. Wallach. Java security: from hotjava to netscape and beyond. In *Symp. Security and Privacy*, 1996.

[10] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[11] Dijjer. http://code.google.com/p/dijjer/, 2010.

[12] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *USENIX Security*, 2004.

[13] M. J. Freedman and D. Mazières. Sloppy hashing and self-organizing clusters. In *IPTPS*, 2003.

[14] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In *NSDI*, 2004.

[15] E. Freudenthal, D. Herrera, S. Gutstein, R. Spring, and L. Longpre. Fern: An updatable authenticated dictionary suitable for distributed caching. In *MMM-ACNS*, 2007.

[16] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Trans. Computer Systems*, 6(1):51–81, 1988.

[17] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *STOC*, 1997.

[18] D. Kristol and L. Montulli. RFC 2965: HTTP state management mechanism, 2000.

[19] A. Kumar, J. Postel, C. Neuman, P. Danzig, and S. Miller. RFC 1536: Common DNS errors and suggested fixes, 1993.

[20] J. Ledlie, P. Gardner, and M. Seltzer. Network coordinates in the wild. In *NSDI*, 2007.

[21] A. Lieuallen, A. Boodman, and J. Sundstrom. Greasemonkey. https://addons.mozilla.org/en-US/firefox/addon/748, 2010.

[22] B. Maggs. Personal communication, 2009.

[23] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *IPTPS*, 2002.

[24] V. Pai, L. Wang, K. Park, R. Pang, and L. Peterson. The dark side of the web: An open proxy's view. In *HotNets*, 2003.

[25] K. Park and V. S. Pai. Scale and performance in the CoBlitz large-file distribution service. In *NSDI*, 2006.

[26] P. Pietzuch, J. Ledlie, and M. Seltzer. Supporting network coordinates on planetlab. In *WORLDS*, 2005.

[27] PlanetLab. http://www.planet-lab.org/, 2010.

[28] B. Raghavan, K. Vishwanath, S. Ramabhadran, K. Yocum, and A. Snoeren. Cloud control with distributed rate limiting. In *SIGCOMM*, 2007.

[29] RedSwoosh. http://www.akamai.com/redswoosh, 2009.

[30] E. Rescorla and A. Schiffman. RFC 2660: The secure hypertext transfer protocol, 1999.

[31] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *USENIX Annual*, 2004.

[32] H. Roberts, E. Zuckerman, and J. Palfrey. 2007 circumvention landscape report: Methods, uses, and tools. Technical report, Berkman Center for Internet & Society, Harvard, 2009.

[33] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proc. IEEE*, 93(9), 1975.

[34] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Trans. Network.*, 11(1):17–32, 2003.

[35] J. Terrace, H. Laidlaw, H. E. Liu, S. Stern, and M. J. Freedman. Bringing P2P to the Web: Security and privacy in the Firecoral network. In *IPTPS*, 2009.

[36] P. Vixie, S. Thomson, Y. Rekhter, and J. Bound. RFC 2136: Dynamic Updates in the Domain Name System, 1997.

[37] L. Wang, V. Pai, and L. Peterson. The effectiveness of request redirection on CDN robustness. In *OSDI*, Dec 2002.

[38] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. Levy. On the scale and performance of cooperative web proxy caching. In *SOSP*, 1999.

[39] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving map-reduce performance in heterogeneous environments. In *OSDI*, 2008.

Whānau: A Sybil-proof Distributed Hash Table

Chris Lesniewski-Laas M. Frans Kaashoek
MIT CSAIL, Cambridge, MA, USA
{ctl,kaashoek}@mit.edu

Abstract

Whānau is a novel routing protocol for distributed hash tables (DHTs) that is efficient and strongly resistant to the Sybil attack. Whānau uses the social connections between users to build routing tables that enable Sybil-resistant lookups. The number of Sybils in the social network does not affect the protocol's performance, but links between honest users and Sybils do. When there are n well-connected honest nodes, Whānau can tolerate up to $O(n/\log n)$ such "attack edges". This means that an adversary must convince a large fraction of the honest users to make a social connection with the adversary's Sybils before any lookups will fail.

Whānau uses ideas from structured DHTs to build routing tables that contain $O(\sqrt{n}\log n)$ entries per node. It introduces the idea of *layered identifiers* to counter clustering attacks, a class of Sybil attacks challenging for previous DHTs to handle. Using the constructed tables, lookups provably take constant time. Simulation results, using social network graphs from LiveJournal, Flickr, YouTube, and DBLP, confirm the analytic results. Experimental results on PlanetLab confirm that the protocol can handle modest churn.

1 Introduction

Decentralized systems on the Internet are vulnerable to the "Sybil attack", in which an adversary creates numerous false identities to influence the system's behavior [9]. This problem is particularly pernicious when the system is responsible for routing messages amongst nodes, as in the Distributed Hash Tables (DHT) [24] which underlie many peer-to-peer systems, because an attacker can prevent honest nodes from communicating altogether [23].

If a central authority certifies identities as genuine, then standard replication techniques can be used to fortify these protocols [4, 20]. However, the cost of universal strong identities may be prohibitive or impractical. Instead, recent work [27, 26, 8, 19, 17, 5] proposes using the weak identity information inherent in a social network to produce a completely decentralized system. This paper resolves an open problem by demonstrating an efficient, structured DHT that enables honest nodes to reliably communicate despite a concerted Sybil attack.

To solve this problem, we build on a social network composed of individual trust relations between honest people (nodes). This network might come from personal or business connections, or it might correspond to something more abstract, such as ISP peering relationships. We presume that each participant keeps track of its immediate neighbors, but that there is no central trusted node storing a map of the network.

An adversary can infiltrate the network by creating many *Sybil nodes* (phoney identities) and gaining the trust of honest people. Nodes cannot directly distinguish Sybil identities from genuine ones (if they could, it would be simple to reject Sybils). As in previous work [27], we assume that most honest nodes have more social connections to other honest nodes than to Sybils; in other words, the network has a *sparse cut* between the honest nodes and the Sybil nodes.

In the context of a DHT, the adversary cannot prevent immediate neighbors from communicating, but can try to disrupt the DHT by creating many Sybil identities which spread misinformation. Suppose an honest node u wants to find a key y and will recognize the corresponding value (e.g., a signed data block). In a typical structured DHT, u queries another node which u believes to be "closer" to y, which forwards to another even-closer node, and so on until y is found. The Sybil nodes can disrupt this process by spreading false information (e.g., that they are close to a particular key), then intercepting honest nodes' routing queries, and responding with "no such key" or delaying queries endlessly. Unstructured protocols that work by flooding or gossip are more robust against these attacks, but pay a performance price, requiring linear time to find a key.

This paper's main contribution is **Whānau** [1], a novel protocol that is the first solution to Sybil-proof routing that has sublinear run time and space usage. Whānau achieves this performance by combining the idea of random walks from recent work [26] with a new way of constructing IDs, which we call *layered identifiers*. To store up to k keys per node, Whānau builds routing tables with $O(\sqrt{kn}\log n)$ entries per node. Using these routing tables, lookups *provably* take $O(1)$ time. Thus, Whānau's security comes at low cost: it scales similarly to one-hop DHTs that provide no security [11, 10]. We have implemented Whānau in simulation and in a simple

[1] *Whānau*, pronounced "far-no", is a Māori word. It is cognate with the Hawai'ian word *'ohana*, and means "extended family" or "kin".

instant-messaging application running on PlanetLab [2]. Experiments with real-world social graphs and these implementations confirm Whānau's theoretical properties.

Whānau provides one-hop lookups, but our implementation is not aware of network locality. Whānau also must rebuild its routing tables periodically to handle churn in the social network and in the set of keys stored in the DHT. However, its routing tables are sufficiently redundant that nodes simply going up and down doesn't impact lookups, as long as enough honest nodes remain online.

The rest of the paper is organized as follows. Section 2 summarizes the related work. Section 3 informally states our goals. Section 4 states our assumptions about the social network, and provides a precise definition of "Sybil-proof". Section 5 gives an overview of Whānau's routing table structure and introduces layered IDs. Section 6 describes Whānau's setup and lookup procedures in detail. Section 7 states lemmas proving Whānau's good performance. Section 8 describes Whānau's implementation on PlanetLab [2] and in a simulator. Using these implementations Section 9 confirms its theoretical properties by simulations on social network graphs from popular Internet services, and investigates its reaction to churn on PlanetLab. Section 10 discusses engineering details and ideas for future work, and Section 11 summarizes.

2 Related work

Shortly after the introduction of scalable peer-to-peer systems based on DHTs, the Sybil attack was recognized as a challenging security problem [9, 16, 23, 22]. A number of techniques [4, 20, 22] have been proposed to make DHTs resistant to a small fraction of Sybil nodes, but all such systems ultimately rely on a certifying authority to perform admission control and limit the number of Sybil identities [9, 21, 3].

Several researchers [17, 19, 8, 5] proposed using social network information to fortify peer-to-peer systems against the Sybil attack. The *bootstrap graph* model [8] introduced a correctness criterion for secure routing using a social network and presented preliminary progress towards that goal, but left a robust and efficient protocol as an open problem.

Recently, SybilGuard and SybilLimit [27, 26] have shown how to use a "fast mixing" social network and random walks on these networks (see Section 4.1) to defend against the Sybil attack in general decentralized systems. Using SybilLimit, an honest node can certify other nodes as "probably honest", accepting at most $O(\log n)$ Sybil identities per attack edge. (Each certification uses $O(\sqrt{n})$ bandwidth.) For example, SybilLimit's vetting procedure can be used to check that at least one of a set of storage replicas is likely to be honest.

A few papers have adapted the idea of random walks for purposes other than SybilLimit. Nguyen *et al.* used it for Sybil-resilient content rating [25], Yu *et al.* applied it to recommendations [28], and Danezis and Mittal used it for Bayesian inference of Sybils [7]. This paper is the first to use random walks to build a Sybil-proof DHT. [2]

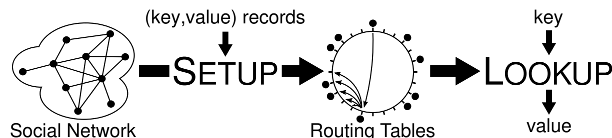


Figure 1: Overview of Whānau. SETUP builds structured routing tables which LOOKUP uses to route queries to keys.

3 Goals

As illustrated in Figure 1, the Whānau protocol is a pair of procedures SETUP and LOOKUP. SETUP$(\cdot)$ is used both to build routing tables and to insert keys. It cooperatively transforms the nodes' local parameters (e.g. key-value records, social neighbors) into a set of routing table structures stored at each node. After all nodes complete the SETUP phase, any node u can call LOOKUP(u, key) to use these routing tables to find the target $value$.

3.1 Scenario

We illustrate Whānau with a simple instant messaging (IM) application which we have implemented on PlanetLab. Whānau provides a rendezvous service for the IM clients. Each user is identified by a public key, and publishes a single self-signed tuple *(public key, current IP address)* into the DHT. [3] To send an IM to a buddy identified by the public key PK, a client looks up PK in the DHT, verifies the returned tuple's signature using PK, and then sends a packet to that IP address.

In our implementation, each user runs a Whānau node which stores that user's record, maintains contact with the user's social neighbors, and contributes to the DHT. (In this example, each node stores a single key-value record, but in general, there may be an arbitrary number k of keys stored per node.) When the user changes location, the client updates the user's record with the new IP address. The user's DHT node need not be continuously available when the user is offline, as long as a substantial fraction of honest nodes are available at any given time.

3.2 Security goal

Whānau handles adversaries who deviate from the protocol in Byzantine ways: the adversaries may make up arbitrary responses to queries from honest nodes and may create any number of pseudonyms (Sybils) which are indistinguishable from honest nodes. When we say that

[2] Our workshop paper [14] noted the opportunity and sketched an early precursor to Whānau.

[3] Of course, a realistic application would require a PKI for human-readable names, protection against replays, privacy controls, and so on.

Whānau is "Sybil-proof", we mean that LOOKUP has a high probability of returning the correct value, despite arbitrary attacks during both the SETUP and LOOKUP phases. (Section 4 makes this definition more precise.)

The adversary can always join the DHT normally and insert arbitrary key-value pairs, including a different value for a key already in the DHT. Thus, Whānau provides availability, but not integrity: LOOKUP should find all values inserted by honest nodes for the specified key, but may also return some values inserted by the adversary. Integrity is an orthogonal concern of the application: for example, the IM application filters out any bad values by verifying the signature on the returned key-value records, and ignoring records with invalid signatures. (As an optimization, DHT nodes may opt to discard bad records proactively, since they are of no use to any client and consume resources to store and transmit.)

3.3 Performance goals

Simply flooding LOOKUP queries over all links of the social network is Sybil-resistant, but not efficient [8]. The adversary's nodes might refuse to forward queries, or they might reply with bogus values. However, if there exists any path of honest nodes between the source node and the target key's node through the social network, then the adversary cannot prevent each of these nodes from forwarding the query to the next. In this way, the query will always reach the target node, which will reply with the correct value. Unfortunately, a large fraction of the participating nodes are contacted for every lookup, doing $O(n)$ work each time.

On the other hand, known one-hop DHTs are very efficient — requiring $O(1)$ messages for lookups and $O(\sqrt{n})$ table sizes[4]— but not secure against the Sybil attack. Our goal is to combine this optimal efficiency with provable security. As a matter of policy and fairness, we believe that a node's table size and bandwidth consumption should be proportional to the node's degree (i.e., highly connected nodes should do more work than casual participants). While it is possible to adapt Whānau to different policies, this paper assumes that the goal is a proportional policy.

4 Defining "Sybil-proof"

Like previous work [27, 26], Whānau relies on certain features of social networks. This section describes our assumptions, outlines why they are useful, and defines what it means for a DHT to be "Sybil-proof" under these assumptions.

[4] If $n = 5 \times 10^8$, the approximate number of Internet hosts in 2010, then a table of $\sqrt{n}$ may be acceptable for bandwidth and storage constrained devices, as opposed to a table that scales linearly with the number of hosts.

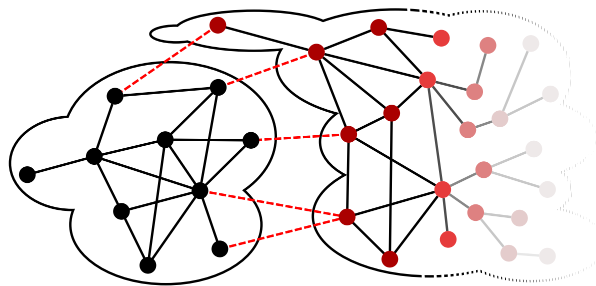


Figure 2: The social network. A sparse cut (the dashed attack edges) separates the honest nodes from the Sybil nodes. The Sybil region's size is not well-defined, since the adversary can create new pseudonyms at will.

4.1 Fast-mixing social networks

The **social network** is an undirected graph whose nodes know their immediate neighbors. Figure 2 conceptually divides the social network into two parts, an **honest region** containing all honest nodes and a **Sybil region** containing all Sybil identities. An **attack edge** is a connection between a Sybil node and an honest node. An **honest edge** is a connection between two honest nodes [27]. An "honest" node whose software's integrity has been compromised by the adversary is considered a Sybil node.

The key assumption is that the number of attack edges, g, is small relative to the number of honest nodes, n. As pointed out by earlier work, one can justify this **sparse cut** assumption by observing that, unlike creating a Sybil identity, creating an attack edge requires the adversary to expend social-engineering effort: the adversary must convince an honest person to create a social link to one of his Sybil identities.

Whānau's correctness will depend on the sparse cut assumption, but its performance will not depend *at all* on the number of Sybils. In fact, the protocol is totally oblivious to the structure of the Sybil region. Therefore, the classic Sybil attack, of creating many fake identities to swamp the honest identities, is ineffective.

Since we rely on the existence of a sparse cut to distinguish the honest region from the Sybil region, we also assume that there is *no* sparse cut dividing the honest region in two. Given this assumption, the honest region forms an expander graph. Expander graphs are **fast mixing**, which means that a short random walk starting from any node will quickly approach the stationary distribution [6]. Roughly speaking, the ending node of a random walk is a random node in the network, with a probability distribution proportional to the node's degree. The **mixing time**, w, is the number of steps a random walk must take to reach this smooth distribution. For a fast mixing network, $w = O(\log n)$. Section 9.1 shows that graphs extracted from some real social networks are fast mixing.

	Typical magnitude	Description
n	arbitrary $n \geq 1$	number of honest nodes
m	$O(n)$	number of honest edges
w	$O(\log n)$	mixing time of honest region
k	arbitrary $k \geq 1/m$	keys stored per (virtual) node
g	$O(n/w)$	number of attack edges
ϵ	$O(gw/n)$	fraction of loser nodes

Table 1: Social network parameters used in our analysis.

4.2 Sampling by random walk

The random walk is Whānau's main building block, and is the only way the protocol uses the social network. An honest node can send out a w-step walk to sample a random node from the social network. If it sends out a large number of such walks, and the social network is fast mixing and has a sparse cut separating the honest nodes and Sybil nodes, then the resulting set will contain a large fraction of random honest nodes and a small number of Sybil nodes [26]. Because the initiating node cannot tell which individual samples are good and which are bad, Whānau treats all sampled nodes equally, relying only on the fact that a large fraction will be good nodes.

Some honest nodes may be near a concentration of attack edges. Such **loser nodes** have been lax about ensuring that their social connections are real people, and their view of the social graph does not contain much information. Random walks starting from loser nodes are more likely to escape into the Sybil region. As a consequence, loser nodes must do more work per lookup than winner nodes, since the adversary can force them to waste resources. Luckily, only a small fraction of honest nodes are losers, because a higher concentration of attack edges in one part of the network means a lower concentration elsewhere. Most honest nodes will be **winner nodes**.

In the stationary distribution, proportionally more random walks will land on high-degree nodes than low-degree nodes. To handle high-degree nodes well, each Whānau participant creates one virtual node [24] per social network edge. Thus, good random samples are distributed uniformly over the virtual nodes. All virtual nodes contribute equal resources to the DHT and obtain equal levels of service (i.e., keys stored/queried). This use of virtual nodes fulfils the policy goal (Section 3.3) of allocating both workload and trust according to each person's level of participation in the social network.

4.3 Main security definition

Table 1 summarizes the social network parameters introduced thus far. We can now succinctly define our main security property:

Definition. A DHT protocol is (g, ϵ, p)**-Sybil-proof** if, against an active adversary with up to g attack edges, the protocol's LOOKUP procedure succeeds with probability $\geq p$ on any honest key, for at least $(1-\epsilon)n$ honest nodes.

Given a $(g, \epsilon, 1/2)$-Sybil-proof protocol, it is always possible to amplify the probability of success p exponentially close to 1 by, for example, running multiple independent instances of the protocol in parallel. [5] For example, running $3\log_2 n$ instances would reduce the failure probability to less than $1/n^3$, essentially guaranteeing that all lookups will succeed with high probability (since there are only n^2 possible source-target node pairs).

The parameter ϵ represents the fraction of loser nodes, which is a function of the distribution of attack edges in the network. If attack edges are distributed uniformly, then ϵ may be zero; if attack edges are clustered, then a small fraction of nodes may be losers.

We use the parameters in Table 1 to analyze our protocol, but do not assume that all of them are known by the honest participants. Whānau needs order-of-magnitude estimates of m, w, and k to choose appropriate table sizes and walk lengths. It does not need to know g or ϵ.

Proving that a protocol is Sybil-proof doesn't imply that it cannot be broken. For example, Whānau is Sybil-proof but can be broken by social engineering attacks that invalidate the assumption that there is a sparse cut between the honest and Sybil regions. Similarly, a protocol may be broken by using cryptographic attacks or attacks on the underlying network infrastructure. These are serious concerns, but these are not the Sybil attack as described by Douceur [9]. Whānau's novel contribution is that it is the first DHT protocol totally insensitive to the number of Sybil identities.

5 Overview of Whānau

This section outlines Whānau's main characteristics.

5.1 Challenge

The Sybil attack poses three main challenges for a structured DHT. First, structured DHTs forward queries using small routing tables at each node. Simply by creating many cheap pseudonyms, an attacker will occupy many of these table entries and can disrupt queries [23].

Second, a new DHT node builds and maintains its routing tables by querying its neighbors' tables. An attacker can reply to these queries with only its own nodes. Over time, this increases the fraction of table entries the attacker occupies [22].

Third, DHTs assign random IDs to nodes and apply hash functions to keys in order to spread load evenly. By applying repeated guess-and-check, a Sybil attacker can choose its own IDs and bypass these mechanisms. This enables **clustering attacks** targeted at a specific key. For example, if the adversary inserts many keys near the targeted key, then it might overflow the tables of honest nodes responsible for storing that part of the key space.

[5] For Whānau, it turns out to be more efficient to increase the routing table size instead of running multiple parallel instances.

Alternatively, the adversary might choose all its IDs to fall near the targeted key. Then, honest nodes might have to send many useless query messages to Sybil nodes before eventually querying an honest node.

5.2 Strawman protocol

To illustrate how random walks apply to the problem of Sybil-proof DHT routing, consider the following strawman protocol. In the setup phase, each node initiates $r = O(\sqrt{km})$ independent length-w random walks on the social network. It collects a random key-value record from the final node of each walk, and stores these nodes and records in a local table.

To perform a lookup, a node u consults its local record table. If the key is not in this table (which is likely), u broadcasts the key to the $O(\sqrt{km})$ nodes $v_1, \ldots, v_r$ in its table. If those nodes' tables are sufficiently large, with high probability, at least one node v_i will have the needed key-value record in its local table.

The strawman protocol shows how random walks address the first and second challenges above. If the number of attack edges is small, most random walks stay within the honest region. Thus, the local tables contain mostly honest nodes and records. Furthermore, nodes use only random walks to build their tables: they never look at each other's tables during the setup process. As a result, the adversary's influence does not increase over time.

The strawman sidesteps the third challenge by eschewing node IDs entirely, but this limits its efficiency. Lookups are "one-hop" in the sense that the ideal lookup latency is a single network round-trip. However, since each lookup sends a large number of messages, performance will become limited by network bandwidth and CPU load as the network size scales up. By adding structure, we can improve performance. The main challenge is to craft the structure in such a way that it cannot be exploited by a clustering attack.

5.3 Whānau's global structure

Whānau's structure resembles other DHTs such as Chord [24], SkipNet [12], and Kelips [11]. Like SkipNet and Chord, Whānau assumes a given, global, circular ordering $\prec$ on keys (e.g., lexical ordering). The notation $x_1 \prec x_2 \prec \cdots \prec x_z$ means that for any indexes $i < j < k$, the key x_j is on the arc (x_i, x_k).

No metric space. Like SkipNet, but unlike Chord and many other DHTs, Whānau does *not* embed the keys into a metric space using a hash function. If Whānau were to use a hash function to map keys into a metric space, an adversary could use guess-and-check to construct many keys that fall between any two neighboring honest keys. This would warp the distribution of keys in the system and defeat the purpose of the hash function. Therefore, Whānau has no *a priori* notion of "distance" between two keys; it can determine only if one key falls between two other keys. This simple ordering provides some structure (e.g., a node can have a successor table), but still requires defenses to clustering attacks.

Fingers and successors. Most structured DHTs have routing tables with both "far pointers", sometimes called *fingers*, and "near pointers", called *leaves* or *successors*. Whānau follows this pattern. All nodes have **layered IDs** (described below) which are of the same data type as the keys. Each node's **finger table** contains $O(\sqrt{km})$ pointers to other nodes with IDs spaced evenly over the key space. Likewise, each node's **successor table** contains the $O(\sqrt{km})$ honest key-value records immediately following its ID. Finger tables are constructed simply by sending out $O(\sqrt{km})$ random walks, collecting a random sample of (honest and Sybil) nodes along with their layered IDs. Successor tables are built using a more complex sampling procedure (described in Section 6.1).

Together, an honest node's finger nodes' successor tables cover the entire set of honest keys, with high probability. This structure enables fast one-hop lookups: simply send a query message to a finger node preceding the target key. The chosen finger is likely to have the needed record in its successor table. (If not, a few retries with different fingers should suffice.) In contrast with the strawman protocol above, this approach uses a constant number of messages on average, and $O(\log n)$ messages (which may be sent in parallel) in the worst case.

Layered IDs. Whānau defends against clustering attacks using **layers**, illustrated in Figure 3. Each node uses a random walk to choose a random key as its layer-0 ID. This ensures that honest nodes' layer-0 IDs are distributed evenly over the keys stored by the system.

To pick a layer-1 ID, each node picks a random entry from its own layer-0 finger table and uses that node's ID. To pick a layer-2 ID, each node takes a random layer-1 finger's ID, and so on for each of the $\ell = O(\log km)$ layers.

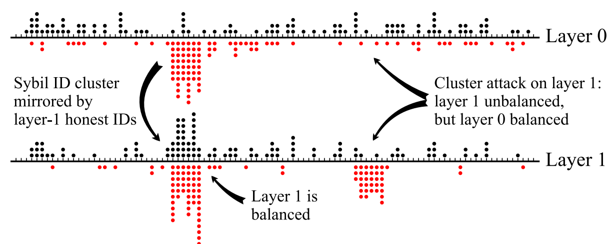


Figure 3: Honest IDs (black dots) in layer 0 are uniformly distributed over the set of keys (X axis), while Sybil IDs (red dots) may cluster arbitrarily. Honest nodes choose their layer $i + 1$ IDs from the set of all layer i IDs (honest and Sybil). Thus, most layers are balanced. Even if there is a clustering attack on a key, it will always be easy to find an honest finger near the key using a random sampling procedure.

ers. In the end, each node is present at, and must collect successor tables for, ℓ positions in the key space.

Layers defend against key clustering and ID clustering attacks. If the attacker inserts many keys near a target key, this will simply cause more honest nodes to choose layer-0 IDs in that range. The number of keys the attacker can insert is limited by the number of attack edges. Thus, a key clustering attack only shifts around the honest nodes' IDs without creating any hot or cold spots.

Nodes choose their own IDs; thus, if the attacker chooses all its layer-0 IDs to fall immediately before a target key, it might later be difficult to find an honest finger near the key. However, if the adversary manages to supply an honest node u with many clustered layer-0 IDs, then this increases the probability that u will pick one of these clustered IDs as its own layer-1 ID. As a result, the distribution of honest layer-1 IDs tends to mimic any clusters in the Sybil layer-0 IDs. This increases the honest nodes' presence in the adversary-chosen range, and increases the likelihood that layer 1 finger tables are balanced between honest and Sybil nodes.

The same pattern of balance holds for layers 2 through ℓ−1. As long as most layers have a good ratio of honest to Sybil IDs in every range, random sampling (as described in Section 6.2) can find honest fingers near any target key.

5.4 Churn

There are three sources of churn that Whānau must handle. First, computers may become temporarily unavailable due to network failures, mobility, overload, crashes, or being turned off daily. We call this **node churn**. Whānau builds substantial redundancy into its routing tables to handle Sybil attacks, and this same redundancy is sufficient to handle temporary node failures. Section 9.5 shows that increasing node churn results in a modest additional overhead.

The second source of churn is changes to the social relationships between participants. This **social churn** results in adding or deleting social connections. A single deleted link doesn't impact Whānau's performance, as long as the graph remains fast mixing and neither endpoint became malicious. (If one did become malicious, it would be treated as an attack edge.) However, Whānau doesn't immediately react to social churn, and can only incorporate added links by rebuilding its routing tables. Nodes which leave the DHT entirely are not immediately replaced. Therefore, until SETUP is invoked, the routing tables, load distribution, and so on will slowly become less reflective of the current social network, and performance will slowly degrade.

Social network churn occurs on a longer time scale than node churn. For example, data from Mislove *et al.* indicates that the Flickr social network's half-life is approximately 6 weeks [18]. Running SETUP every day, or every few minutes, would keep Whānau closely in sync with the current social graph.

The final and most challenging source of churn is changes to the set of keys stored in the DHT. This **key churn** causes the distribution of keys to drift out of sync with the distribution of finger IDs. Reacting immediately to key additions and deletions can create "hot spots" in successor tables; this can only be repaired by re-running SETUP. Thus, in the worst case, newly stored keys will not become available until the tables are rebuilt. For some applications — like the IM example, in which each node only ever stores one key — this is not a problem as long as tables are refreshed daily. Other applications may have application-specific solutions.

Unlike key churn, turnover of values does not present a challenge for Whānau: updates to the value associated with a key may always be immediately visible. For example, in the IM application, a public key's current IP address can be changed at any time by the record's owner. Value updates are not a problem because Whānau does not use the value fields when building its routing tables.

Key churn presents a trade-off between the bandwidth consumed by rebuilding tables periodically and the delay from a key being inserted to the key becoming visible. This bandwidth usage is similar to stabilization in other DHTs; however, insecure DHTs can make inserted keys visible immediately, since they do not worry about clustering attacks. We hope to improve Whānau's responsiveness to key churn; we outline one idea in Section 10.

6 The Whānau protocol

This section defines SETUP and LOOKUP in detail.

6.1 Setup

The SETUP procedure takes each node's social connections and the local key-value records to store as inputs, and constructs four routing tables:

- $ids(u, i)$: u's layer-i ID, a random key x.
- $fingers(u, i)$: u's layer-i fingers as $(id, address)$ pairs.
- $succ(u, i)$: u's layer-i successor $(key, value)$ records.
- $db(u)$: a sample of records used to construct $succ$.

The global parameters r_f, r_s, r_d, and ℓ determine the sizes of these tables; SETUP also takes an estimate of the mixing time w as a parameter. Typically, nodes will have a fixed bandwidth and storage budget to allocate amongst the tables. Section 7 and Section 9 will show how varying these parameters impacts Whānau's performance.

The SETUP pseudocode (Figure 4) constructs the routing tables in ℓ+1 phases. The first phase sends out r_d random walks to collect a sample of the records in the social network and stores them in the db table. These samples are used to build the other tables. The db table has the good property that each honest node's stored records are frequently represented in other honest nodes' db tables.

```
SETUP(stored-records(·), neighbors(·); w, r_d, r_f, r_s, ℓ)
1  for each node u
2      do db(u) ← SAMPLE-RECORDS(u, r_d)
3  for i ← 0 to ℓ − 1
4      do for each node u
5             do ids(u, i)      ← CHOOSE-ID(u, i)
6                fingers(u, i)  ← FINGERS(u, i, r_f)
7                succ(u, i)     ← SUCCESSORS(u, i, r_s)
8  return fingers, succ

SAMPLE-RECORDS(u, r_d)
1  for j ← 1 to r_d
2      do v_j ← RANDOM-WALK(u)
3         (key_j, value_j) ← SAMPLE-RECORD(v_j)
4  return {(key_1, value_1), . . . , (key_{r_d}, value_{r_d})}

SAMPLE-RECORD(u)
1  (key, value) ← RANDOM-CHOICE(stored-records(u))
2  return (key, value)

RANDOM-WALK(u_0)
1  for j ← 1 to w
2      do u_j ← RANDOM-CHOICE(neighbors(u_{j−1}))
3  return u_w

CHOOSE-ID(u, i)
1  if i = 0
2     then (key, value) ← RANDOM-CHOICE(db(u))
3          return key
4     else (x, f) ← RANDOM-CHOICE(fingers(u, i − 1))
5          return x

FINGERS(u, i, r_f)
1  for j ← 1 to r_f
2      do v_j ← RANDOM-WALK(u)
3         x_j ← ids(v_j, i)
4  return {(x_1, v_1), . . . , (x_{r_f}, v_{r_f})}

SUCCESSORS(u, i, r_s)
1  for j ← 1 to r_s
2      do v_j ← RANDOM-WALK(u)
3         R_j ← SUCCESSORS-SAMPLE(v_j, ids(u, i))
4  return R_1 ∪ · · · ∪ R_{r_s}

SUCCESSORS-SAMPLE(u, x_0)
1  {(key_1, value_1), . . . , (key_{r_d}, value_{r_d})} ← db(u)
        (sorted so that x_0 ⪯ key_1 ⪯ · · · ⪯ key_{r_d} ≺ x_0)
2  return {(key_1, value_1), . . . , (key_t, value_t)}   (for small t)
```

Figure 4: SETUP procedure to build structured routing tables. Each function's first parameter is the node it executes on.

The remaining phases are used to construct the ℓ layers. For each layer i, SETUP chooses each node's IDs and constructs its successor and finger tables. The layer-0 ID is chosen by picking a random key from db. Higher-layer IDs and finger tables are defined mutually recursively. FINGERS(u, i, r_f) sends out r_f random walks and collects the resulting nodes and i^{th} layered IDs into u's i^{th} layer finger table. For $i > 0$, CHOOSE-ID(u, i) chooses u's i^{th} layered ID by picking a random finger ID stored in u's $(i-1)^{\text{th}}$ finger table. As explained in Section 5.3, this causes honest IDs to cluster wherever Sybil IDs have clustered, ensuring a rough balance between good fingers and bad fingers in any given range of keys.

Once a node has its ID for a layer, it must collect the successor list for that ID. It might seem that we could solve this the same way Chord does, by bootstrapping off LOOKUP to find the ID's first successor node, then asking it for its own successor list, and so on. However, as pointed out in Section 5.1, this recursive approach would enable the adversary to fill up the successor tables with bogus records over time. To avoid this, Whānau fills each node's *succ* table without using any other node's *succ* table; instead, it uses only the *db* tables.

The information about any layered ID's successors is spread around the *db* tables of many other nodes, so the SUCCESSORS subroutine must contact many nodes and collect little bits of the successor list together. The straightforward way to do this is to ask each node v for the closest record in $db(v)$ following the ID.

The SUCCESSORS subroutine repeatedly calls SUCCESSORS-SAMPLE r_s times, each time accumulating a few more potential-successors. SUCCESSORS-SAMPLE works by contacting a random node and sending it a query containing the ID. The queried node v, if it is honest, sorts all of the records in its local $db(v)$ by key, and then returns the closest few records to the requestor's ID. The precise number t of records sampled does not matter for correctness, so long as t is small compared to r_d. Section 7's analysis simply lets $t = 1$.

This successor sampling technique ensures that for appropriate values of r_d and r_s, the union of the repeated queries will contain all the desired successor records. Section 7.1 will state this quantitatively, but the intuition is as follows. Each SUCCESSORS-SAMPLE query is an independent and random sample of the set of keys in the system which are near the ID. There may be substantial overlap in the result sets, but for sufficiently large r_s, we will eventually receive all immediate successors. Some of the records returned will be far away from the ID and thus not really successors, but they will show up only a few times. Likewise, bogus results returned by Sybil nodes consume some storage space, but do not affect correctness, since they do not prevent the true successors from being found.

```
LOOKUP(u, key)
1  v ← u
2  repeat  value ← TRY(v, key)
3          v ← RANDOM-WALK(u)
4    until TRY found valid value, or hit retry limit
5  return value

TRY(u, key)
1  {(x_1, f_1), ..., (x_{r_f}, f_{r_f})} ← fingers(u, 0)
                       (sorted so key ⪯ x_1 ⪯ ··· ⪯ x_{r_f} ≺ key)
2  j ← r_f
3  repeat  (f, i) ← CHOOSE-FINGER(u, x_j, key)
4          value ← QUERY(f, i, key)
5          j ← j − 1
6    until QUERY found valid value, or hit retry limit
7  return value

CHOOSE-FINGER(u, x_0, key)
1  for i ← 0 to ℓ − 1
2      do F_i ← { (x, f) ∈ fingers(u, i) | x_0 ⪯ x ⪯ key }
3  i ← RANDOM-CHOICE({i ∈ {0, ..., ℓ−1} | F_i non-empty})
4  (x, f) ← RANDOM-CHOICE(F_i)
5  return (f, i)

QUERY(u, i, key)
1  if (key, value) ∈ succ(u, i) for some value
2    then return value
3    else error "not found"
```

Figure 5: LOOKUP procedure to retrieve a record by key.

In order to process requests quickly, each node should sort its finger tables by ID and its successor tables by key.

6.2 Lookup

The basic goal of the LOOKUP procedure is to find a finger node which is honest and which has the target record in its successor table. The SETUP procedure ensures that any honest finger f which is "close enough" to the target key y will have $y \in succ(f)$. Since every finger table contains many random honest nodes, each node is likely to have an honest finger which is "close enough" (if r_f is big enough). However, if the adversary clusters IDs near the target key, then LOOKUP might have to waste many queries to Sybil fingers before finding this honest finger. LOOKUP's pseudocode (Figure 5) chooses fingers carefully to foil this category of attack.

To prevent the adversary from focusing its attack on a single node's finger table, LOOKUP tries first using its own finger table, and, if that fails, repeatedly chooses a random delegate and retries the search from there.

The TRY subroutine searches the finger table for the closest layer-zero ID x_0 to the target key key. It then chooses a random layer i to try, and a random finger f whose ID in that layer lies between x_0 and the target key. TRY then queries f for the target key.

If there is no clustering attack, then the layer-zero ID x_0 is likely to be an honest ID; if there is a clustering attack, that can only make x_0 become closer to the target key. Therefore, in *either* case, any honest finger found between x_0 and key will be close enough to have the target record in its successor table.

Only one question remains: how likely is CHOOSE-FINGER to pick an honest finger versus a Sybil finger? Recall from Section 5.3 that, during SETUP, if the adversary clustered his IDs in the range $[x_0, key]$ in layer i, then the honest nodes tended to cluster in the same range in layer $i + 1$. Thus, the adversary's fingers cannot dominate the range in the majority of layers. Now, the layer chosen by CHOOSE-FINGER is random — so, probably not dominated by the adversary — and therefore, a finger chosen from that layer is likely to be honest.

In conclusion, for most honest nodes' finger tables, CHOOSE-FINGER has a good probability of returning an honest finger which is close enough to have the target key in its successor table. Therefore, LOOKUP should almost always succeed after only a few calls to TRY.

7 Analysis of Whānau's performance

For the same reason as a flooding protocol, Whānau's LOOKUP will always eventually succeed if it runs for long enough: some random walk (LOOKUP, line 3) will find the target node. However, the point of Whānau's added complexity is to improve lookup performance beyond a flooding algorithm. This section sketches the reasoning why LOOKUP uses $O(1)$ messages to find any target key, leaving out most proofs; more detailed proofs can be found in an accompanying technical report [15].

To the definitions in Section 4, we will add a few more in order to set up our analysis.

Definition (good sample probability)**.** Let $\boldsymbol{p}$ be the probability that a random walk starting from a winner node returns a good sample (a random honest node). p decreases with the number of attack edges g. Specifically, we have previously shown that $p \geq \frac{1}{2}\left(1 - \frac{gw}{\epsilon n}\right)$ for any ϵ [15]. We are interested in the case where $g < \frac{\epsilon n}{2w} = O\left(\frac{n}{w}\right)$. In this case, we have that $p > 1/4$, so a substantial fraction of random walks return good samples.

Definition ("the database")**.** Let $\boldsymbol{\mathcal{D}}$ be the disjoint union of all the honest nodes' db tables:

$$\mathcal{D} \stackrel{\text{def}}{=} \biguplus_{\text{honest } u} db(u)$$

Intuitively, we expect honest nodes' records to be heavily represented in $\mathcal{D}$. $\mathcal{D}$ has exactly $r_d m$ elements; we expect at least $(1 - \epsilon) p r_d m$ of those to be from honest nodes.

Definition (distance metric $\boldsymbol{d_{xy}}$)**.** Recall from Section 5.3 that Whānau has no *a priori* notion of distance between two keys. However, with the definition of $\mathcal{D}$, we can construct an *a posteriori* distance metric.

Let $\mathcal{D}_{xy} \stackrel{\text{def}}{=} \{z \in \mathcal{D} \mid x \preceq z \prec y\}$ be all the records (honest and Sybil) in $\mathcal{D}$ on the arc $[x, y)$. Then define

$$d_{xy} \stackrel{\text{def}}{=} \frac{|\mathcal{D}_{xy}|}{|\mathcal{D}|} = \frac{|\mathcal{D}_{xy}|}{r_d m} \in [0, 1)$$

Note that d_{xy} is not used (or indeed, observable) by the protocol; we use it only in the analysis.

7.1 Winner successor tables are correct

Recall that SETUP (Figure 4) uses the SUCCESSORS subroutine, which calls SUCCESSORS-SAMPLE r_s times, to find all the honest records in $\mathcal{D}$ immediately following an ID x. Consider an arbitrary successor key $y \in \mathcal{D}$. If the r_d and r_s are sufficiently large, and d_{xy} is sufficiently small, then y will almost certainly be returned by some call to SUCCESSORS-SAMPLE. Thus, any winner node u's table $succ(u, i)$ will ultimately contain all records y close enough to the ID $x = ids(u, i)$.

Lemma. *Call* SUCCESSORS-SAMPLE(x) r_s *times. We then have (for* $r_d, d_{xy}^{-1} \gg 1$ *and* $r_s \ll n$*) a* $\text{Prob}[\mathit{fail}]$ *of:*

$$\text{Prob}[y \notin succ(u, i)] \lesssim \left[1 - \frac{(1-\epsilon)p}{1 + \frac{km}{pr_d}} e^{-r_d d_{xy}}\right]^{r_s}$$

Under the simplifying assumption $r_d < d_{xy}^{-1} \ll km$*:*

$$\text{Prob}[y \notin succ(u, i)] \lesssim e^{-e(1-\epsilon)p^2 \frac{r_s r_d}{km}} \qquad (1)$$

We can intuitively interpret this result as follows: to get a complete successor table with high probability, we need $r_s r_d = \Omega(km \log km)$. This is related to the Coupon Collector's Problem: the SUCCESSORS subroutine examines $r_s r_d$ random elements from $\mathcal{D}$, and it must examine the entire set of km honest records.

7.2 Layer zero IDs are evenly distributed

Consider an arbitrary winner u's layer-zero finger table $\mathcal{F}_0 = \mathit{fingers}(u, 0)$: approximately pr_f of the nodes in $\mathcal{F}_0$ will be random honest nodes. Picking a random honest node $f \in \mathcal{F}_0$ and then picking a random key from $db(f)$ is the same as picking a random key from $\mathcal{D}$. Thus, pr_f of the IDs in $\mathcal{F}_0$ are random keys from $\mathcal{D}$. For any keys $x, y \in \mathcal{D}$, the probability that a random honest finger's layer-zero ID falls in the range $[x, y)$ is simply d_{xy}.

Lemma. *With* r_f *fingers, we have a* $\text{Prob}[\mathit{fail}]$ *of:*

$$\text{Prob}[\text{ no layer-0 finger in } [x, y)\,] \lesssim (1 - d_{xy})^{pr_f} \qquad (2)$$

We expect to find approximately $pr_f d_{xy}$ *of these honest fingers with IDs in the range* $[x, y)$*.*

We can intuitively interpret this result as follows: to see $\Omega(1)$ fingers in $[x, y)$ with high probability, we need $r_f = \Omega(\log m / d_{xy})$. In other words, large finger tables enable nodes to find a layer-0 finger in any small range of keys. Thus layer-0 finger tables tend to cover $\mathcal{D}$ evenly.

7.3 Layers are immune to clustering

The adversary may attack the finger tables by clustering its IDs. CHOOSE-ID line 4 causes honest nodes to respond by clustering their IDs on the same keys.

Pick any keys $x, y \in \mathcal{D}$ sufficiently far apart that we expect at least one layer-zero finger ID in $[x, y)$ with high probability (as explained above). Let β_i ("bad fingers") be the average (over winners nodes' finger tables) of the number of Sybil fingers with layer-i IDs in $[x, y)$. Likewise, let γ_i ("good fingers") be the average number of winner fingers in $[x, y)$. Define $\boldsymbol{\mu} \stackrel{\text{def}}{=} (1 - \epsilon)p$.

Lemma. *The number of good fingers in* $[x, y)$ *is proportional to the total number of fingers in the previous layer:*

$$\gamma_{i+1} \gtrsim \mu(\gamma_i + \beta_i)$$

Corollary. *Let the density* $\boldsymbol{\rho_i}$ *of winner fingers in layer* i *be* $\rho_i \stackrel{\text{def}}{=} \gamma_i/(\gamma_i + \beta_i)$*. Then* $\prod_{i=0}^{\ell-1} \rho_i \gtrsim \mu^{\ell-1}/(1-\mu)r_f$*.*

Because the density of winner fingers ρ_i is bounded below, this result means that the adversary's scope to affect ρ_i is limited. The adversary may strategically choose any values of β_i between zero and $(1 - \mu)r_f$. However, the adversary's strategy is limited by the fact that if it halves the density of good nodes in one layer, the density of good nodes in another layer will necessarily double.

Theorem. *The average layer's density of winner fingers is at least* $\bar{\rho} \stackrel{\text{def}}{=} \frac{1}{\ell} \sum_{i=0}^{\ell-1} \rho_i \gtrsim \frac{\mu}{e} \left[(1 - \mu)\mu r_f\right]^{-\frac{1}{\ell}}$*.*

Observe that as $\ell \to 1$, the average layer's density of good fingers shrinks exponentially to $O(1/r_f)$, and that as $\ell \to \infty$, the density of good fingers asymptotically approaches the limit μ/e. We can get $\bar{\rho}$ within a factor of e of this ideal bound by setting the number of layers ℓ to

$$\ell = \log\left[(1 - \mu)\mu r_f\right] \qquad (3)$$

For most values of $\mu \in [0, 1]$, $\ell \approx \log r_f$. However, when μ approaches 1 (no attack) or 0 (strong attack), $\ell \to 1$.

7.4 Main result: lookup is fast

The preceding sections' tools enable us to prove that Whānau uses a constant number of messages per lookup.

Theorem (Main theorem)**.** *Define* $\boldsymbol{\kappa} = kme/(1 - \epsilon)p^3$*. Suppose that we pick* r_s*,* r_f*,* r_d*, and* ℓ *so that* (3) *and* (4) *are satisfied, and run* SETUP *to build routing tables.*

$$r_s r_f > \frac{r_s r_d}{p} > \kappa \qquad (4)$$

Now run LOOKUP *on any valid key* y*. Then, a single iteration of* TRY *succeeds with probability better than* $\text{Prob}[\text{ success }] > \frac{1}{20}(1 - \epsilon)p = \Omega(1)$*.*

The value κ is the aggregate storage capacity km of the DHT times an overhead factor $e/(1-\epsilon)p^3$ which represents the extra work required to protect against Sybil attacks. When $g < \frac{\epsilon n}{2w}$, this overhead factor is $O(1)$.

The formula (4) may be interpreted to mean that both $r_s r_d$ and $r_s r_f$ must be $\Omega(\kappa)$: the first so that SUCCESSORS-SAMPLE is called enough times to collect every successor, and the second so that successor lists are longer than the distance between fingers. These would both need to be true even with no adversary.

Proof sketch. Let $x \in \mathcal{D}$ be a key whose distance to the target key y is $d_{xy} = 1/pr_f$, the average distance between honest fingers.

First, substitute the chosen d_{xy} into (2). By the lemma, the probability that there is an honest finger $x_h \in [x, y)$ is at least $1 - 1/e$. TRY line 1 finds x_{r_f}, the closest layer-zero finger to the target key, and TRY passes it to CHOOSE-FINGER as x_0. x_0 may be an honest finger or a Sybil finger, but in either case, it must be at least as close to the target key as x_h. Thus, $x_0 \in [x, y)$ with probability at least $1 - 1/e$.

Second, recall that CHOOSE-FINGER first chooses a random layer, and then a random finger f from that layer with ID $x_f \in [x_0, y]$. The probability of choosing any given layer i is ℓ^{-1}, and the probability of getting an honest finger from the range is ρ_i from Section 7.3. Thus, the total probability that CHOOSE-FINGER returns an honest finger is simply the average layer's density of good nodes $\frac{1}{\ell}\sum \rho_i = \bar{\rho}$. Since we assumed (3) was satisfied, Section 7.3 showed that the probability of success is at least $\bar{\rho} \gtrsim (1-\epsilon)p/e^2$.

Finally, if the chosen finger f is honest, the only question remaining is whether the target key is in f's successor table. Substituting $d_{x_f y} < d_{xy}$ and (4) into (1) yields $\mathrm{Prob}[\, y \in \mathit{succ}(f)\,] \gtrsim 1 - 1/e$. Therefore, when QUERY$(f, y)$ checks f's successor table, it succeeds with probability at least $1 - 1/e$.

A TRY iteration will succeed if three conditions hold: (1) $x_f \in [x, y)$; (2) CHOOSE-FINGER returns a winning finger f; (3) $y \in \mathit{succ}(f)$. Combining the probabilities calculated above for each of these events yields the total success probability $\left(1-\frac{1}{e}\right)\frac{(1-\epsilon)p}{e^2}\left(1-\frac{1}{e}\right) > \frac{1}{20}(1-\epsilon)p$. □

Corollary. *The expected number of queries sent by* LOOKUP *is bounded by* $\frac{20}{(1-\epsilon)p} = O(1)$. *With high probability, the maximum number of queries is* $O(\log n)$.

7.5 Routing tables are small

Each (virtual) node has $S = r_d + \ell(r_f + r_s)$ table entries in total. To minimize S subject to (4), set $r_s = r_f = \sqrt{\kappa}$ and $r_d = p\sqrt{\kappa}$. Therefore, the optimal total table size is $S \approx \sqrt{\kappa}\log\kappa$, so $S = O(\sqrt{km}\log km)$, as expected.

As the number of attack edges g increases, the required table size grows as $(1-\epsilon)^{-1/2}p^{-3/2}$. A good approximation for this security overhead factor is $1 + 2\sqrt{\frac{gw}{n}} + 6\frac{gw}{n}$ when $g < \frac{n}{6w}$. Thus, overhead grows linearly with g.

As one might expect for a one-hop DHT, the optimum finger tables and the successor tables are the same size. The logarithmic factor in the total table size comes from the need to maintain $O(\log km)$ layers to protect against clustering attacks. If the number of attack edges is small, (3) indicates that multiple layers are unnecessary. This is consistent with the experimental data in Section 9.3.

8 Implementation

We have implemented Whānau in a simulator and on PlanetLab. To simulate very large networks — some of the social graphs we use have millions of nodes — we wrote our own simulator. Existing peer-to-peer simulators don't scale to such a large number of nodes, and our simulator uses many Whānau-specific optimizations to reduce memory consumption and running time. The simulator directly implements the protocol as described in Figures 4 and 5, takes a static social network as input, and provides knobs to experiment with Whānau's different parameters. The simulator does not simulate real-world network latencies and bandwidths, but only counts the number of messages that Whānau sends. The primary purpose of the simulator is to validate the correctness and scaling properties of Whānau with large social networks.

We also implemented Whānau and the IM application in Python on PlanetLab. This implementation runs a message-passing protocol to compute SETUP and uses RPC to implement LOOKUP. When a user starts a node, the user provides the keys and current IP addresses that identify their social neighbor nodes. The IM client stores its current IP address into the DHT. When a user wants to send an IM to another user, the IM client looks up the target user's contact information in the DHT and authenticates the returned record using the key. If the record is authentic, the IM application sends the IM to the IP address in the record. Whānau periodically rebuilds its tables to incorporate nodes which join and leave.

The average latency for a lookup is usually one round-trip on PlanetLab. Using locality-aware routing, Whānau could achieve lower than one network round-trip on average, but we haven't implemented this feature yet.

Our PlanetLab experiments were limited by the number of PlanetLab nodes available and their resources: we were able to run up to 4000 Whānau nodes simultaneously. Unfortunately, at scales smaller than this, Whānau nearly reduces to simple broadcast. Given this practical limitation, it was difficult to produce insightful scaling results on PlanetLab. Furthermore, although results were broadly consistent at small scales, we could not cross-validate the simulator at larger scales. The PlanetLab experiments primarily demonstrated that Whānau works on a real network with churn, varying delays, and so on.

	n=#nodes	m=#edges	avg. degree
Flickr	1,624,992	15,476,835	9.52
LiveJournal	5,189,809	48,688,097	9.38
YouTube	1,134,890	2,987,624	2.63
DBLP	511,163	1,871,070	3.66

Table 2: Properties of the input data sets.

9 Results

This section experimentally verifies several hypotheses: (1) real-world social networks exhibit the properties that Whānau relies upon; (2) Whānau can handle clustering attacks (tested by measuring its performance versus table size and the number of attack edges); (3) layered IDs are essential for handling clustering attacks; (4) Whānau achieves the same scalability as insecure one-hop DHTs; and (5) Whānau can handle node churn in Planetlab.

Our Sybil attack model permits the adversary to create an unlimited number of pseudonyms. Since previous DHTs cannot tolerate this attack at all, this section does not compare Whānau's Sybil-resistance against previous DHTs. However, in the non-adversarial case, the experiments do show that Whānau scales like any other insecure one-hop DHT, so (ignoring constant factors such as cryptographic overhead) adding security is "free". Also, similarly to other (non-locality-aware) one-hop DHTs, the lookup latency is one network round-trip.

9.1 Real-world social nets fit assumptions

Nodes in the Whānau protocol bootstrap from a social network to build their routing tables. It is important for Whānau that the social network is fast mixing: that is, a short random walk starting from any node should quickly approach the stationary distribution, so that there is roughly an equal probability of ending up at any edge (virtual node). We test if this fast-mixing property holds for social network graphs, extracted from Flickr, LiveJournal, YouTube, and DBLP, which have also been used in other studies [18, 26]. These networks correspond to real-world users and their social connections. The LiveJournal graph was estimated to cover 95.4% of the users in Dec 2006, and the Flickr graph 26.9% in Jan 2007.

We preprocessed the input graphs by discarding unconnected nodes and transforming directed edges into undirected edges. (The majority of links were already symmetric.) The resulting graphs' basic properties are shown in Table 2. The node degrees follow power law distributions, with coefficients between 1.6 and 2 [18].

To test the fast-mixing property, we sample the distribution of random walks as follows. We pick a random starting edge i, and for each ending edge j, compute the probability p_{ij} that a walk of length w ends at j. Computing p_{ij} for all m possible starting edges i is too time-intensive, so we sampled 100 random starting edges i and computed p_{ij} for all m end edges j. For a fast-mixing network, we expect the probability of ending up at a particular edge to approach $1/m$ as w increases to $O(\log n)$.

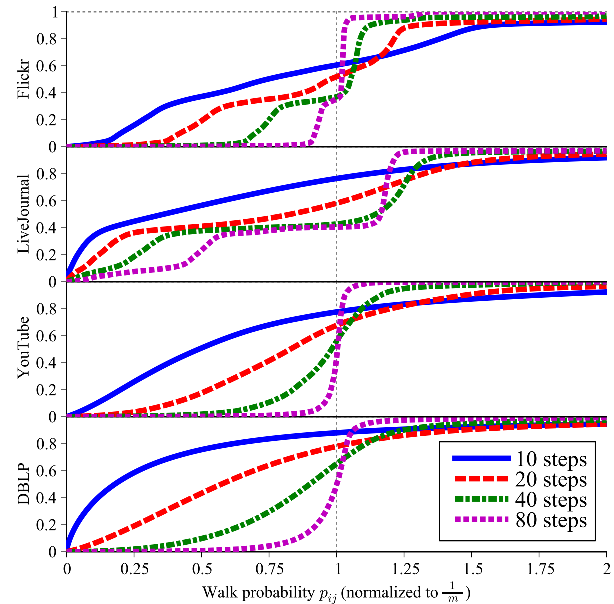


Figure 6: Mixing properties of social graphs. Each line shows a CDF of the probability that a w-step random walk ends on a particular edge. The X axis is normalized so that the mean is 1.

Figure 6 plots the CDF of p_{ij} for increasing values of w. To compare the different social graphs we normalize the CDFs so that they have the same mean. Thus, for all graphs, $p_{ij} = 1/m$ corresponds to the ideal line at 1. As expected, as the number of steps increases to 80, each CDF approaches the ideal uniform distribution.

The CDFs at $w = 10$ are far from the ideal distribution, but there are two reasons to prefer smaller values of w. First, the amount of bandwidth consumed scales as w. Second, larger values of w increase the chance that a random walk will return a Sybil node. Section 9.2 will show that Whānau works well even when the distribution of random walks is not perfect.

Recall from Section 4.2 that when a fast-mixing social network has a sparse cut between the honest nodes and Sybil nodes, random walks are a powerful tool to protect against Sybil attacks. To confirm that this approach works with real-world social networks, we measured the probability that a random walk escapes the honest region of the Flickr network with different numbers of attack edges. To generate an instance with g attack edges, we marked random nodes as Sybils until there were at least g edges between marked nodes and non-marked nodes, and then removed any honest nodes which were connected only to Sybil nodes. For example, for the Flickr graph, in the instance with $g = 1,940,689$, there are $n = 1,442,120$ honest nodes (with $m = 13,385,439$ honest edges) and 182,872 Sybil nodes. Since increasing

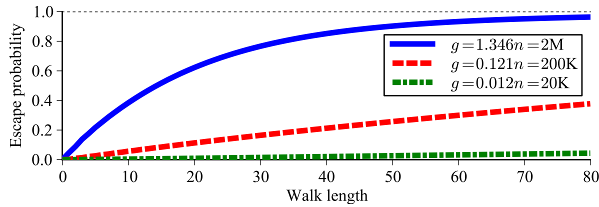


Figure 7: Escape probability on the Flickr network.

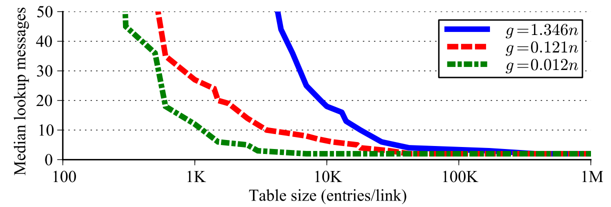


Figure 8: Number of messages used by LOOKUP decreases as table size increases (Flickr social network).

the number of attack edges this way actually consumes honest nodes, it is not possible to test the protocol against g/n ratios substantially greater than 1.

Figure 7 plots the probability that a random walk starting from a random honest node will cross an attack edge. As expected, this escape probability increases with the number of steps and with the number of attack edges. When the number of attack edges is greater than the number of honest nodes, the adversary has convinced essentially all of the system's users to form links to its Sybil identities. In this case, long walks almost surely escape from the honest region; however, short walks still have substantial probability of reaching an honest node. For example, if the adversary controls 2 million attack edges on the Flickr network, then each user has an average of 1.35 links to the adversary, and random walks of length 40 are 90% Sybils. On the other hand, random walks of length 10 will return 60% honest nodes, although those honest nodes will be less uniformly distributed than a longer random walk.

9.2 Performance under clustering attack

To evaluate Whānau's resistance against the Sybil attack, we ran instances of the protocol using a range of table sizes, number of layers, and adversary strengths. For each instance, we chose random honest starting nodes and measured the number of messages used by LOOKUP to find randomly chosen target keys. Our analysis predicted that the number of messages would be $O(1)$ as long as $g \ll n/w$. Since we used a fixed $w = 10$, the number of messages should be small when the number of attack edges is less than 10% of the number of honest nodes. We also expected that increasing the table size would reduce the number of messages.

Our simulated adversary employs a clustering attack on the honest nodes' finger tables, choosing all of its IDs to immediately precede the target key. In a real-world deployment of Whānau, it is only possible for an adversary to target a small fraction of honest keys in this way: to increase the number of Sybil IDs near a particular key, the adversary must move some Sybil IDs away from other keys. However, in our simulator, we allowed the adversary to change its IDs between every LOOKUP operation: that is, it can start over from scratch and adapt its attack to the chosen target key. Our results therefore show Whānau's worst case performance, and not the average case performance for random target keys.

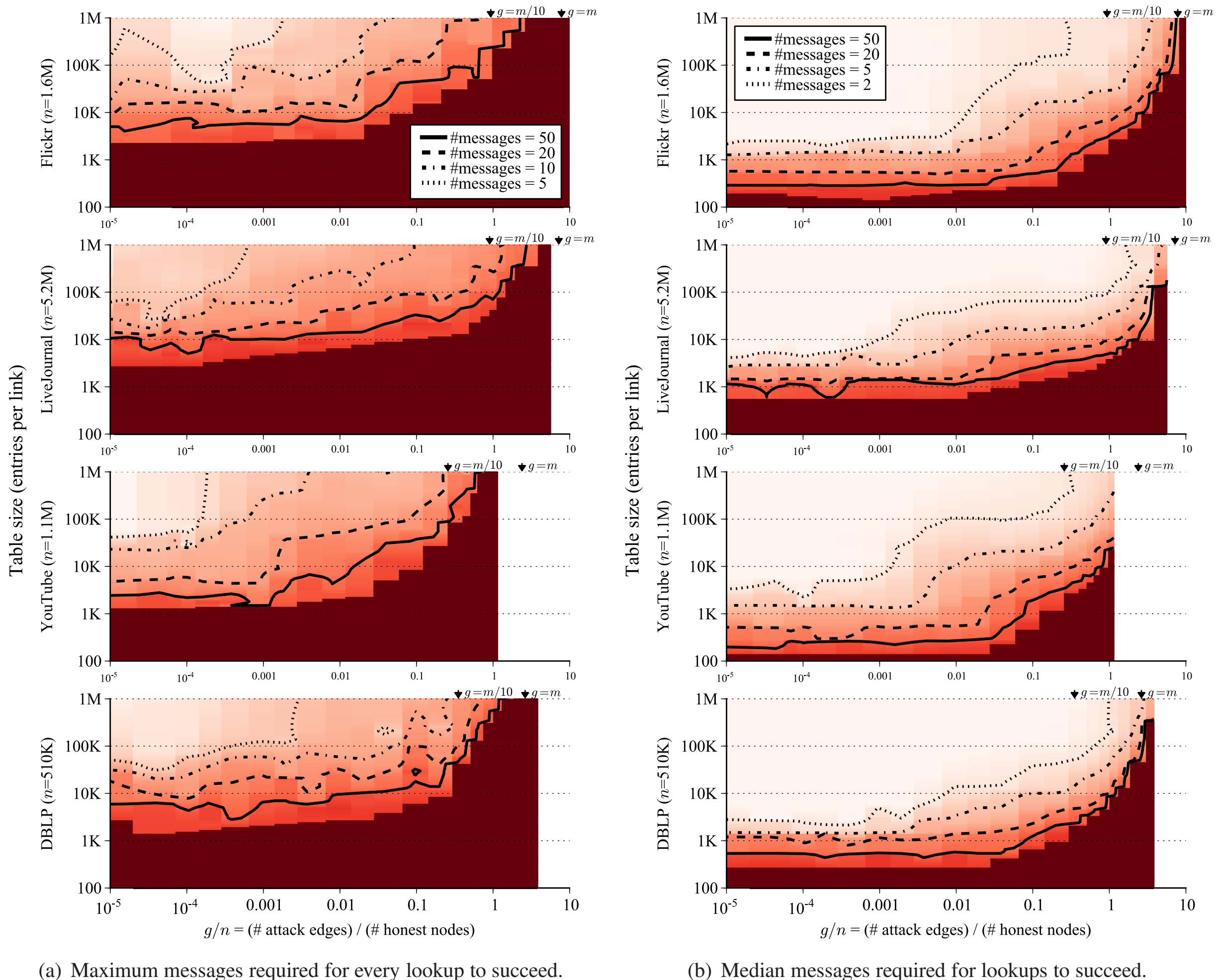
Figure 8 plots the number of messages required by LOOKUP versus table size. Since our policy is that resources scale with node degree (Section 3.3), we measure table size in number of entries per social link. Each table entry contains a key and a node's address (finger tables) or a key-value pair (successor and db tables).

As expected, the number of messages decreases with table size and increases with the adversary's power. For example, on the Flickr network and with a table size of 10,000 entries per link, the median LOOKUP required 2 messages when the number of attack edges is 20,000, but required 20 messages when there are 2,000,000 attack edges. The minimum resource budget for fast lookups is $1,000 \approx \sqrt{n}$: below this table size, LOOKUP messages increased rapidly even without any attack. Under a massive attack ($g > n$) LOOKUP could still route quickly, but it required a larger resource budget of $\gtrsim 10,000$.

Figure 9 shows the full data set of which Figure 8 is a slice. Figure 9(a) shows the number of messages required for 100% of our test lookups to succeed. Of course, most lookups succeeded with far fewer messages than this upper bound. Figure 9(b) shows the number of messages required for 50% of lookups to succeed. The contour lines for maximum messages are necessarily noisier than for median messages, because the lines can easily be shifted by the random outcome of a single trial. The median is a better guideline to Whānau's expected performance: for a table size of 5,000 on the Flickr graph, most lookups will succeed within 1 or 2 messages, but a few outliers may require 50 to 100 messages.

We normalized the X-axis of each plot by the number of honest nodes in each network so that the results from different datasets could be compared directly. Our theoretical analysis predicted that Whānau's performance would drop sharply (LOOKUP messages would grow exponentially) when $g > n/10$. However, we observed that, for all datasets, this transition occurs in the higher range $m/10 < g < m$. In other words, the analytic prediction was a bit too pessimistic: Whānau functions well until a substantial fraction of all edges are attack edges.

When the number of attack edges g was below $n/10$, we observed that performance was more a function of ta-



(a) Maximum messages required for every lookup to succeed.

(b) Median messages required for lookups to succeed.

Figure 9: Heat map and contours of the number of messages used by LOOKUP, versus attacker strength and table size. In the light regions at upper left, where there are few attack edges and a large resource budget, LOOKUP succeeded using only one message. In the dark regions at lower right, where there are many attack edges and a small resource budget, LOOKUP needed more than the retry limit of 120 messages. Wedges indicate where $g = m/w$ and $g = m$; when $g \gg m/w$, LOOKUP performance degrades rapidly. The plots' right edges do not line up because it was not always possible to create an adversary instance with $g = 10n$.

ble size, which must always be at least $\Omega(\sqrt{m})$ for Whānau to function, than of g. Thus, Whānau's performance is insensitive to relatively small numbers of attack edges.

9.3 Layers vs. clustering attacks

Section 9.2 showed that Whānau handles clustering attacks. For the plots in Figure 9, we simulated several different numbers of layers and chose the best-performing value for a given table size. This section evaluates whether layers are important for Whānau's attack resistance, and investigates how the number of layers should be chosen.

Are layers important? We ran the same experiment as in Section 9.2, but we held the total table size at a constant 100,000 entries per link. We varied whether the protocol spent those resources on more layers, or on bigger per-layer routing tables, and measured the median number of messages required by LOOKUP.

We would expect that for small-scale attacks, one layer is best, because layers come at the cost of smaller per-layer tables. For more large-scale attacks, more layers is better, because layers protect against clustering attacks. Even for large-scale attacks, adding more layers yields quickly diminishing returns, and so we only simulated numbers of layers between 1 and 10.

The solid lines in Figure 10 shows the results for the clustering attack described in Section 9.2. When the number of attack edges is small, the best performance would be achieved by spending all resources on bigger routing tables, mostly avoiding layers. For Flickr, layers become important when the number of attack edges exceeds 5,000 (0.3% of n); for $g > 20,000$, a constant

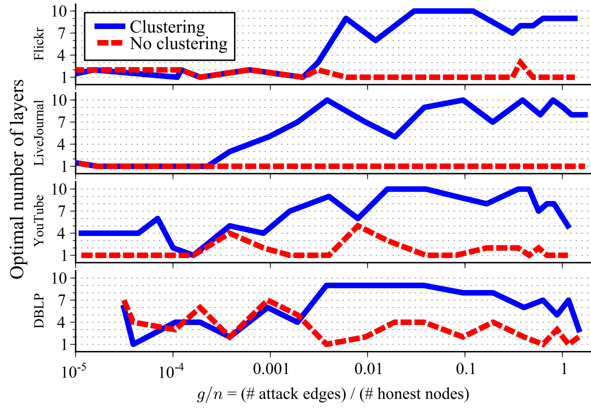


Figure 10: Optimal layers versus attacker power. The resource budget was fixed at 100K table entries per link.

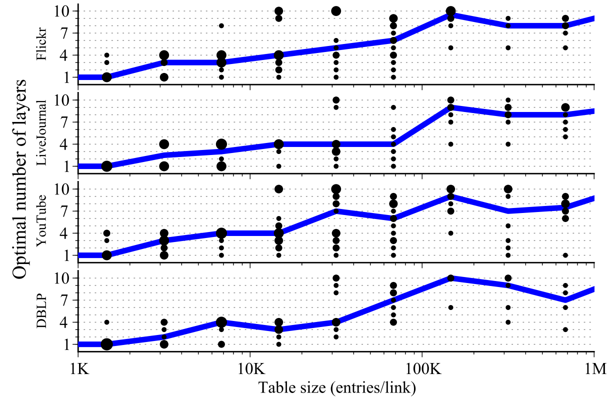


Figure 11: Optimal layers versus resource budget. Each point is a table size / attacker power instance. Larger points correspond to multiple instances. The trend line passes through the median point for each table size.

number of layers (around 8) would yield the best performance. At high attack ratios (around $g/n \gtrsim 1$), the data becomes noisy because performance degrades regardless of the choice of layers.

The dashed lines in Figure 10 show the same simulation, but pitted against a naïve attack: the adversary swallows all random walks and returns bogus replies to all requests, but does not cluster its IDs. This control data clearly shows that multiple layers are only helpful against a clustering attack. The trends are clearer for the larger graphs (Flickr and LiveJournal) than for the smaller graphs (YouTube and DBLP). 100,000 table entries is very large in comparison to the smaller graphs' sizes, and therefore the differences in performance between small numbers of layers are not as substantial.

How many layers should nodes use? The above data showed that layers improve Whānau's resistance against powerful attacks but are not helpful when the DHT is not under attack. However, we cannot presume that nodes know the number of attack edges g, so the number of layers must be chosen in some other way. Since layers cost resources, we would expect the optimal number of layers to depend on the node's resource budget. If the number of table entries is large compared to $\sqrt{m}$, then increasing the number of layers is the best way to protect against powerful adversaries. On the other hand, if the number of table entries is relatively small, then no number of layers will protect against a powerful attack; thus, nodes should use a smaller number of layers to reduce overhead.

We tested this hypothesis by re-analyzing the data collected for Section 9.2. For a given table size, we computed the number of layers that yielded optimal performance over a range of attack strengths. The results are shown in Figure 11. The overall trend is clear: at small table sizes, fewer layers is preferred, and at large table sizes, more layers is better.

The optimal number of layers is thus a function of the social network size and the resource budget, and we presume that honest nodes know both of these values at least approximately. Since Whānau's performance is not very sensitive to small changes in the number of layers, a rough estimate is sufficient to get good performance over a wide range of situations.

9.4 Whānau's scalability

Whānau is designed as a one-hop DHT. We collected simulated data to confirm that Whānau's performance scales asymptotically the same as an insecure one-hop DHT such as Kelips [11]. Since we don't have access to a wide range of social network datasets of different sizes, we generated synthetic social networks with varying numbers of nodes using the standard technique of preferential attachment [1], yielding power-law degree distributions with exponents close to 2. For each network, we simulated Whānau's performance for various table sizes and layers, as in the preceding sections. Since our goal was to demonstrate that Whānau reduces to a standard one-hop DHT in the non-adversarial case, we did not simulate any adversary.

Figure 12 plots the median number of LOOKUP messages versus table size and social network size. For a one-hop DHT, we expect that, holding the number of messages to a constant $O(1)$, the required table size scales as $O(\sqrt{m})$: the blue line shows this predicted trend. The heat map and its contours (black lines) show simulated results for our synthetic networks. For example, for $m = 10,000,000$, the majority of lookups succeeded using 1 or 2 messages for a table size of $\approx 2,000$ entries per link. The square and triangle markers plot our four real-world datasets alongside the synthetic networks for comparison. While each real network has idiosyncratic

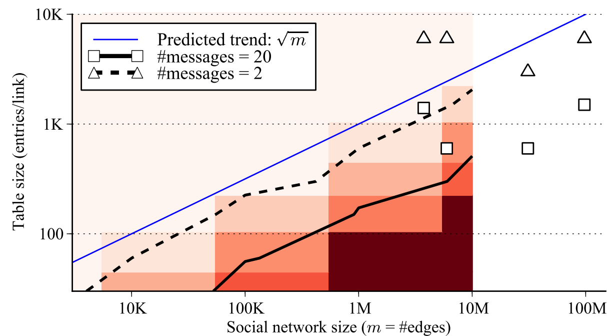


Figure 12: Number of messages used by LOOKUP, versus system size and table size. The heat map and contour lines show data from synthetic networks, while the markers show that real-world social networks fall roughly onto the same contours. Whānau scales like a one-hop DHT.

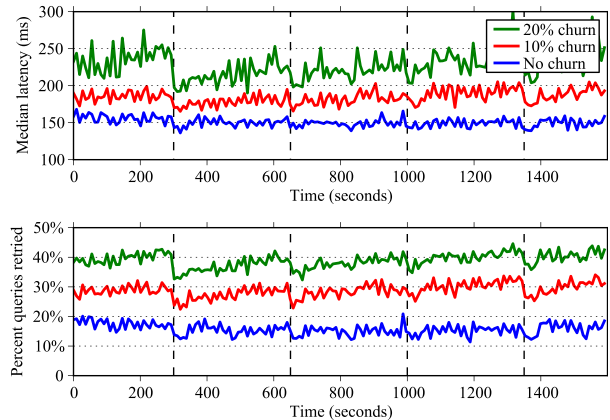


Figure 13: Lookup latency and fraction of lookups which required retries on PlanetLab under various levels of node churn. Vertical lines indicate when SETUP installed new routing tables. Under churn, the retry frequency slowly increases until SETUP runs again, at which point it reverts to the baseline.

features of its own, it is clear that the table sizes follow the $O(\sqrt{m})$ scaling trend we expect of a one-hop DHT.

9.5 PlanetLab and node churn

Whānau's example IM application runs on PlanetLab. We performed an experiment in which we started 4000 virtual nodes, running on 400 PlanetLab nodes. This number of virtual nodes is large enough that, with a routing table size of 200 entries per social link, most requests cannot be served from local tables. Each node continuously performed lookups on randomly-chosen keys.

We simulated node churn by inducing node failure and recovery events according to a Poisson process. These events occurred at an average rate of two per second, but we varied the average node downtime. At any given time, approximately 10% or 20% of the virtual nodes were offline. (In addition to simulating 10% and 20% failures, we simulated an instance without churn as a control.) We expected lookup latency to increase over time as some finger nodes became unavailable and some lookups required multiple retries. We also expected latency to go down whenever SETUP was re-run, building new routing tables to reflect the current state of the network.

Figure 13 plots the lookup latency and retries for these experiments, and shows that Whānau is largely insensitive to modest node churn. The median latency is approximately a single network roundtrip within PlanetLab, and increases gradually as churn increases. As expected, the fraction of requests needing to be retried increased with time when node churn was present, but running SETUP restored it to the baseline.

While this experiment's scale is too small to test Whānau's asymptotic behavior, it demonstrates two points: (1) Whānau functions on PlanetLab, and (2) Whānau's simple approach for maintaining routing tables is sufficient to handle reasonable levels of churn.

10 Discussion

This section discusses some engineering details and suggests some improvements that we plan to explore in future work.

Systolic mixing process. Most of Whānau's bandwidth is used to explore random walks. Therefore, it makes sense to optimize this part of the protocol. Using a recursive or iterative RPC to compute a random walk, as suggested by Figure 4, is not very efficient: it uses w messages per random node returned.

A better approach, implemented in our PlanetLab experiment, is to batch-compute r walks at once. Suppose that every node maintains a pool of r addresses of other nodes; the pools start out containing r copies of the node's own address. At each time step, each node randomly shuffles its pool and divides it equally amongst its social neighbors. For the next time step, the node combines the messages it received from each of its neighbors to create a new pool, and repeats. After w such mixing steps, each node's pool is a randomly shuffled assortment of addresses. If r is sufficiently large, this process approximates sending out r random walks from each node.

Very many or very few keys per node. The protocol described in this paper handles $1 \lesssim k \lesssim m$ well, where k is the number of keys per honest node. The extreme cases outside this range will require tweaks to Whānau to handle them. Consider the case $k > m$. Any DHT requires at least $k = \Omega(m)$ resources per node just to transmit and store the keys. This makes the task easier, since we could use $O(m)$ bandwidth to collect a nearly-complete list of all other honest nodes on each honest node. With such a list, the task of distributing successor records is a simple variation of consistent hashing [13].

The analysis in Section 7.1 breaks down for $k > m$: more than m calls to SUCCESSORS-SAMPLE will tend to have many repeats, and thus can't be treated as independent trials. To recover this property, we can treat each node as k/m virtual nodes, as we did with node degrees.

Now consider the other extreme: $k < 1$, i.e. only some nodes are storing key-value records in the system. The extreme limiting case is only a single honest node storing a key-value record into the system, i.e. $k = 1/m$. Whānau can be modified to handle the case $k < 1$ by adopting the systolic mixing process described above and omitting empty random walks. This reduces to flooding in the extreme case, and smoothly adapts to larger k.

Handling key churn. It is clear that more bandwidth usage can be traded off against responsiveness to churn: for example, running SETUP twice as often will result in half the latency from key insertion to key visibility. Using the observation that the DHT capacity scales with the table size squared, we can improve this bandwidth-latency tradeoff. Consider running SETUP every T seconds with R resources, yielding a capacity of $K = O(R^2)$ keys. Compare with this alternative: run SETUP every $T/2$ seconds using $R/2$ resources, and save the last four instances of the routing tables. Each instance will have capacity $K/4$, but since we saved four instances, the total capacity remains the same. The total resource usage per unit time also remains the same, but the responsiveness to churn doubles, since SETUP runs twice as often.

This scaling trick might seem to be getting "something for nothing". Indeed, there is a price: the number of lookup messages required will increase with the number of saved instances. However, we believe it may be possible to extend Whānau so that multiple instances can be combined into a single larger routing table, saving both storage space and lookup time.

11 Summary

This paper presents the first efficient DHT routing protocol which is secure against powerful denial-of-service attacks from an adversary able to create unlimited pseudonyms. Whānau combines previous ideas — random walks on fast-mixing social networks — with the idea of layered identifiers. We have proved that lookups complete in constant time, and that the size of routing tables is only $O(\sqrt{km}\log km)$ entries per node for an aggregate system capacity of km keys. Simulations of an aggressive clustering attack, using social networks from Flickr, LiveJournal, YouTube, and DBLP, show that when the number of attack edges is less than 10% of the number of honest nodes and the routing table size is $\sqrt{m}$, most lookups succeed in only a few messages. Thus, the Whānau protocol performs similarly to insecure one-hop DHTs, but is strongly resistant to Sybil attacks.

Acknowledgements. Many thanks to Alan Mislove for providing invaluable social network datasets. We are grateful to the anonymous reviewers and to our shepherd, Timothy Roscoe, for their many helpful comments. This research was supported by the T-Party Project (a joint research program between MIT and Quanta Computer Inc., Taiwan) and by the NSF FIND program.

References

[1] A.-L. Barabási and R. Albert. Emergence of Scaling in Random Networks. *Science,* 286(5439), 1999.

[2] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating Systems Support for Planetary-Scale Network Services. *NSDI,* San Francisco, CA, Mar. 2004.

[3] N. Borisov. Computational Puzzles as Sybil Defenses. *Peer-to-Peer Computing,* Cambridge, UK, Sept. 2006.

[4] M. Castro, P. Druschel, A. J. Ganesh, A. I. T. Rowstron, and D. S. Wallach. Secure Routing for Structured Peer-to-Peer Overlay Networks. *OSDI,* Boston, MA, Dec. 2002.

[5] A. Cheng and E. Friedman. Sybilproof Reputation Mechanisms. *Workshop on the Economics of Peer-to-Peer Systems,* Philadelphia, PA, Aug. 2005.

[6] F. Chung. *Spectral Graph Theory.* American Mathematical Society, 1997.

[7] G. Danezis and P. Mittal. SybilInfer: Detecting Sybil Nodes Using Social Networks. *NDSS,* San Diego, CA, Feb. 2009.

[8] G. Danezis, C. Lesniewski-Laas, M. F. Kaashoek, and R. J. Anderson. Sybil-Resistant DHT Routing. *ESORICS,* 2005.

[9] J. R. Douceur. The Sybil Attack. *IPTPS,* Cambridge, MA, Mar. 2002.

[10] A. Gupta, B. Liskov, and R. Rodrigues. Efficient Routing for Peer-to-Peer Overlays. *NSDI,* San Francisco, CA, Mar. 2004.

[11] I. Gupta, K. P. Birman, P. Linga, A. J. Demers, and R. van Renesse. Kelips: Building an Efficient and Stable P2P DHT through Increased Memory and Background Overhead. *IPTPS,* Berkeley, CA, Feb. 2003.

[12] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. *USITS,* Seattle, WA, Mar. 2003.

[13] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. *STOC,* El Paso, TX, May 1997.

[14] C. Lesniewski-Laas. A Sybil-Proof One-Hop DHT. *Workshop on Social Network Systems,* Glasgow, Scotland, April 2008.

[15] C. Lesniewski-Laas and M. F. Kaashoek. Whanaungatanga: Sybil-Proof Routing with Social Networks. MIT, Technical Report MIT-CSAIL-TR-2009-045, Sept. 2009.

[16] B.N. Levine, C. Shields, and N.B. Margolin. A Survey of Solutions to the Sybil Attack. University of Massachusetts Amherst, Amherst, MA, 2006.

[17] S. Marti, P. Ganesan, and H. Garcia-Molina. DHT Routing Using Social Links. *IPTPS,* La Jolla, CA, Feb. 2004.

[18] A. Mislove, M. Marcon, P. K. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and Analysis of Online Social Networks. *IMC,* San Diego, CA, Oct. 2007.

[19] B. C. Popescu, B. Crispo, and A. S. Tanenbaum. Safe and Private Data Sharing with Turtle: Friends Team-Up and Beat the System. *Security Protocols Workshop,* Cambridge, UK, Apr. 2004.

[20] R. Rodrigues and B. Liskov. Rosebud: A Scalable Byzantine-Fault-Tolerant Storage Architecture. MIT CSAIL, Technical Report TR/932, Dec. 2003.

[21] H. Rowaihy, W. Enck, P. McDaniel, and T. L. Porta. Limiting Sybil Attacks in Structured P2P Networks. *INFOCOM,* Anchorage, AK, May 2007.

[22] A. Singh, T.-W. Ngan, P. Druschel, and D. S. Wallach. Eclipse Attacks on Overlay Networks: Threats and Defenses. *INFOCOM,* Barcelona, Spain, Apr. 2006.

[23] E. Sit and R. Morris. Security Considerations for Peer-to-Peer Distributed Hash Tables. *IPTPS,* Cambridge, MA, Mar. 2002.

[24] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. *ToN,* 11(1), 2003.

[25] D. N. Tran, B. Min, J. Li, and L. Subramanian. Sybil-Resilient Online Content Rating. *NSDI,* Boston, MA, Apr. 2009.

[26] H. Yu, P. B. Gibbons, M. Kaminsky, and F. Xiao. A Near-Optimal Social Network Defense Against Sybil Attacks. *IEEE Symposium on Security and Privacy,* Oakland, CA, May 2008.

[27] H. Yu, M. Kaminsky, P. B. Gibbons, and A. Flaxman. SybilGuard: Defending Against Sybil Attacks Via Social Networks. *SIGCOMM,* Pisa, Italy, Sept. 2006.

[28] H. Yu, C. Shi, M. Kaminsky, P. B. Gibbons, and F. Xiao. DSybil: Optimal Sybil-Resistance for Recommendation Systems. *IEEE Symposium on Security and Privacy,* Oakland, CA, May 2009.

Crom: Faster Web Browsing Using Speculative Execution

James Mickens, Jeremy Elson, Jon Howell, and Jay Lorch
Microsoft Research
mickens,jelson,jonh,lorch@microsoft.com

Abstract

Early web content was expressed statically, making it amenable to straightforward prefetching to reduce user-perceived network delay. In contrast, today's rich web applications often hide content behind JavaScript event handlers, confounding static prefetching techniques. Sophisticated applications use custom code to prefetch data and do other anticipatory processing, but these custom solutions are costly to develop and application-specific.

This paper introduces Crom, a generic JavaScript speculation engine that greatly simplifies the task of writing low-latency, rich web applications. Crom takes preexisting, non-speculative event handlers and creates speculative versions, running them in a cloned browser context. If the user generates a speculated-upon event, Crom commits the precomputed result to the real browser context. Since Crom is written in JavaScript, it runs on unmodified client browsers. Using experiments with speculative versions of real applications, we show that pre-commit speculation overhead easily fits within user think time. We also show that speculatively fetching page data and precomputing its layout can make subsequent page loads an order of magnitude faster.

1 Introduction

With the advent of web browsing, humans began a new era of waiting for slow networks. To reduce user-perceived download latencies, researchers devised ways for browsers to prefetch content and hide the fetch delay within users' "think time" [4, 15, 17, 20, 23]. Finding prefetchable objects was straightforward because the early web was essentially a graph of static objects stitched together by declarative links. To discover prefetchable data, one merely had to traverse these links.

In the web's second decade, static content graphs have been steadily replaced by *rich Internet applications* (RIAs) that mimic the interactivity of desktop applications. RIAs manipulate complex, time-dependent server-side resources, so their content graphs are dynamic. RIAs also use client-side code to enhance interactivity. This eliminates the declarative representation of the content graph's edges, since now content can be dynamically named and fetched in response to the execution of an imperative event handler.

1.1 New Challenges to Latency Reduction

RIAs introduce three impediments to reducing user-perceived browser latencies. First, prefetching opportunities that once were statically enumerable are now hidden behind imperative code such as event handlers. Since event handlers have side effects that modify application state, they cannot simply be executed "early" to trigger object fetches and warm the browser cache.

Second, user inputs play a key role in naming the content to fetch. These inputs may be as simple as the clicking of a button, or as unconstrained as the entry of arbitrary text into a search form. Given the potentially combinatorial number of objects that are nameable by future user inputs, a prefetcher must identify a promising *subset* of these objects that are likely to be requested soon.

Third, RIAs spend a non-trivial amount of time updating the screen. Once the browser has fetched the necessary objects, it must devise a layout tree for those objects and render the tree on the display. For modern, graphically intensive applications, screen updates can take hundreds of milliseconds and consume 40% of the processor cycles used by the browser [22]. Screen updates contribute less to page load latencies than network delays do, but they are definitely noticeable to users. Unfortunately, warming the browser cache before a page is loaded will not reduce its layout or rendering cost.

1.2 Prior Solutions

To address these challenges, some RIAs use custom code to speculate on user intent. For example, email clients may prefetch the bodies of recently arrived messages, or speculatively upload attachments for emails that have not yet been sent. Photo gallery applications often prefetch large photos. Online maps speculatively download new map tiles that may be needed soon. The results page for a web search may prefetch the highest ranked targets to reduce their user-perceived load time. While such speculative code provides the desired latency reductions, it is often difficult to write and tightly integrated into an application's code, making it impossible to share across applications.

1.3 Our Solution: Crom

To ease the creation of low-latency web applications, we built Crom, a reusable framework for speculative

JavaScript execution. In the simplest case, the Crom API allows applications to mark individual JavaScript event handlers as *speculable*. For each such handler `h`, Crom creates a new version `h_shadow` which is semantically equivalent to `h` but which updates a shadow copy of the browser state. During user think-time, e.g., when the user is looking at a newly loaded page, Crom makes a shadow copy of the browser state and runs `h_shadow` in that context. As `h_shadow` runs, it fetches data, updates the shadow browser display, and modifies the application's shadow JavaScript state. Once `h_shadow` finishes, Crom stores the updated shadow context; this context includes the modified JavaScript state as well as the new screen layout. Later, if the user actually generates the speculated-upon event, Crom commits the shadow context. In most cases, the commit operation only requires a few pointer swaps and a screen redraw. This is much faster than synchronously fetching the web data, calculating a new layout, and rendering it on the screen.

To constrain the speculation space for arbitrarily-valued input elements, applications use *mutator functions*. Given the current state of the application, a mutator generates probable outcomes for an input element. For example, an application may provide an autocompleting text box which displays suggested words as the user types. Once a user has typed a few letters, e.g., "red", the application passes Crom a mutator function which modifies fresh shadow domains to represent appropriate speculations; in this example, the mutator may set one shadow domain's text box to "red sox", and another domain's text box to "red cross". Later, when the user generates an actual event for the input, Crom uses application-defined *equivalence classes* to determine whether any speculative domain is the outcome for the actual event and thus appropriate to commit.

The Crom API contains additional functionality to support speculative execution. For example, it provides an explicit AJAX cache to store prefetched data that the regular browser cache would ignore. It also provides a server-side component that makes it easier to speculatively upload client data. Taken as a whole, the Crom library makes it easier for developers to reason about asynchronous speculative computations.

1.4 Our Contributions

This paper makes the following contributions:

- We identify four sources of delay in rich web applications, explaining how they can be ameliorated through speculative execution (§4).
- We describe an implementation of the Crom API which is written in standard JavaScript and runs on unmodified browsers (§5). The library, which is 65 KB in size, dynamically creates new browser contexts, rewrites event handlers to speculatively execute inside of these contexts, and commits them when appropriate.
- We describe three implementation optimizations that mitigate JavaScript-specific performance limitations on speculative execution (§5).
- Using these optimizations, we demonstrate the feasibility of browser speculation by measuring the performance of three modified applications that use the Crom library. We show that Crom's pre-commit speculation overhead is no worse than 114 ms, making it feasible to hide speculative computation within user think time. We also quantify Crom's reduction of user-perceived latency, showing that Crom can reduce load times by an order of magnitude under realistic network conditions (§6).

By automating the low-level tasks that support speculative execution, Crom greatly reduces the implementation effort needed to write low-latency web applications.

2 Background: Client-side Scripting

The language most widely used for client-side scripting is JavaScript [7], a dynamically typed, object-oriented language. JavaScript interacts with the browser through the *Document Object Model* (DOM) [24], a standard, browser-neutral interface for examining and manipulating the content of a web page. Each element in a page's HTML has a corresponding object in the *DOM tree*. This tree is a property of the `document` object in the global `window` name space. The browser exposes the DOM tree as a JavaScript data structure, allowing client-side applications to manipulate the web page by examining and modifying the properties of DOM JavaScript objects, often referred to as *DOM nodes*. The DOM allows JavaScript code to find specific DOM nodes, create new DOM nodes, and change the parent-child relationships among DOM nodes.

A JavaScript programmer can create other objects besides DOM nodes. These are called *application heap* objects. All non-primitive objects, including functions, are essentially dictionaries that maps property names to property values. A property value is either another object or a primitive such as a number or boolean. Properties may be dynamically added to, and deleted from, an object. The `for-in` construct allows code to iterate over an object's property names at run-time.

Built-in JavaScript objects like a `String` or a DOM node have *native code implementations*— their methods are executed by the browser in a way that cannot be introspected by application-level JavaScript. In contrast, JavaScript *can* fetch the source code of a user-defined method by calling its `toString()` method.

JavaScript uses *event handlers* to make web pages interactive. An event handler is a function assigned to a special property of a DOM node; the browser will in-

voke that property when the associated event occurs.[1] For example, when a user clicks a button, the browser will invoke the `onclick` property of that button's DOM object. This gives application code an opportunity to update the page in response to user activity.

The AJAX interface [9] allows a JavaScript application to asynchronously fetch web content. AJAX is useful because JavaScript programs are single-threaded and network operations can be slow. To issue an AJAX request, JavaScript code creates an `XmlHTTPRequest` object and assigns an event handler to its `onreadystatechange` property. The browser will call this handler when reply data arrives.

3 Design

Crom's goal is to reduce the developer effort needed to create speculative applications. In particular, Crom tries to minimize the amount of *custom code* that developers must write to speculate on *user inputs* like keyboard and mouse activity. Crom's API leverages the fact that event-driven applications already have a natural grammar for expressing speculable user actions—each action can be represented as an event handler and a particular set of arguments to pass to the handler. Using the Crom API, applications can mark certain actions as speculable. Crom will then automatically perform the low-level tasks needed to run the speculative code paths, isolate their side-effects, and commit their results if appropriate.

Crom provides an *application-agnostic* framework for speculative execution. This generality allows a wide variety of programs to benefit from Crom's services. However, as a consequence of this generality, Crom requires some developer guidance to ensure correctness and to prevent speculative activity from consuming too many resources. In particular:

- Speculating on all possible user inputs is computationally infeasible. Thus, Crom relies on the developer to constrain the speculation space and suggest reasonable speculative inputs for a particular application state (§4.1.1 and §5.1.3).
- Speculative execution should only occur when the application has idle resources, e.g., during user think time. Crom's speculations are explicitly initiated by the developer, and Crom trusts the developer to only issue speculations when resources would otherwise lie fallow.
- A speculative context should only be committed if it represents a realizable outcome for the current application state. In particular, the *initial* state of a committing speculative context must have been equal to the *current* state of the application. This guarantees that the speculative event handler did the same things that its non-speculative version would do if executed now. We call this safety principle *start-state equivalence*. Crom could automatically check for this in an application-agnostic way by bit-comparing the current browser state with the initial state for the speculative context. However, performing such a comparison would often be expensive. Thus, Crom requires the developer to leverage application knowledge and define an equivalence function that determines whether a speculative context is appropriate to commit for a given application state (§4.1.1 and §5.1.3).
- A client-side event handler may generate writes to client-side state *and* server-side state. The developer must ensure that client-side speculation is read-only with respect to server state, or that server-side updates issued by speculative code can be undone, e.g., by resetting the "message read" flags on an email server (§ 4.1.2).

Writing speculative code is inherently challenging. Crom hides some of the implementation complexity, but it does not completely free the developer from reasoning about speculative operations. We believe that Crom strikes the appropriate balance between the competing tensions of correctness, ease of use, and performance.

Crom ensures correctness by rewriting speculative code to only touch shadow state, and by using developer-defined equivalence functions to determine commit safety. Crom provides ease of use through its generic speculation API. With respect to performance, Crom's goal is *not* to be as CPU-efficient as custom speculative code. Indeed, Crom's speculative call trees will generally be slower than hand-crafted speculation code. Such custom code has no rewriting or cloning overhead, and it can speculate in a targeted way, eliding the code in an event handler call chain that is irrelevant to, say, warming a cache. However, Crom's speculations only need to be "fast enough"—they must fit within user think time, be quick with respect to network latencies, and not disturb foreground computations. If these conditions are satisfied, Crom's computational inefficiency relative to custom speculation code will be moot, and Crom's speculations will mask essentially as much network latency as hand-coded speculations would.

In addition to processor cycles, speculative activity requires network bandwidth to exchange data with servers. Crom does not seek to reduce this inherent cost of speculation. As with hand-crafted speculative solutions, developers must be mindful of network overheads and be judicious in how many Crom speculations are issued.

Figure 1 lists the primary Crom API. We discuss this API in greater detail in the next two sections. Most of the technical challenges lie with the implementation of `Crom.makeSpeculative()`, which allows an application to define speculable user actions.

[1] This description applies to the DOM Level 0 event model. The DOM Level 2 model is also common, but it has incompatible semantics across browsers. We do not discuss it in this paper.

`Crom.makeSpeculative(DOMnode,eventName, mutator,mutatorArgs,stateSketch, DOMsubtree)`	Register `DOMnode.eventName` as a speculable event handler (§4.1 and §5.1). The `mutator`, `mutatorArgs`, and `stateSketch` arguments constrain the speculation space and define the equivalence classes for commits (§4.1.1 and §5.1.3). `DOMsubtree` defines a speculation zone (§5.5.2).
`Crom.autoSpeculate`	Boolean which determines whether Crom should automatically respeculate after committing a prior speculation (§4.1.1).
`Crom.forceSpeculations()`	If `autoSpeculate` is `false`, this method forces Crom to issue pending speculations.
`Crom.maxSpeculations(N)`	Limits number of speculations Crom issues (§5.6).
`Crom.createContextPool(N, stateSketch, DOMsubtree)`	Proactively make `N` speculative copies of the current browser context; tag them using the `stateSketch` function (§5.5.3).
`Crom.rewriteCG(f)`	Rewrite a closure-generating function to make it amenable to speculation (§5.1.5).
`Crom.prePOST(formInput, specUploadDoneCallback)`	Collaborates with Crom's server-side component to make a form element speculatively upload data (§4.2 and §5.4).
`Crom.cacheAdd(key, AJAXdata)` `Crom.cacheGet(key)`	Used to cache speculatively fetched AJAX data that would otherwise be ignored by the regular browser cache (§4.1.2 and §5.3).

Figure 1: Crom API calls and configuration settings.

4 Speculation Opportunities & Techniques

In this section, we describe four ways that speculative execution can reduce latency in rich Internet applications. We also explain how to leverage the Crom API from Figure 1 to exploit these speculative opportunities.

4.1 Simple Prefetching

When a user triggers a heavyweight state change in a RIA, the browser does four things. First, it executes JavaScript code associated with an event handler. In turn, this code fetches data from the local browser cache or external web servers. Once the content is fetched, the browser determines the new display layout for the page. Finally, the browser draws the content on the screen.

Pulling data across the network is typically the slowest task, so warming the browser cache by speculatively prefetching data can dramatically improve user-perceived latencies. For example, a photo gallery or an interactive map application can prefetch images to avoid synchronous fetches through a high-latency or low-bandwidth network connection. As another example, consider the DHTMLGoodies tab manager [12], which allows applications to create tab pane GUIs. When the user clicks a "new tab" link, the tab manager issues an AJAX request to fetch the new tab's content. When the AJAX request completes, a callback dynamically creates a new display tab and inserts the returned HTML into the DOM tree. Figure 2 demonstrates how to make this operation speculative. When the application calls `Crom.makeSpeculative()`, Crom automatically makes a shadow copy of the current browser state and rewrites the `onclick` event handler, creating a speculative version that accesses shadow state. When the application calls `Crom.forceSpeculations()`, Crom runs the rewritten handler in the hidden context, fetching the AJAX data and warming the real browser cache.

```
<div id='tab-container'>
   <div class='dhtmlgoodies_aTab'>
      This is the initial tab.
      <a href='#' id='loadLink'>
          Click to load new tab.
      </a>
   </div>
</div>
<script>
var link = document.getElementById('loadLink');
link.onclick = function(){
   //Invoke DHTMLGoodies API to make new tab
   createNewTab('http://www.foo.com');
};
Crom.makeSpeculative(link, 'onclick');
Crom.forceSpeculations();
</script>
```

Figure 2: Creating a new tab GUI element.

Once the browser cache has been warmed, Crom can discard the speculative context. However, saving the context for future committing provides greater reductions in user-perceived fetch latencies (§4.3).

4.1.1 Speculating on Multi-valued Inputs

In the previous example, an application speculated on an input element with one possible outcome, i.e., being clicked. Other input types can generate multiple speculable outcomes. For example, a list selector allows one of several options to be chosen, and a text box can accept arbitrary character inputs.

To speculate on a multi-valued input, an application passes a *mutator function*, a *state sketch function*, and a vector of N *mutator argument sets* to `Crom.makeSpeculative()`. Crom makes N copies of the current browser state, and for each argument set, Crom runs the rewritten mutator with that argument set in the context of a shadow browser environment. This generates N distinct contexts, each representing a different speculative outcome for the input element. Crom passes each context to the sketch function, generating an application-defined signature string for that context. Crom tags each context with its unique sketch and then runs the speculative event handler in each context, saving the modified domains. Later, when the user actually generates an event, Crom determines the sketch for the current, non-speculative application state. If Crom has a speculative context with a matching tag, Crom can safely commit that context due to start-state equivalence.

Figure 3 shows an example of how Crom speculates on multi-valued inputs. In this example, we use the autocompleting text box from the popular script.aculo.us JavaScript library [21]. The first two lines of HTML define the text input and the button which triggers a data fetch based on the text string. We create a custom JavaScript object called `acManager` to control the autocompletion process and register it with the script.aculo.us library. The library invokes `acManager.customSelector()` whenever the user generates a new input character. Once the user has typed five characters, `acManager` uses AJAX to speculatively fetch the data associated with each suggested autocompletion. The state sketch function simply returns the value of the search text—a speculative context is committable if its search text is equivalent to that of the real domain.

Note that the code sets `Crom.autoSpeculate` to `false`, indicating that Crom should not automatically respeculate on the handler when a prior speculation for the handler commits. The autocompletion logic explicitly forces speculation when the user has typed enough text to generate completion hints.

```
<input id='sText' type='text' />
<button id='sButton'>Search</button>
<script>
var sText = document.getElementById('sText');
var sButton = document.getElementById('sButton');
sButton.onclick = function(){
   updateDisplayWithAJAXdata(sText.value);
};

Crom.autoSpeculate = false;
function mutator(arg){ //Set value of text input
   sText.value = arg;
}
function sketch(ctx){
   var doc = ctx.document;
   var textInput = doc.getElementById('sText');
   return textInput.value;
}
var acManager = {
   getHints: function(textSoFar){
      //Logic for generating autcompletions;
      //returns an array of strings
   },
   customSelector: function(textSoFar){
      if(textSoFar.length == 5){
         var hints = this.getHints(textSoFar);
         Crom.makeSpeculative(sButton,
            'onclick',mutator,hints,sketch);
         Crom.forceSpeculations();
      }
      //Display autocompletions to user...
   }
};
new Autocompleter('sText', acManager);
</script>
```

Figure 3: Speculating on an autocompletion event.

4.1.2 Separating Reads and Updates

In some web applications, pulling data from a server has side effects on the client *and* the server. In these situations, speculative computations must not disturb foreground state on either host. For example, the Decimail webmail client [5] uses AJAX to wrap calls to an IMAP server. The `fetchNewMessage()` operation updates client-side metadata (e.g., a list of which messages have been fetched) and server-side metadata (e.g., which messages should be marked as seen).

To speculate on such a read/write operation, the developer must explicitly decompose it into a read portion and a write portion with respect to server state. For example, to add Crom speculations to the Decimail client, we had to split the preexisting `fetchNewMessage()` operation into a read-only `downloadMessage()` and a metadata-writing `markMessageRead()`. The read-only operation downloads an email from the server, but specifies in the IMAP request that the server should not mark the message as seen. The `markMessageRead()` tells the server to update this flag, effectively committing the message fetch on the server-side. Inside `fetchNewMessage()`, the call

```
<form action='server.com/recv.py' method='post'>
   <div>
      <label>File 1:</label>
      <input type='file' id='fInput'/>
   </div>
   <div>
      <input type='submit' value='Send data!'/>
   </div>
</form>

<script>
var fInput = document.getElementById('fInput');
Crom.speculativePOST(fInput,
           function(){alert('File uploaded!')});
</script>
```

Figure 4: Making a POST operation speculative.

to `markMessageRead()` is conditioned on whether `fetchNewMessage()` is running speculatively; code can check whether this is true by reading the special `Crom.isSpecEx` boolean.

Although `downloadMessage()` may be read-only with respect to the server, it may update client-side JavaScript state. So, when speculating on `fetchNewMessage()`, we run `downloadMessage()` in a speculative execution context. In speculative or non-speculative mode, `downloadMessage()` places AJAX responses into a new cache provided by Crom. Later, when `downloadMessage()` runs in non-speculative mode, it checks this cache for the message and avoids a refetch from the server.

Like the regular browser cache, Crom's AJAX cache persists across speculations (although not application reloads). The regular cache will store AJAX results containing "expires" or "cache control" headers [6], so an application-level AJAX cache may seem superfluous. However, some AJAX servers do not provide caching headers, making it impossible to rely on the regular cache to store AJAX data if the client side of the application is developed separately from the server side. Examples of such scenarios include mash-ups and aggregation sites.

4.2 Pre-POSTing Uploads

Prefetching allows Crom to hide download latency. However, in some situations, such as Decimail's attach-file function, the user is stalled by upload (HTTP POST) delays. To hide this latency, Crom's client and server components cooperate to create a *POST cache*. When a user specifies a file to send, Crom speculates that the user will later commit the send. Crom asynchronously transfers the data to the server's POST cache. Later, if the user commits the send, the asynchronous POST will be finished (or at least already in-progress).

Figure 4 demonstrates how to make a POST operation speculative. The web application simply registers the relevant `input` element with the Crom library. Once the user has selected a file, Crom automatically starts uploading it to the server. When the speculative upload completes, Crom invokes an optionally provided callback function; this allows the application to update foreground (i.e., non-speculative) GUI state to indicate that the file has safely reached the server.

Speculative uploading is not a new technique, and it is used by several popular services like GMail. Crom's contribution is providing a generic framework for adding speculative uploads to non-speculative applications.

4.3 Saving Client Computation

When an application updates the screen, the browser uses CPU cycles for *layout* and *rendering*. During layout, the browser traverses the updated DOM tree and determines the spatial arrangement of the elements. During rendering, the browser draws the laid-out content on the screen. Speculative cache warming can hide fetch latency, but it cannot hide layout or rendering delays.

Crom stores each speculative browser context inside an invisible `<iframe>` tag. As a speculative event handler executes, it updates the layout of its corresponding `iframe`. When the handler terminates, Crom saves the already laid-out `iframe`. Later, if the user generates the speculated-upon event, Crom commits the speculative DOM tree in the `iframe` to the live display, paying the rendering cost but avoiding the layout cost. The result is a visibly smoother page load.

4.4 Server Load Smoothing

Some client delays are due not to network delays, but to congestion at the server due to spiky client loads. Using selective admission control at the server, speculative execution spreads client workload across time, just as speculation plus differentiated network service smooths peak network loads [3]. When the server is idle, speculations slide requests earlier in time, and when the server is busy, speculative requests are rejected and the associated load remains later in time. This paper does not explore server smoothing further, but the techniques described above, together with prioritized admission control at the server, should adequately expose this opportunity.

5 Implementation

The client-side Crom API could be implemented inside the browser or by a regular JavaScript library. For deployability, we chose the latter option. In this section, we describe our library implementation and the optimizations needed to make it performant.

5.1 Making Event Handlers Speculative

To create a speculative version of an event handler bound to DOM node `d`, an application calls `Crom.makeSpeculative(d, eventName)`; Figures 2 and 3 provide sample invocations of this function.

The `makeSpeculative()` method does two things. First, it creates a shadow browser context for the speculative computation. Second, it creates a new event handler that performs the same computation as the original one, but reads and writes from the speculative context instead of the real one. We discuss context cloning and function rewriting in detail below.

5.1.1 The Basics of Object Copying

Crom clones different types of objects using different techniques. For primitive values, Crom just returns the value. For built-in JavaScript objects like `Dates`, Crom calls the relevant built-in constructor to create a semantically equivalent but referentially distinct object.

JavaScript functions are first-class objects. Calling a function's `toString()` method returns the function's source code. To clone a function `f`, Crom calls `eval(f.toString())`, using the built-in `eval()` routine to parse the source and generate a semantically equivalent function. Like any object, `f` may have properties. So, after cloning the executable portion of `f`, Crom uses a `for-in` loop to discover `f`'s properties, copying primitives by value and objects using deep copies.

To clone a non-function object, Crom creates an initially empty object, finds the source object's properties using a `for-in` loop, and copies them into the target object as above. Since object graphs may contain cycles, Crom uses standard techniques from garbage collection research [11] to ensure that each object is only copied once, and that the cloned object graph is isomorphic to the real one.

To clone a DOM tree with a root node `n`, Crom calls the native DOM method `n.cloneNode(true)`, where the boolean parameter indicates that `n`'s DOM children should be recursively copied. The `cloneNode()` method does not copy event handlers or other application-defined properties belonging to a DOM node. Thus, Crom must copy these properties explicitly, traversing the speculative DOM tree in parallel with the real one and updating the properties for each speculative node. Non-event-handler properties are deep-copied using the techniques described above. Since Crom rewrites handlers and associates special metadata with them, Crom assumes that user-defined code does not modify or introspect event handlers. So, Crom shallow-copies event handlers by reference.

5.1.2 Cloning the Entire Browser State

To clone the whole browser context, Crom first copies the real DOM tree. Crom then creates an invisible `<iframe>` tag, installing the cloned DOM tree as the root tree of the `iframe`'s `document` object. Next, Crom copies the application heap, which is defined as all JavaScript objects in the global namespace and all objects reachable from those roots. Crom discovers the global properties using a `for-in` loop over `window`. Crom deep-copies each of these properties and inserts the cloned versions into an initially empty object called `specContext`. `specContext` will later serve as the global namespace for a speculative execution.

Global properties can be referenced with or without the `window.` prefix. To prevent `window.globalVar` from falling through to the real `window` object, Crom adds a property to `specContext` called `window` that points to `specContext`. Crom also adds a `specContext.document` property that points to the hidden `<iframe>`'s `document` object. As we explain in Section 5.1.4, this forces DOM operations in the speculative execution to touch the speculative DOM tree instead of the real one.

5.1.3 Commit Safety & Equivalence Classes

As described so far, a shadow context is initialized to be an exact copy of the browser state at clone time. This type of initialization has an important consequence: it prevents us from speculating on user intents that are not an immediate extension of the current browser state. For example, a text input generates an `onchange` event when the user types some characters and then shifts input focus to another element. If the text input is empty when Crom creates a speculative domain, Crom can speculatively determine what the `onchange` handler would do when confronted with an empty text box. However, if Crom creates shadow contexts as exact copies of the current browser state, Crom has no way to speculate on what the handler would do if the user had typed, say, "valhalla" into the text input.

To address this problem, we allow applications to provide three additional arguments to `Crom.makeSpeculative()`: a *mutator function*, a *mutator argument vector*, and a *state sketch function*. Section 4.1.1 provides an overview of these parameters. Here, we only elaborate on the sketch function and its relationship to committability.

The sketch function accepts a global namespace, speculative or real, and returns a unique string identifying the salient application features of that name space. Each speculative context is initially a perfect copy of the real browser context at time t_0. Thus, at t_0, before any speculative code has run, the new speculative context has the same sketch as the real context. Crom tags each speculative context with the state sketch of its source context. Later, at time t_1, when Crom must decide whether the speculative context is committable, it calculates the state sketch for the real browser context at t_1. A speculative context is only committable if its sketch tag matches the sketch for the current browser context. This ensures that

the speculative context started as a semantically equivalent copy of the current browser state, and therefore represents the appropriate result for the user's new input.

State sketches provide a convenient way for applications to map semantically identical but bit-different browser states to a single speculable outcome. For example, the equivalence function in Figure 3 could canonicalize the search strings `blue\tbook` and `blue\t\tbook` to the same string.

5.1.4 Rewriting Handlers

After creating a speculative browser context, Crom must create a speculative version of the event handler, i.e., one that is semantically equivalent to the original but which interacts with the speculative context instead of the real one. To make such a function, Crom employs JavaScript's `with` statement. Inside a `with(obj){...}` statement, the properties of `obj` are pushed to the front of the name resolution chain. For example, if `obj` has a property `p`, then references to `p` touch `obj.p` rather than a globally defined `p`.

To create a speculative version of an event handler, Crom fetches the handler's source code by calling its `toString()` method. Next, Crom alters the source code string, placing it inside a `with(specContext)` statement. Finally, Crom uses `eval()` to generate a compiled function object. When Crom executes the new handler, each handler reference to a global property will be directed to the cloned property in `specContext`.

The `with()` statement binds lexically, so if the original event handler calls other functions, Crom must rewrite those as well. Crom does this lazily: for every function or method call `f()` inside the original handler, Crom inserts a new variable declaration `var __rewritten_f = Crom.rewriteFunction(f, specContext);`, and replaces calls to `f()` with calls to `__rewritten_f()`.

The `document` object mediates application access to the DOM tree. `specContext.document` points to the shadow DOM tree, so speculative DOM operations can only affect speculative DOM state. Since `document` methods do not touch application heap objects, Crom does not need to rewrite them.

The names of function parameters may shadow those of global variables. Speculative references to these variables should *not* resolve to `specContext`. When rewriting functions with shadowed globals, Crom passes a new speculative scope to `with()` statements; this scope is a copy of `specContext` that lacks references to shadowed globals.

If speculative code creates a new global property, it may slip past the `with` statement into the real global namespace. Fortunately, JavaScript is single-threaded, so Crom can check for new globals after the speculative handler has finished and sweep them into `specContext` before they are seen by other code.

When speculative code `deletes` a global property, this only removes the property from `specContext`. When the speculation commits, this property must also be deleted from the global namespace. To accomplish this, Crom rewrites `delete` statements to additionally collect a list of deleted property names. If the speculation later commits, Crom removes these properties from the real global namespace.

5.1.5 Externally Shared Closures

Whenever a function is created, it stores its lexical scope in an implicit activation record, using that scope for subsequent name resolution. Unfortunately, these activation records are not introspectable. If they escape cloning, they become state shared with the real (i.e., non-speculative) browser context. To avoid this fate, Crom rewrites closures to use explicit activation objects. Later, when Crom creates a speculative context, it can clone the activation object using its standard techniques.

Consider the code in Figure 5, which creates a closure and makes it the event handler for two different buttons. During non-speculative execution, clicking either button updates the same counter inside the shared closure. However, `closureFunc.toString()` merely returns "function (){alert(x++);}", with no mention of `x`'s closure binding. Using the rewriting techniques described so far, a rewritten handler would erroneously look for `x` in the global scope.

```
function genClosure(x){
   function f(){alert(x++);}
   return f;
}
var closureFunc = genClosure(0);
button0.onclick = closureFunc;
button1.onclick = closureFunc;
```

Figure 5: Variable `x` persists in a hidden closure scope.

```
function genClosure(x){
   var __cIndex = Crom.newClosureId();
   __closureEnvironment[__cIndex].x = x;
   function f(){
     alert(__closureEnvironment[__cIndex].x++);
   }
   f.__cIndex = __cIndex;
   return f;
}
```

Figure 6: The rewritten closure scope is explicit.

Figure 6 shows `genClosure()` rewritten to use an explicit activation record. `Crom.newClosureId()` creates an empty activation record, pushes it onto a global array `__closureEnvironments`, and returns its index. Crom rewrites each property that implicitly references the closure scope to explicitly reference the activation record via a function property `__cIndex`.

Later, when rewriting the `button0` or `button1` event handler, Crom detects that the handler is a closure by its `__cIndex` property. If the value of `f.__cIndex` is, say, 2, Crom rewrites the closure function by adding the statement `var __cIndex = 2;` immediately before the `with(specContext)` in the string passed to `eval()`. This gives the rewritten handler enough state to access the proper explicit activation record. Like any other global, `__closureEnvironments` is speculatively cloned, so each speculative execution has a private snapshot of the state of all closures.

Currently, applications must explicitly invoke Crom to rewrite functions which return closures. They do this by executing `g = Crom.rewriteCG(g)` for each closure-generating function `g`. Future versions of Crom will use lexical analysis to perform this rewriting automatically.

5.2 Committing Speculative State

Committing a speculative context is straightforward. First, Crom updates the DOM tree root in the non-speculative context, making it point to the DOM tree in the committing speculation's hidden `iframe`. Next, Crom updates the heap state. A `for-in` loop enumerates the heap roots in `specContext` and assigns them to the corresponding properties in the real global name space; this moves both updated and newly created roots into place. Finally, Crom iterates through the list of global properties deleted by the speculative execution and removes them from the real global name space.

5.3 AJAX Caching

To support the caching of AJAX results, client-side programs call the Crom methods `Crom.cacheAdd(key, AJAXresult)` and `Crom.cacheGet(key)`. These methods allow applications to associate AJAX results with arbitrary cache identifiers. Like the regular browser cache, Crom's AJAX cache is accessible to speculative and non-speculative code. For example, in the modified Decimail client, the event handler for the "fetch new message" operation is broken into two functions, `downloadMessage()` and `markMessageRead()`. `downloadMessage()` is read-only on the server side, but it modifies client-side state, e.g., a JavaScript array that contains metadata for each fetched message. Thus, when the Decimail client speculates on a message fetch, it rewrites `downloadMessage()`'s call tree and runs it in a speculative context. The speculative `downloadMessage()` looks for the message in Crom's cache and does not find it. It fetches the new email using AJAX and inserts it into Crom's cache. Later, when the user actually triggers the "fetch new message" handler, `downloadMessage()` runs in non-speculative mode and finds the requested email in the cache. Decimail then calls `markMessageRead()` to inform the server of the user's action.

5.4 Speculative Uploads

An upload form typically consists of a *file input* text box and an enclosing *submit form*. After the user types a file name into the text box, the input element generates an `onchange` event. However, the file is not uploaded to the server until the user triggers the `onsubmit` event of the enclosing form, typically by clicking a button inside the form. At this point, the application's `onsubmit` handler is called to validate the file name. Unless this handler returns `false`, the browser POSTs the form to the server and sends the server's HTTP response to the target of the form. By default, the target is the current window; this causes the browser to overwrite the current page with the server's response.

Crom implements speculative uploads with a client/server protocol. On the client, the developer specifies which file input should be made speculative by calling `Crom.prePost(fileInput,callback)`. Inside `prePost()`, Crom saves a reference to any user-specified `onsubmit` form-validation handler, since Crom will supply its own `onsubmit` handler shortly. Crom installs an `onchange` event handler for the file input which will be called when the user selects a file to upload. The handler creates a cloned version of the upload form in a new invisible `iframe`, with all file inputs removed except the one representing the file to speculatively upload. If the application's original `onsubmit` validator succeeds, Crom's `onchange` handler POSTs the speculative form to a server URL that only accepts speculative file uploads. Crom's server component caches the uploaded file and its name, and the client component records that the upload succeeded.

`Crom.prePost()` also installs an `onsubmit` handler that lets Crom introspect the form before a real click would POST it. If Crom finds a file that has already been cached at the server, Crom replaces the associated file input with an ordinary text input having the value `ALREADY_SENT:`*filename*. Upon receipt, Crom's server component inserts the cached file data before passing the form to the application's server-side component.

The interface given above is least invasive to the application, but a speculation-aware application can provide upload progress feedback to the user by registering a progress callback with Crom. Crom invokes this handler in the real domain when the speculative upload completes, allowing the application to update its GUI.

5.5 Optimizations

To conclude this section, we describe three techniques for reducing speculative cloning overheads.

5.5.1 Lazy Cloning

For complex web sites, *eager cloning* of the entire application heap may be unacceptably slow. Thus, Crom offers a *lazy cloning* mode in which objects are only copied when a speculative execution is about to access them. Since this set of objects is typically much smaller than the set of all heap objects, lazy cloning can produce significant savings.

In lazy mode, Crom initially copies only the DOM tree and the heap variables referenced by DOM nodes. As the speculative computation proceeds, Crom dynamically rewrites functions as before. However, object cloning is now performed as a side effect of the rewriting process. Crom's lexical analysis identifies which variable names refer to locals, globals, and function parameters. Locals do not need cloning. Strictly speaking, Crom only needs to clone globals that are written by a speculative execution; reads can be satisfied by the non-speculative objects. However, a global that is read at one point in a call chain may be passed between functions as a parameter and later written. To avoid the bookkeeping needed to track these flows, Crom sacrifices performance for implementation simplicity and clones a global variable whenever a function reads or writes it. The function is then rewritten using the techniques already described. Function parameters need not be cloned as such—if they represent globals, they will be cloned by the ancestor in the call chain that first referenced them.

Lazy cloning may introduce problems at commit time. Suppose that global objects named `X` and `Y` have properties `X.P` and `Y.P` that refer to the same underlying object `obj`. If a speculative call chain writes to `X.P`, Crom will deep-copy `X`, cloning `obj` via `X.P`. The speculation will write to the new clone `obj'`. If the call chain never accesses `Y`, `Y` will not be cloned since it is not reachable from the object tree rooted at `X`. Later, if the speculation commits, Crom will set the real global `X` to `specContext.X`, ensuring that the real `X.P` points to `obj'`. However, `Y.P` will refer to the original (and now stale) `obj`.

In practice, we have found that such stale references arise infrequently in well-designed, modular code. For example, the autocompletion widget is a stand-alone piece of JavaScript code. When a developer inserts a speculative version of it into an enclosing web page, the widget will not be referenced by other heap variables, and running in lazy mode as described will not cause stale child references. Regardless, to guarantee correctness at commit time, Crom provides a *checked lazy mode*. Before Crom issues any speculative computations, it traverses every JavaScript object reachable from the heap roots or the DOM tree, and annotates each object with `__parents`, a list of parent pointers. A parent pointer identifies the parent object and the property name by which the parent references the child. Crom copies `__parents` by reference when cloning an object. When a lazy speculation commits, Crom uses the `__parents` list to update stale child references in the original heap.

Crom also has an *unchecked lazy mode* in which Crom clones lazily but assumes that stale child references never occur, thereby avoiding the construction and the checking of the parent map. For applications with an extremely large number of objects and/or a highly convoluted object graph, unchecked lazy mode may be the only cloning technique that provides adequate performance. We return to this issue in the evaluation section. For now, we make three observations. First, our experience has been that the majority of event handlers will run safely in unchecked mode without modification. Second, Crom provides an interactive execution mode that allows developers to explicitly verify whether their speculative event handlers are safe to run in unchecked lazy mode. During speculative commits in interactive mode, Crom reports which committing objects have parents that were not lazily cloned and thus would point to stale children post-commit. Crom automatically determines the object tree roots that the programmer must explicitly reference in the event handler to ensure that the appropriate objects are cloned. The programmer can then perform this simple refactoring to make the handler safe to run in unchecked lazy mode.

Third, and most importantly, Section 6 shows that most of Crom's benefits arise from speculatively warming the browser cache. Committing speculative DOM nodes and heap objects can mask some computational latency, but network fetch penalties are often much worse. Thus, if speculating in checked lazy mode is too slow, or checked mode refactoring is too painful, an application can pass a flag (not shown in Figure 1) to `Crom.makeSpeculative()` which instructs Crom to discard speculative contexts after the associated executions have terminated. In this manner, applications can use unchecked lazy speculations solely to warm the browser cache, forgoing complications due to commit issues, but deriving most of the speculation benefit.

5.5.2 Speculation Zones

An event handler typically modifies a small fraction of the total application heap. Similarly, it often touches a small fraction of the total DOM tree. Lazy cloning exploits the first observation, and *speculation zones* exploit the second. An application may provide an optional `DOMsubtree` parameter to `Crom.makeSpeculative()` that specifies the root of the DOM subtree that an event handler modifies. At speculation time, Crom will only clone the DOM nodes associated with this branch of the tree. At commit time, Crom will splice in the speculative DOM branch but leave the rest of the DOM tree undisturbed.

Speculation zones are useful when an event handler is tightly bound to a particular part of the visual display. For example, the autocompletion widget in Figure 3 only modifies the `<div>` tag that will hold the fetched search results. Speculation zones are also useful when the DOM tree contains rich objects whose internal state is opaque to the native `cloneNode()` method. Examples of such objects are Flash movies and Java applets. When `cloneNode()` is invoked upon such an object, the returned clone is "reset" to the initial state of the source object; for example, a Flash movie is rewound to its first frame. Since it is difficult for JavaScript code to reason about the state of such objects, applications can use speculation zones to "speculate around" these opaque parts of the DOM tree.

5.5.3 Context Pools

Crom hides speculative computations within the "think time" of a user. The shorter the think time, the faster Crom must launch speculations to reduce user-perceived fetch latencies. For example, time pressure is comparatively high for the autocompletion widget (§4.1.1) since a fast typist may generate her search string quickly.

To reduce the synchronous overhead of issuing new speculations, applications can call `Crom.createContextPool(N, stateSketch, DOMsubtree)`. This method generates and caches `N` clones of the current browser environment, tagging each with the sketch value generated when `stateSketch` is passed the current browser context.

Using context pools, the entire cost of eager cloning or the initial cost of lazy cloning is paid in advance. At speculation time, Crom finds an unused context that is tagged with the application's current sketch. Crom immediately specializes it using a mutator and issues the new speculation.

5.6 Limitations and Future Work

Many of the limitations of the current Crom prototype arise from the fact that the client-side portion runs as slow JavaScript code instead of fast C++ code inside the browser. For example, we pursued the optimizations described in Section 5.5 after discovering that in many cases, copying the entire browser context using JavaScript code would result in unacceptably slow performance. Ideally, browsers would natively support context cloning, and applications could always use checked mode speculations without fear of excessive CPU usage.

Speculative fetches compete with non-speculative fetches for bandwidth. A native implementation of Crom could measure the traffic across multiple flows and prioritize non-speculative fetches over speculative ones. Our JavaScript implementation cannot measure such browser-wide network statistics. However, developers can use `Crom.maxSpeculations(N)` to place an upper limit on the number of speculations that can be triggered by a single call to `Crom.forceSpeculations()`.

As mentioned in Section 5.5.2, opaque browser objects like applets and Flash applications cannot be introspected by JavaScript code. Our JavaScript Crom library can only clone them in a crude fashion by recreating their HTML tags (and thereby reinitializing the clones to a virgin state). In practice, we expect most applications to avoid this issue by using speculation zones. However, some applications might benefit from the ability to clone rich objects in more sophisticated ways. An in-browser implementation of Crom could do this, but we leave an exploration of these mechanisms for future work.

The parsing engine for our client-side rewriter is fairly unsophisticated. Implemented with regular expressions, it is sufficient for analyzing the real applications described in the next section. However, it does not parse the complete JavaScript grammar. Future versions of the client-side library will use an ANTLR-driven parser [18]. An in-browser implementation could obviously reuse the browser's native parsing infrastructure.

6 Evaluation

Ideally, we would evaluate Crom by taking a preexisting application that has custom speculation code, replacing that code with Crom calls, and comparing the performance of the two versions. Unfortunately, custom speculation code is often tightly integrated with the rest of the application's code base, making it difficult to remove the code in a principled way and provide a fair comparison with Crom's speculation API. Thus, our evaluation explores the performance of the real applications from Section 4 that we modified to use the Crom API. We show that Crom's computational overheads are hideable within user think time, and that Crom can reduce user-perceived fetch latencies by an order of magnitude under realistic network conditions.

Our Crom prototype has been tested most extensively on the Firefox 3.5 browser. Its core functionality has also been tested on IE7 and Safari 4.0. However, we are still fixing compatibility issues with the latter set of browsers, so this section only contains Firefox results.

6.1 Performance of Modified Applications

To test the speculative autocompletion widget and tab manager, we downloaded real web pages onto our local web server; Figure 7 lists the pages that we examined. We inserted the Crom library and the speculative applications into the pages, then loaded the pages using a browser to test their performance. Decimail is a standalone application, so we tested it by itself, i.e., we did not

Web site	Primitives	Objects	Functions	DOM nodes
Google	16	5	31	77
Gmail-o	2574	622	6	66
Gmail-i	0	0	3	1999
ESPN	1053	414	596	1422
YouTube	329	119	894	823
Live	49	8	80	184
Yahoo	150	26	51	694
MySpace	1447	236	904	499
eBay	826	589	1360	633
MSN	227	87	210	932
Amazon	13112	2895	1145	3003
CNN	859	129	588	1733

Figure 7: The types of JavaScript variables in several popular web pages. Gmail-o refers to Gmail's outer control frame and Gmail-i refers to the inner frame which displays the inbox message list.

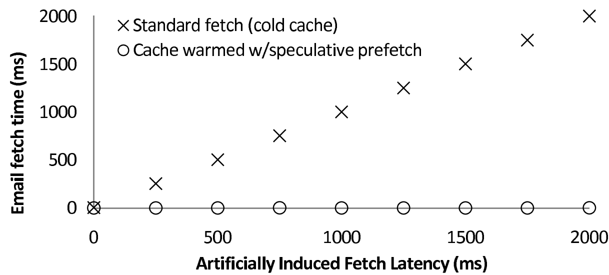


Figure 8: Latency reductions using Crom's AJAX cache.

embed it within an enclosing web page. Stripped of comments and extraneous whitespace, Crom's JavaScript code added 65 KB to each application's download size. All experiments ran on an HP xw4600 workstation with a dual-core 3GHz CPU and 4 GB of RAM. Web content was fetched from a custom localhost web server that introduced tunable fetch delays. This allowed us to measure Crom's latency-hiding benefits as a function of the fetch penalty. All experiments represent the average of 10 trials. Standard deviations were less than 6% in all cases.

6.1.1 Decimail Client

The modified Decimail client [5] used Crom's AJAX cache to store results from speculative mail fetches. Running in unchecked lazy mode, Decimail's cloning and rewriting overheads were less than 5 ms per fetch, so we elide further discussion of them. Figure 8 shows the benefit of finding a requested message in the Crom cache instead of having to fetch it synchronously. Unsurprisingly, a cache hit took no more than 3 ms to serve, whereas the cache miss penalty was the fetch latency to the server.

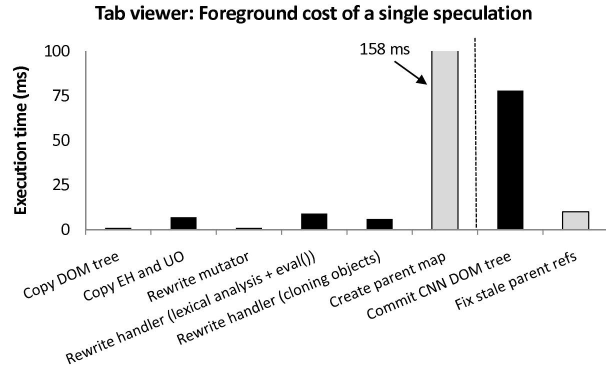


Figure 9: Tab manager pre- and post-commit overheads (checked mode costs in grey).

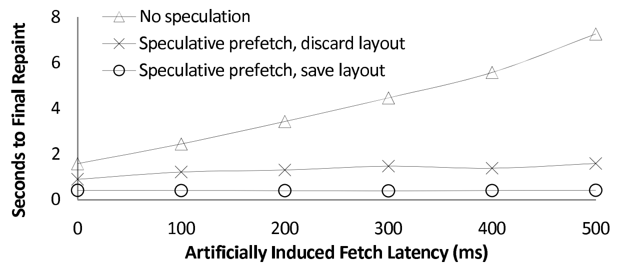


Figure 10: User-perceived latencies for tab manager.

6.1.2 Tab Manager

Figure 9 depicts the speculative overheads in the tab manager application [12]. In this experiment, the tab manager was embedded within the ESPN front page. Clicking the "make new tab" button generated an AJAX request for a web page. Once fetched, the page was rendered inside a <div> tag; this tag was a child of the enclosing <div> tree controlled by the tab manager. Crom used this enclosing tag as the speculation zone. We configured the tab manager to speculatively fetch the CNN home page, a complex page with over 1700 DOM nodes.

Figure 9 breaks the speculation overheads into pre-commit costs (the left side of the dotted line) and during-commit costs (the right side of the line). The black bars represent the overheads for unchecked lazy mode. The grey bars represent the additional costs that would arise from running lazy cloning in checked mode.

Unchecked lazy mode: When speculating in unchecked lazy mode, aggregate pre-commit speculation costs were low, totalling 24 ms. Copying the speculated-upon DOM tree was fast (1 ms), since the speculation zone consisted of a small <div> tree of depth 3. Walking the cloned DOM tree and copying event handlers and user-defined objects took only 7 ms. There was no mutator to rewrite, but Crom did have to rewrite an event handler call tree with a maximum depth of 4. Figure 9 breaks the rewriting cost into two parts.

The first part represents the cost of lexical analysis, source code modification, and calling `eval()` on the modified source. The second part represents the cost of copying heap objects during the rewriting process. Both costs were less than 10 ms each since the call chain touched a minority of the total page state.

At commit time, there were no stale child references to fix since Crom was running in unchecked mode. Committing speculative heap objects took less than 1 ms, since Crom merely had to make global variables in the real domain point to speculative object trees. Over 99% of the commit overhead was generated by the splice of the speculative DOM subtree into the non-speculative one. Although the splice was handled by native code, it required a screen redraw, since the formerly invisible `CNN` context was now visible. Screen redraws are one of the most computationally intensive browser activities. However, Crom avoided the additional (and more expensive) *reflow* cost, since the layout for the `CNN` content was determined during the speculative execution. Since even non-speculative computations must pay the redraw cost upon updating the screen, Crom added less than 1 ms to the inherent cost of displaying the new content.

Checked lazy mode: In this mode, Crom built a parent map before issuing any speculations and checked for stale object references at commit time. The grey bars in Figure 9 depict these costs, which must be paid in addition to the rewriting and cloning costs in black. Constructing the parent mapping took 158 ms, leading to an overall pre-commit cost of 182 ms. Although the mapping cost is amortizable across multiple speculations, an aggregate CPU overhead of 182 ms pushes against the limits of acceptability. The current implementation of Crom does not stage its pre-commit operations, i.e., Crom holds the processor for the entire duration of the cloning, rewriting, and parent mapping process. Depending on the application, a pre-commit overhead of 182 ms may interfere with foreground, non-speculative JavaScript that needs CPU cycles. Recent versions of Firefox provide a JavaScript `yield` statement for implementing generators; future versions of Crom may be able to stage pre-commit costs using such generators.

When committing a checked-mode speculation, Crom must patch stale object references. In this experiment, the cost was only 5 ms. As explained in Section 6.2.4, the patching overhead is proportional to the number of objects cloned, which is typically much smaller than the total number of objects in the application.

User-perceived benefits: To keep the GUI responsive, long redraws in Firefox are split into a synchronous part and several asynchronous ones. The commit-time black bar in Figure 9 only captures the synchronous cost. Figure 10 quantifies the end-to-end *user-perceived* latency reduction that is enabled by speculative prefetching and

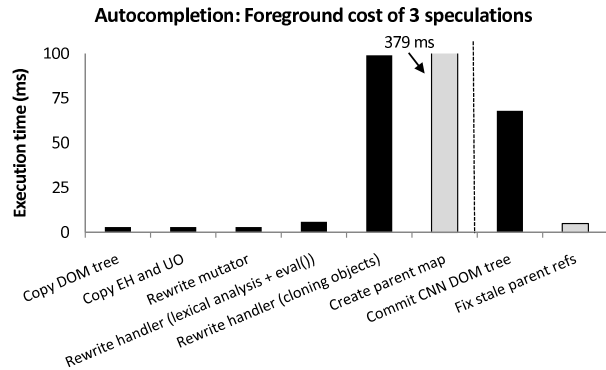


Figure 11: Autocompleter pre- and post-commit costs (checked-mode costs in grey).

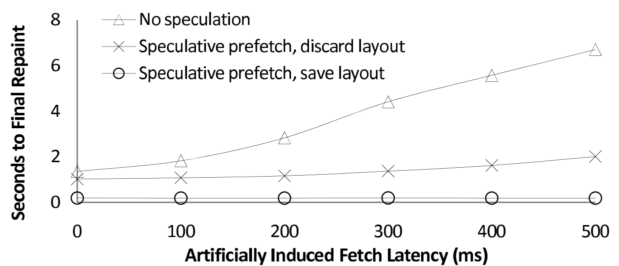


Figure 12: User-perceived latencies for autocompleter.

pre-layout. The injected web server latency varies on the x-axis, and the y-axis provides the delay between the start of the page load and the final screen repaint; repaint activity was detected using the custom Firefox event `MozAfterPaint`.

Figure 10 shows that speculative execution can dramatically reduce user-perceived latencies. For example, with a 300 ms fetch penalty, Crom needed 399 ms to complete the page load, whereas a non-speculative synchronous load with a cold cache required 3,427 ms. Speculatively prefetching the data but discarding the layout (i.e., not committing the speculative context) required 1,302 ms of user-perceived load time. The load penalty relative to the full speculative case arose from two sources: recalculating the layout, and for some objects, waiting for the server to respond to cache validation messages. Such messages are not generated when a precomputed DOM subtree is inserted into the foreground DOM tree.

6.1.3 Autocompletion Widget

To evaluate the speculation overheads for the autocompleter [21], we embedded it inside the `Amazon` front page, simulating a new search box for the page. When the user hit the widget's "submit" button, the widget used AJAX to fetch and display content associated with the user's search terms. The autocompleter speculated after the user had typed two letters, and we limited the number

of speculations to three. To provide a comparison with the tab manager experiment, the widget always fetched the CNN front page. Crom ran in lazy mode with a speculation zone consisting of the <div> tag that would display the fetched content.

Figure 11 depicts the speculation overheads. Examining the overheads from left to right, we see that copying the DOM tree, the event handlers, and the user objects was extremely fast. This is because the autocompleter used a speculation zone that limited the DOM copying to a single div subtree. The mutator function was also very simple, so rewriting it was cheap. However, rewriting the event handler call tree was not cheap due to high cloning costs. The JavaScript objects that implemented the autocompletion logic were fairly complex, and copying the whole set took 33 ms. Since the autocompleter issued three speculations, this object set had to be copied three times.

In unchecked lazy mode, the total pre-commit speculation cost was 114 ms, not counting the 22 ms needed to preemptively create three speculative domains (see Section 5.5.3 for details). Creating a parent mapping would add 379 ms, making the total pre-commit overhead a prohibitive 493 ms. Thus, the autocompleter requires unchecked lazy mode to make Crom's speculations feasible.

Commit costs were similar for the tab manager and the autocompletion widget. As Figure 12 shows, the reduction in user-perceived latency was equally dramatic.

6.2 Exploring Speculation Costs

The previous section showed that Crom's overheads are low when using unchecked lazy cloning and speculation zones. In this section, we examine the costs of full heap and checked lazy speculation, as well as the cost of copying the entire DOM tree. These types of speculation require the least effort from the web developer, since there is no need to identify speculable DOM subtrees or perform checked-mode refactoring. We believe that these activities are fairly straightforward for well-designed code. Unfortunately, poorly written JavaScript code abounds, and even good developers may generate tangled code when working on a constantly evolving web page. Thus, it is important to understand when checked mode or full copy speculation is feasible when the Crom API is not natively implemented by the browser.

6.2.1 Copying the Full Application Heap

Figure 7 describes the JavaScript environment for several popular web sites[2]. As expected, there is wide variation across sites. Figure 13 shows the time needed by

[2]Figure 7 provides a lower bound on the number of variables in each site; Crom cannot discover objects which are only reachable by object trees rooted in DOM 2 handler state.

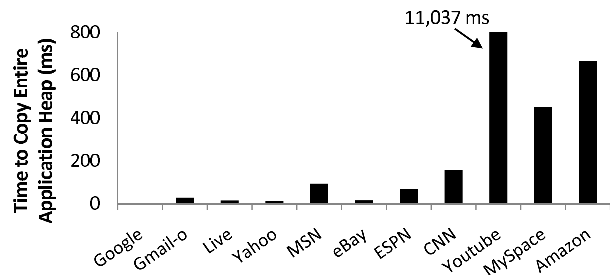


Figure 13: Copying the full application heap.

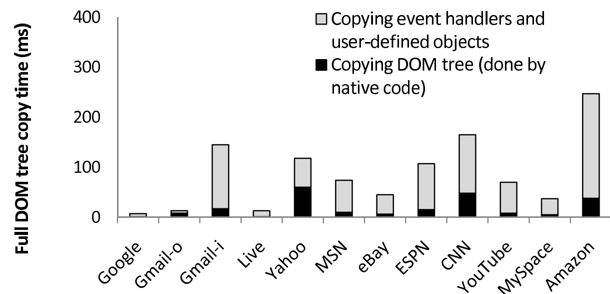


Figure 14: Copying the full DOM tree.

Firefox to copy the full application heap in these sites. For many sites, the one-time copy overhead was low. For example, in a fairly complex site like MSN, the copy cost was 94 ms. However, this is a *per-speculation* cost, since each speculative execution needs a private copy of the heap. The larger the cost, the fewer speculations can be issued in a window of a few hundred milliseconds.

For more complex sites like Amazon, YouTube, and MySpace, full heap copying was prohibitively expensive, mainly due to the overhead of function cloning. This overhead was proportional to both the number and the complexity of the functions. So, even though MySpace and YouTube had a similar number of functions (904 versus 894), it took 452 ms to clone MySpace's heap, but 11 *seconds* to clone YouTube's heap. The high cloning overhead for YouTube is a consequence of its sophisticated JavaScript functionality: it contains over 200K of code for manipulating Flash movies, performing click analytics, and managing advertisements. For sites like this, full heap copying is completely infeasible, making lazy cloning a prerequisite for high-performance speculation.

6.2.2 Copying the Full DOM Tree

To copy the DOM tree (or a branch of the tree), Crom first calls `cloneNode(true)` on the root of the tree. Then, Crom traverses the cloned tree and the base tree in parallel, reference-copying the event handlers and deep-copying the user-defined objects belonging to the non-speculative DOM nodes. Figure 14 shows the relative costs of these two steps. Since `cloneNode()` is imple-

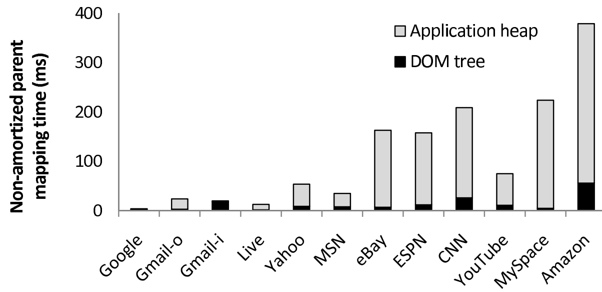


Figure 15: Creating a parent map.

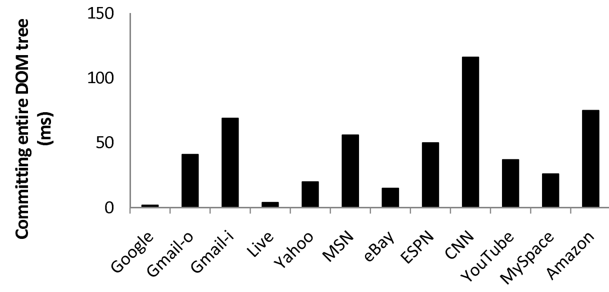


Figure 16: Committing a full DOM tree.

mented in native code, the first step was very fast. The second step was more onerous since it was implemented by non-native Crom code. The aggregate copy cost represents a per-speculation overhead, since each speculative computation must possess a private copy of the tree. Amazon had the highest copy overhead (247 ms), but 40% of the remaining websites had copy penalties above 100 ms. Thus, we expect that most speculative web sites will avoid full DOM copying and use speculation zones.

6.2.3 Creating the Parent Map

Figure 15 shows the cost of creating a parent map in various web sites. The overhead is split into two parts: the cost of traversing the DOM tree, and the cost of walking the object forest rooted in the application heap. Traversing the DOM tree was much faster than traversing the application heap. Whereas Crom could get a list of all DOM nodes using the native code `document.getElementByTagName("*")`, it had to walk the application heap using a user-level breadth-first traversal.

Creating a parent map and updating stale references is unnecessary if Crom copies the entire application heap. However, given the choice between unchecked execution with full heap copying, and checked execution with lazy copying, many applications will prefer the latter. This is because the overhead of parent mapping is amortizable across multiple speculations, whereas full heap copy costs cannot be shared. Comparing Figures 13 and 15 reveals that for sites with small enough heaps to make full heap copying reasonable, creating the parent map is often no more expensive than a single full heap copy; thus, when issuing multiple speculations, lazy copying quickly repays the initial parent mapping overhead.

6.2.4 Committing a Speculative Domain

Committing a speculative context requires three actions. First, the speculative DOM tree must be spliced into the real DOM tree. Second, the cloned object trees from the application heap must be inserted into the real global name space. Third, if Crom is running in checked mode, the non-speculative parents of speculative objects must have their child references patched.

Repairing the heap name space is essentially free—for each reference to an object tree root, Crom just makes it point to the speculative copy of the root object. The time required for parent patching is $O(SP)$, where S is the number of speculatively cloned objects and P is the average number of parents per object. S is typically much smaller than the total number of objects, so in our experience, fixing stale child references takes no more than 20 ms.

Splicing the speculative DOM tree into the real one is handled by native code. However, the splice may be the slowest step of the commit if the speculative tree is large. In these situations, the browser is forced to re-render large regions of the display, a computationally intensive task. Figure 16 shows the splice costs in the absence of speculation zones, i.e., when the entire DOM tree is copied and then reinserted. With the exception of `CNN`, each splice took less than 75 ms. Regardless, DOM splicing is a cost that must be paid by both speculative and non-speculative code, and even checked commits add little overhead to the fundamental cost of updating the browser display.

7 Related Work

File systems often use prefetching to hide disk and network latency [13, 19], and many of these techniques can be applied to the web domain. Empirical studies show that over half of all web objects can be effectively prefetched, doubling the latency reductions available from caching alone [14]. Various academic and commercial projects have tried to exploit this opportunity [4, 15, 20]. For example, Padmanabhan's algorithm uses server-collected access statistics to generate prefetching hints for clients [17]. The current draft of the HTML 5 protocol allows such hints to be specified using a new "prefetch" attribute for `<link>` tags [23]. The Fasterfox extension for Firefox automatically prefetches statically referenced content in a page [10]. All of these approaches are limited to prefetching across static content graphs. In contrast, Crom allows prefetching in web pages with interactive client-side code and dynamic content.

Speculative execution has been used to drive prefetching in file systems [1, 8] and parallel computation systems [2]. In these environments, a process is speculated forward, possibly on incorrect data, to discover what I/O requests may appear in the future. The sole purpose of speculative execution is to warm the cache; other side effects are discarded. In contrast, Crom speculates on user activity rather than the results of I/O operations. When appropriate, Crom can also commit speculative contexts to hide computational latencies associated with screen redraws.

The Speculator Linux kernel [16] supports speculative execution in distributed file systems. A client-side file system process speculates on the results of remote operations and continues to execute until it tries to generate an externally visible output like a network packet. The process is blocked until its speculative input dependencies are definitively resolved. At that point, the process is either marked as non-speculative or rolled back to a checkpoint. Unlike Speculator, Crom has a server-side component which allows speculations to externalize network activity. However, Crom requires applications to be modified, whereas Speculator only modifies the OS kernel.

8 Conclusion

In this paper, we describe why speculative execution is a natural optimization technique for rich web applications. We introduce a high-level API through which applications can express speculative intent, and a JavaScript implementation of this API that runs on unmodified browsers. This implementation, called Crom, automatically converts event handlers to speculative versions and runs them in isolated browser contexts, caching both the fetched data and the screen layout for that data. Experiments show that Crom-enabled applications can reduce user-perceived delays by an order of magnitude, greatly improving the browsing experience. By abstracting away the programmatic details of speculative execution, Crom successfully lowers the barrier to producing rich, low-latency web applications.

References

[1] CHANG, F., AND GIBSON, G. A. Automatic I/O hint generation through speculative execution. In *Proc. 15th ACM Symposium on Operating System Design and Implementation (OSDI)* (1999), pp. 1–14.

[2] CHEN, Y., BYNA, S., SUN, X.-H., THAKUR, R., AND GROPP, W. Hiding I/O latency with pre-execution prefetching for parallel applications. In *Proc. ACM/IEEE Conference on Supercomputing (SC)* (2008), pp. 1–10.

[3] CROVELLA, M., AND BARFORD, P. The network effects of prefetching. In *Proc. IEEE Infocom* (1998), pp. 1232–1239.

[4] DAVISON, B. D. Assertion: Prefetching with GET is not good. In *Proc. 6th International Workshop on Web Caching and Content Distribution (WCW)* (2001), pp. 203–215.

[5] DECIMAIL PROJECT. `http://freshmeat.net/projects/decimail/`.

[6] FIELDING, R., GETTYS, J., MOGUL, J.,FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999.

[7] FLANAGAN, D. *JavaScript: The Definitive Guide, Fifth Edition*. O'Reilly Media, Inc., Sebastopol, CA, 2006.

[8] FRASER, K., AND CHANG, F. Operating system I/O speculation: How two invocations are faster than one. In *Proc. USENIX Annual Technical Conference* (2003), pp. 325–338.

[9] GARRETT, J. Ajax: A New Approach to Web Applications. `http://www.adaptivepath.com/ideas/essays/archives/000385.php`, February 18, 2005.

[10] GENTILCORE, T. Fasterfox Firefox Extension. `http://fasterfox.mozdev.org`, 2005.

[11] JONES, R., AND LINS, R. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[12] KALLELAND, A. DHTML tab widget. `http://www.dhtmlgoodies.com/scripts/tab-view/tab-view.html`.

[13] KORNER, K. Intelligent caching for remote file service. In *Proc. International Conference on Distributed Computing Systems (ICDCS)* (May/Jun. 1990), pp. 220–226.

[14] KROEGER, T. M., LONG, D. D. E., AND MOGUL, J. C. Exploring the bounds of web latency reduction from caching and prefetching. In *Proc. 1st USENIX Symposium on Internet Technologies and Systems (USITS)* (1997), pp. 2–2.

[15] MARKATOS, E. P., AND CHRONAKI, C. E. A top-10 approach to prefetching on the Web. In *Proc. 8th Annual Conference of the Internet Society (INET)* (1998).

[16] NIGHTINGALE, E. B., CHEN, P. M., AND FLINN, J. Speculative execution in a distributed file system. In *Proc. 20th ACM Symposium on Operating Systems Principles (SOSP)* (2005), pp. 191–205.

[17] PADMANABHAN, V. N., AND MOGUL, J. C. Using predictive prefetching to improve World Wide Web latency. *Computer Communication Review 26* (1996), 22–36.

[18] PARR, T. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[19] PATTERSON, R. H., GIBSON, G. A., GINTING, E., STODOLSKY, D., AND ZELENKA, J. Informed prefetching and caching. In *Proc. 15th ACM Symposium on Operating Systems Principles (SOSP)* (1995), pp. 79–95.

[20] PITKOW, J., AND PIROLLI, P. Mining longest repeating subsequences to predict World Wide Web surfing. In *Proc. 2nd USENIX Symposium on Internet Technologies and Systems (USITS)* (1999), pp. 13–13.

[21] SCRIPT.ACULO.US JAVASCRIPT LIBRARY. `http://script.aculo.us/`.

[22] STOCKWELL, C. IE8 Performance. `http://blogs.msdn.com/ie/archive/2008/08/26/ie8-performance.aspx`.

[23] WEB HYPERTEXT APPLICATION TECHNOLOGY WORKING GROUP. HTML 5 Draft Standard. `http://whatwg.org/html5`, fetched July 2, 2009.

[24] WORLD WIDE WEB CONSORTIUM. Document Object Model (DOM). `http://www.w3.org/DOM/`, 2005.

WebProphet: Automating Performance Prediction for Web Services

Zhichun Li[§] Ming Zhang[†] Zhaosheng Zhu[‡] Yan Chen[§] Albert Greenberg[†] Yi-Min Wang[†]
[§] *Northwestern University* [†] *Microsoft Research* [‡] *Data Domain Inc.*

Abstract

Today, large-scale web services run on complex systems, spanning multiple data centers and content distribution networks, with performance depending on diverse factors in end systems, networks, and infrastructure servers. Web service providers have many options for improving service performance, varying greatly in feasibility, cost and benefit, but have few tools to predict the impact of these options.

A key challenge is to precisely capture web object dependencies, as these are essential for predicting performance in an accurate and scalable manner. In this paper, we introduce WebProphet, a system that automates performance prediction for web services. WebProphet employs a novel technique based on timing perturbation to extract web object dependencies, and then uses these dependencies to predict the performance impact of changes to the handling of the objects. We have built, deployed, and evaluated the accuracy and efficiency of WebProphet. Applying WebProphet to the Search and Maps services of Google and Yahoo, we find WebProphet predicts the median and 95th percentiles of the page load time distribution with an error rate smaller than 16% in most cases. Using Yahoo Maps as an example, we find that WebProphet reduces the problem of performance optimization to a small number of web objects whose optimization would reduce the page load time by nearly 40%.

1 Introduction

Software vendors and service providers are increasingly delivering services to users through the Internet. Large-scale web services, such as maps, search, and social networking, have proliferated, attracting hundreds of millions of users worldwide. On the client side, these services heavily leverage Asynchronous Javascript and XML (AJAX) to provide a seamless and consistent user experience across devices and form factors. Behind the scenes, significant amounts of data and computation are provided by servers in the cloud.

Many web services are extremely complex, since they aim to match or even exceed the rich user experience offered by traditional desktop application. For instance, the "driving directions" webpage of Yahoo Maps comprises about 110 embedded objects and 670KB of Javascript code. These objects are retrieved from many different servers, sometimes even from multiple data centers (DCs) and content distribution networks (CDNs). These dispersed objects meet only at a client machine, where they are assembled by a browser to form a complete webpage. Since service providers lack object-level measurements obtained from clients, it is hard for them to assess and study user-perceived performance. Moreover, there exist a plethora of dependencies between different objects. Many objects cannot be downloaded until some other objects are available. For instance, an image download may have to await a Javascript download because the former is requested by the latter. These multiple factors make it highly challenging to understand and predict the performance of web services.

The performance of web services has direct impact on user satisfaction. Poor page load times (PLT) result in low service usage, which in turn may undermine service income. For instance, a study by Amazon reported roughly 1% sales loss as the cost of a 100 ms extra delay. Another study by Google found a 500 ms extra delay in display search results may reduce revenues by up to 20% [16]. Even worse, users may simply abandon a service provider for another offering, as switching barriers are often low.

Ideally, service providers would like to predict the effects of potential optimizations before actual deployment. Yet it is seldom clear what benefits various possible options for improvement might bring a service provider – whether optimization to the object structure of the page, or optimizations in the manner in which content is placed and delivered over the Internet. User-

perceived PLT is affected by the loading time of web objects and their dependencies. The loading time of each individual object is further affected by a variety of *delay factors*, including DNS lookup time, network round trip time (RTT), server response time, and client execution time.

One compelling way to predict performance is to first measure the PLT through experiments on the service itself (*e.g.*, A/B tests [16] by varying a given property of the service), and then to extrapolate estimates using some form of regression. However, such experiments can be difficult to setup and expensive to sustain. It is not uncommon for such experiments to run for days or even weeks, limiting the capacity for adding additional experiments. Furthermore, it is extremely challenging to sweep the space of all possible scenarios since the number of scenarios grows exponentially with the number of objects and delay factors. Without detailed knowledge of object dependencies, it is difficult to decide how many distinct scenarios need to be measured to attain accurate predictions.

Existing approaches for performance prediction generally fall into two broad categories: provider based *vs.* end-system based. In the first category, WISE [23] predicts performance based on server logs collected at the service provider's data centers. As a result, this approach has limited visibility into some client-side factors that are crucial for user-perceived PLT, such as page rendering time, object dependencies, and multiple data sources (crossing data centers and content providers). In the second category, Link Gradients [10] proposes to predict end-to-end response times of untested system configurations, assuming the effects of change in individual factors are completely independent of each other. While this assumption may hold in small-scale enterprise applications, it is inapplicable to complex web services in which inter-component dependencies are prevalent.

To overcome these challenges and shortcomings, this paper presents WebProphet, a tool that predicts the impact of various optimizations on user-perceived PLT of web services. First, WebProphet aims to be applicable to a diverse set of web services. Second, WebProphet aims to automatically produce accurate predictions. Given the number of web services and the churns in their implementations, a tool that involves manual effort can be overly burdensome and error prone.

WebProphet consists of a measurement engine, a dependency extractor, and a performance predictor. The dependency extractor employs a novel algorithm to infer dependencies between web objects by perturbing the download times of individual objects. Our key observation is the delay of an individual object will be propagated to all of its dependent objects. While others have noticed that timing perturbation can convey information (in particular, [20] uses such techniques to transmit data covertly), we are the first to apply it to systematically discovering web object dependencies. Given the dependency graph of a webpage, the performance predictor implements a simple and yet accurate method to simulate the page load process of a web browser. It can make fast and accurate PLT prediction under any combination of changes in objects and delay factors. It can also predict the statistical properties (*e.g.*, median or 95th-percentile) of a PLT distribution under a hypothetical scenario.

We applied WebProphet to four widely-used web services: Maps and Search of Google and Yahoo. We verified that our system successfully extracts the dependency graphs for all these services, even though some of the complex webpages comprise over 100 objects. We used WebProphet to predict PLT on real, popular web browsers using controlled experiments and the PlanetLab testbed. Our evaluation shows that the predictions of WebProphet are highly accurate, with error rates mostly under 16%. This is quite promising given the inherent noise (*e.g.*, different loss conditions) in these experiments. We then apply WebProphet to finding cost-effective optimization strategies for real applications. For instance, Yahoo Maps contains 110 objects and has a median PLT of 3.987 seconds measured from Northwestern University. By simply optimizing the client execution time of 14 objects and moving 5 static objects from Yahoo data centers to the Akamai CDN, the median PLT of Yahoo Maps can be cut by nearly 40%.

We continue to discuss the problem formulation and present an overview of WebProphet in §2. We describe dependency extraction in §3 and performance prediction in §4. The implementation is covered in §5. In §6 and §7, we show the results of dependency extraction and performance prediction respectively. We demonstrate how WebProphet helps to optimize the PLT of Yahoo Maps in §8. We evaluate the systems performance in §9. Finally, we review the related work in §10 and conclude in §11.

2 Problem Context

Many web services are delivered to users in form of webpages that can be rendered by a browser. Sophisticated webpages may contain many static and dynamic objects arranged hierarchically. To load a page, a browser typically first downloads a main HTML object that defines the structure of the page. Next, it may download a Cascading Style Sheets (CSS) object that describes the presentation of the page. The main HTML object may embed many Javascript objects that are executed locally to interact with a user. As the page is being rendered, an HTML or a Javascript object may request additional ob-

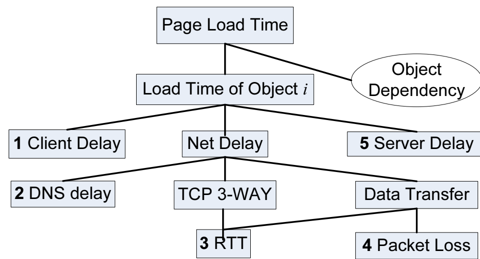


Figure 1: The page load time decomposition.

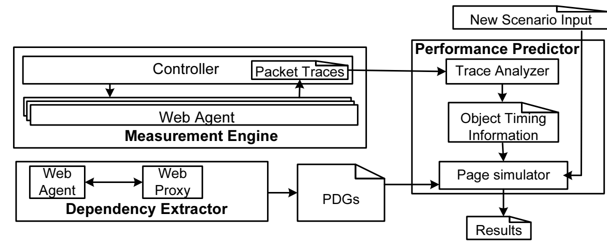


Figure 2: System architecture.

jects, such as images and Javascripts. This process continues until all relevant objects are loaded.

We define page load time (PLT) as the time between when the user triggers the page starting to load and when all the objects in the page are loaded. Sometimes, users do not care about all the objects in a page. For instance, a page may contain invisible images, advertisements, or user tracking services. Moreover, a user action may only trigger a few new objects to be loaded after the initial page load. Accordingly, we could also define PLT as the time to load a subset of objects in a page that are relevant to user-perceived performance. Note that there is a subtle difference between when objects are loaded and when objects are perceived by the user. While the latter is more directly related to user satisfaction, it is also harder to define and measure precisely. Therefore, we choose to focus on the former in this paper.

As illustrated in Figure 1, we may decompose the loading time of each object into client delay, network delay, and server delay. The client delay is due to various browser activities such as page rendering and Javascript execution. The network delay can be further decomposed into DNS lookup time, TCP three-way handshake time, and data transfer time. TCP handshake time and data transfer time are influenced by network path conditions such as RTT and packet loss. The server delay is produced by various server processing tasks such as retrieving static content or generating dynamic content.

Service providers have many different options to improve the PLT of a webpage. For instance, they may upgrade the back-end infrastructure to reduce server response time for dynamic objects. They may use a CDN service to reduce the network delay for static objects. They may also optimize the implementation code to reduce the client execution time for computation-intensive objects. While optimizing for an individual object or delay factor (or for a combination of multiple objects or delay factors) will bring some benefits, they may also incur significant costs in development and management. It is economically infeasible for a service provider to optimize for every object/delay factor, and is often unclear where to find the biggest bang for the buck. Our goal is to build an automated system that can accurately predict the PLT improvement under any combination of changes in object and delay factor. A service provider can easily use our system to narrow down the optimization strategies that could bring the most benefits.

For a web service, WebProphet predicts PLT based on a performance model extracted from client-side observations. Compared to server-side techniques [23], our approach can take into account a few important factors that are visible only at the client. First, a modern webpage usually contains many objects which have dependencies between each other. As a result, the PLT cannot be estimated simply based on the page size and TCP-level characteristics such as RTT, packet loss, and congestion window size. In fact, the dependencies will determine when an object can be loaded and which objects can be loaded in parallel. Second, many webpages comprise sophisticated HTML and Javascript objects to provide a rich user experience. Nonetheless, HTML rendering and Javascript execution may introduce significant client delay. Third, the objects in a page may come from multiple data centers and CDN nodes. For example, Yahoo Maps uses both the Akamai CDN and Yahoo data centers to deliver page content. Though the client side is the ideal place where we can measure the user-perceived PLT accounting for all these effects end-to-end, existing client browsers lack the measurement hooks needed.

As shown in Figure 2, WebProphet has three major components. Given a webpage, the *dependency extractor* infers the dependencies between objects by perturbing the download times of individual objects. The *measurement engine* controls multiple automated *web agents* which can drive a full-featured web browser (Firefox 3) to load the page. The measurement engine also collects one packet trace for each page load. Using the extracted dependency graph and the packet trace in a baseline scenario, the *performance predictor* estimates the PLT in a new scenario by simulating the page load process.

The PLT of a webpage will not be a constant due to the variations of network latency, server response time, and load on the client. WebProphet can predict the statistical properties (*e.g.*, median or 95th-percentile) of the PLT

distribution under a new scenario. For this purpose, we first collect a reasonably large number of page load traces in a baseline scenario using a web agent. Then, for each of these traces, we run performance prediction to obtain the PLT in the new scenario, and therefore produce the PLT distribution in a new scenario.

Currently, we do not explicitly consider the effect of packet loss in our model. In other words, we assume the same loss condition in the baseline and new scenarios. Differences in loss conditions can change the number of round trips involved in loading an object, which in turn lead to prediction errors (§4.1). The impact of packet loss on PLT can be highly variable, and highly dependent on factors such as network transients, TCP congestion states, and specific TCP loss recovery mechanisms. In spite of this limitation, as shown in §7, WebProphet attains high prediction accuracy in both controlled and real-world experiments under normal loss conditions.

3 Dependency Extraction

In this section, we first present an overview of dependency relationships between web objects and describe the types of dependencies that we aim to discover. We then explain the details of our dependency extraction algorithm based on timing perturbation.

3.1 What are dependencies?

Modern webpages may contain many types of objects, including HTML, Javascript, CSS, and image. These embedded objects are downloaded via separate requests on potentially multiple TCP connections instead of all at once. For instance, the main HTML object may contain a Javascript object whose execution will lead to additional downloads of HTML and image objects. We say one object *depends* on the other if the former cannot be downloaded until the latter is available. Dependencies between objects can be caused by a number of reasons. Common ones include: i) The embedded objects in an HTML page will depend on the HTML page; ii) Since many objects are dynamically requested during Javascript execution, these objects depend on the corresponding Javascript; iii) The download of an external CSS or Javascript object may block the download of other types of objects in the same HTML page [22]; iv) Object downloads may depend on certain events in Javascript object or web browser. For instance, a Javascript object may download image B only after image A is loaded.

Given an object A, its dependent objects usually cannot be requested before A is *completely* downloaded. However, there are exceptions. Today's browsers render an HTML page in a streamlined fashion, by which we mean the HTML page can be partially displayed even before its download finishes. For example, if an HTML page has an embedded image, the image can be downloaded and displayed in parallel with the download of the HTML page. The image download may start once the tag `<img src=... />` (identified by a byte offset in the HTML page) has been parsed. We call an HTML object a *stream* object. We use *dependency* $offset_A(img)$ to denote the offset of the last byte of `<img src=... />` in the stream object A. We observed this streamlined processing behavior in major browsers including IE, Firefox and Chrome.

Given an object X, we use $descendant(X)$ to denote the set of objects that depend on X and use $ancestor(X)$ to denote the set of objects that X depends on. By definition, X cannot be requested until all the objects in $ancestor(X)$ are available. Among the objects in $ancestor(X)$, we are particularly interested in object Y which is the last to become available. We call Y the *last parent* of X. If Y is a stream object, its *available-time* is when the dependency $offset_Y(X)$ has been loaded. If Y is a non-stream object, its available-time is when Y is completely loaded. In §4.1, we will explain how to use the available-time of Y to estimate the start time of X's download. Essentially, this will allow us to predict the PLT of a webpage. While X only has one last parent in one particular page load, its last parent may change across different page loads due to variations in the available-time of its ancestors. We use $parent(X)$ to denote the subset of the objects in $ancestor(X)$ which may be the last parent of X.

Given a webpage, we use a *parental dependency graph (PDG)* to encapsulate the parental relationship between objects in the page. A $PDG = (V, E)$ is a Directed Acyclic Graph (DAG) and includes a set of nodes and directed links. Each node is a web object. Each link $Y \leftarrow X$ means Y is a parent of X.

3.2 How to extract dependencies?

WebProphet extracts the dependencies of a webpage by perturbing the download of individual objects. Our key observation is the delay of an individual object will be propagated to its descendants. While conceptually simple, the major challenge is to extract the stream parent of an object and the corresponding dependency offset. Suppose an object X has a stream parent Y. To discover this parental relationship and the dependency offset, the available-time of $offset_Y(X)$ must be later than that of all the other parents of X in a particular page load. This requires the ability to control the download of not only each non-stream parent of X as a whole but also each partial download of each stream parent of X. As we will

see in §7.4, correctly extracting stream parents and dependency offsets is critical for accurate PLT prediction.

Discovering ancestors/descendants: Given a webpage and its embedded objects, we discover the descendants of each object iteratively. In each round, we reload the page and delay the download of an object X for τ seconds. Here, X is an object which has not been processed and τ is much greater than the normal loading time of any object. The descendants of X are the objects whose download is delayed together with X for at least τ seconds. We repeat this process until the descendants of all the objects are discovered. Note that the order by which we delay each object has no influence on the final result.

Our approach for dependency extraction makes two assumptions. First, we assume the dependencies of a webpage do not change during the discovery process. This may not hold in practice. When a page is reloaded, there could be some minor changes in the new page. For instance, there could be parameter changes in the Universal Resource Identifier (URI) of certain objects. We tackle this problem by matching similar URIs in different rounds according to *edit distance*. Moreover, there could be object changes due to reasons such as new advertisements. We find such changes tend to have limited impact on the overall structure of the page or the PDG. This is because the number of affected objects is small and they usually do not have any descendants. In § 7, we will show that our prediction results are highly accurate in spite of minor changes in webpages.

The second assumption we make is that the artificially injected delay will not change the dependencies in the page. Among the pages we studied, we found only one exception in the "driving directions" webpage of Google Maps. There are two Javascripts `A.js` and `B.js` which have the same parent `main.js`. We use `A` and `B` to represent the names of these two Javascripts, given their original names are very long. When `main.js` is severely delayed, `A.js` and `B.js` sometimes are combined into one single Javascript named `AB.js`. This probably reflects the fact that `main.js` attempts to adapt when it detects poor download speed. We identified this application behavior because it leads to inconsistencies in the extracted dependencies of the page. Among the applications we studied, only Google Maps exhibits this behavior which is handled with a simple heuristic. In the future, we plan to devise a more systematic solution to deal with such behavior.

Extracting non-stream parents: Given a non-stream object X and its descendant Z, we observe that X is the parent of Z if and only if there does not exist an object Y which is the descendant of X and the ancestor of Z. On the one hand, if such Y exists, the available-time of Y will always be later than that of X. This is because X is a non-stream object and Y cannot be downloaded until X is available, which implies X cannot be the parent of Z. On the other hand, if Y does not exist, we can imagine a scenario where X is delayed until all the other ancestors of Z are available. This is possible because none of the other ancestors of Z depend on X. This implies X may indeed be the parent of Z. Based on this observation, Algorithm `ExtractNonStreamParent` takes the set of objects and the set of descendants of each object (inferred from the previous step) as input and computes the parent set of each object.

```
ExtractNonStreamParent(Object, Descendant)
For X in Object
    For Z in Descendant(X)
        IsParent = True
        For Y in Descendant(X)
            If (Z in Descendant(Y))
                IsParent = False
                Break
            EndIf
        EndFor
        If (IsParent) add X to Parent(Z)
    EndFor
EndFor
```

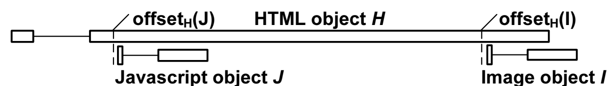


Figure 3: Stream parent example.

Extracting stream parents and dependency offsets: The method described above may not be useful for discovering the stream parent of an object. We illustrate this with an example in Figure 3. A large HTML object H contains a Javascript J and an image I. J and I are embedded in the beginning and the end of H respectively ($offset_H(J) < offset_H(I)$). Because the URI of I is defined in J, I cannot be downloaded until J is executed. This causes I to depend on both H and J while J only depends on H. According to the previous method, H cannot be the parent of I since J is the descendant of H and the ancestor of I. Nonetheless, when the download of H is slow, J may have been downloaded and executed before $offset_H(I)$ becomes available. In this case, H becomes the last parent of I.

Given a stream object H and its descendant I, we use the following method to determine whether H is the parent of I. We first reload the whole page and control the download of H at an extremely low rate λ. If H is the parent of I, all the other ancestors of I should have been available by the time $offset_H(I)$ is available. We can then estimate $offset_H(I)$ with $offset_H(I)'$, where the latter is the offset of H that has been downloaded when the request of I starts to be sent out. $offset_H(I)'$ can be directly inferred from network traces and is usually a bit larger than $offset_H(I)$. This is because it may take

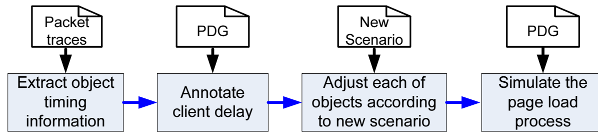


Figure 4: Performance prediction procedure.

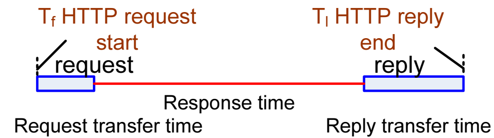


Figure 5: Decomposition of an HTTP activity.

some extra time to request I after $offset_H(I)$ is available. Since H is downloaded at an extremely low rate, these two offsets should be very close.

Given $offset_H(I)'$, we perform an additional parental test to determine whether H is the parent of I. We reload the whole page again. This time, we control the download of H at the same low rate λ as well as delay the download of all the known non-stream parents of I by τ. Let $offset_H(I)''$ be the offset of H that has been downloaded when the request of I is sent out in this run. If $offset_H(I)'' - offset_H(I)' \ll \tau \times \lambda$, this indicates the delay of I's known parents has little effect on when I is requested. Therefore, H should be the last parent of I.

The choice of λ reflects the trade-off between measurement accuracy and efficiency. A smaller λ allows us to estimate $offset_H(I)$ more accurately but leads to longer running times. The parameter τ directly affects the accuracy of parental tests. If τ is too small, the results may be susceptible to noise in experiment, increasing the chance of missing true parents. If τ is too large, we may mistakenly infer a parental relationship because $offset_H(I)'' - offset_H(I)'$ is bounded by $size_H - offset_H(I)$ where $size_H$ is the page size of H. In our current system, we use $\lambda = size_H/200$ bytes/sec and $\tau = 2$ seconds. This means the HTML object H will take 200 seconds to transfer. We will study the accuracy of dependency extraction in § 6.

Discussion: We currently infer the timing information from the packet trace of a page load (§4.1). One alternate approach is to extract dependencies through some combination of static and dynamic program analysis. In fact, it is quite straightforward to parse an HTML object to extract its dependencies. However, extracting the dependencies of Javascript objects requires extensive browser instrumentations. Since the PDG of a page may vary depending on how the page is rendered by a browser, we will have to instrument each type and each version of the major browsers. In comparison, our trace-based approach can more easily work with different browsers.

4 Performance Prediction

In this section, we describe our methodology for predicting performance under hypothetical scenarios. Given the PLT of a webpage in a baseline scenario, we aim to predict the new PLT when there are changes in the *delay factors* (including client delay, server delay, RTT, and DNS lookup time) of any objects in the page. The basic idea is to develop a model that can simulate the page load process of a browser under any hypothetical scenarios. In practice, the page load process can be very complex, since it also relates to browser behavior and parameters, web objects dependencies, versions of TCP and HTTP protocols, and network conditions. The key challenge is to keep the model simple and yet accurate. This requires us to provide the right level of abstraction in the model which captures the most fundamental characteristics of webpages and browsers.

Figure 4 illustrates the overall flow of performance prediction in WebProphet. We first infer the timing information of each object from the packet trace of a page load in a baseline scenario. Based on the PDG of the page, we further annotate each object with additional timing information related to client delay. We then adjust the object timing information to reflect the changes from the baseline scenario to the new one. Finally, we simulate the page load process with the new object timing information to estimate the new PLT. We will explain the first three steps in §4.1 and leave the details of the last step in §4.2.

4.1 Acquiring object timing information

Inferring basic object timing information: We infer web objects and their timing information from the packet trace of a page load collected on the client side. This makes our approach easily deployable since it does not require any instrumentation in browsers or applications. We identify three types of *activities* in the trace:

- DNS: the time used for looking up a domain name.
- TCP connection: the time used for establishing a TCP connection.
- HTTP: the time of loading a web object. As illustrated in Figure 5, an HTTP activity can be further decomposed into three parts: (i) Request transfer time: the time to transfer the first byte to the last byte of an HTTP request; (ii) Response time: the time from when the last byte of the HTTP request is sent out to when the first byte of the HTTP reply is received. This includes one RTT plus server delay; (iii) Reply transfer time: the

time to transfer the first byte to the last byte of an HTTP reply.

In addition, we infer the RTT for each TCP connection. The RTT of a TCP connection should be quite stable since the entire page load process usually lasts for only a few seconds. We also infer the number of round-trips involved in transferring an HTTP request or reply. Such information allows us to predict HTTP transfer times when RTT changes. We will provide the details of packet trace analysis in §5.3.

Adding client delay information: When the last parent of an object X becomes available, the browser will not issue a request for X immediately. This is because the browser needs time to do some additional processing, *e.g.*, parsing HTML page or executing Javascript. For object X, we use the *client delay* to denote the time from when its last parent is available to when the browser starts to request it. When the browser loads a sophisticated webpage or the client machine is slow, client delay may have significant impact on PLT. We infer the client delay of each object by combining basic object timing information with the PDG of the page. Note that when the browser starts to request an object, the first activity can be DNS, TCP connection, or HTTP depending on the current state and behavior of the browser.

Many browsers limit the maximum number of TCP connections to a host, *e.g.*, six in IE 8 and Firefox 3. This can cause the request for an object to wait for available connections even when it is ready to be sent. Therefore, the client delay we observe in a trace may be longer than the actual browser processing time. To overcome this problem, when collecting the packet trace in a baseline scenario, we set the TCP connection limit of the browser to a large number, for instance, 30. This helps to eliminate the effects of connection waiting time. Nonetheless, we will still predict the PLT in a new scenario under the default TCP connection limit of the browser (§4.2).

Adjusting object timing information according to new scenario: So far, we have obtained the object timing information under the baseline scenario. We need to adjust the timing information for each object according to the new scenario. Let $server_\delta$ be the server delay difference between the new and the baseline scenario. We simply add $server_\delta$ to the response time of each object to reflect the server delay change in the new scenario. We use similar methods to adjust DNS activity and client delay for each object. RTT change (rtt_δ) needs some special handling. Suppose the HTTP request and response transfers involve m and n round-trips for object X. We will add $(m + n + 1) \times rtt_\delta$ to the HTTP activity of X and rtt_δ to the TCP connection activity if a new TCP connection is required for loading X. Our assumption is that the number of round-trips involved in loading an object is the same in the baseline and new scenarios. Our results in §7 confirm the validity of this assumption in PlanetLab experiments. This assumption could be violated if bandwidth becomes the bottleneck, *e.g.*, in DSL link. Further research is needed to deal with such scenarios.

Discussion on object & DNS cache: Besides the four delay factors mentioned above, the PLT of a page in a new scenario will also be affected by the object and DNS cache. To handle cached objects and DNS names in a new scenario, we collect page load traces with the same set of cached objects and DNS names in a baseline scenario. We will explain how to control object and DNS cache in §5.1. Suppose Ψ is the PDG of a page when no object is cached. When an object x is cached, Ψ will transform into a new PDG Ψ' where x is removed and each of its children x_c is directly connected with each of its parents x_p. Accordingly, the client delay of (x_c, x_p) in Ψ' will include the cache lookup time of x and the client delay of (x, x_p) and (x_c, x) in Ψ. Hence, there is no need to explicitly consider the timing information of x in Ψ'.

Our current approach can only predict the PLT under the same caching state. Given a page with n objects, we will need to measure 2^n baseline scenarios to handle all the possible caching states. To reduce the measurement overheads, we could explicitly model the timing information of a cached object x in three cases: i) TTL has not expired: x is directly looked up from cache; ii) TTL has expired but x has not changed: x is revalidated and then looked up from cache; iii) TTL has expired and x has changed: x is revalidated and downloaded from the server. To predict the PLT under any caching state, we simply need to extend our model to include the cache lookup time and the number of round trips involved in the revalidation of each object. We can use a small constant to represent the former and perform controlled experiments to measure the latter. The details are out of the scope of this paper.

4.2 Simulating page load process

We now describe our methodology for predicting PLT based on object timing information. The key challenge here is object downloads are not independent from each other. The download of an object may be blocked because its dependent objects are unavailable or because there are no TCP connections ready for use. To tackle these problems, we simulate the page load process by taking into account the constraints of browser and PDG.

Browser behavior: We studied a few popular browsers including IE, Firefox and Chrome. They share a few important features. Presently, they all use HTTP/1.1 either with HTTP pipelining disabled by default or without pipelining support at all. This is because HTTP pipelin-

Case	I	II	III	IV	V
First web object of a domain	Y	Y	N	N	N
Cached DNS name	N	Y	-	-	-
Available TCP connections	-	-	Y	N	N
Max # of parallel connections	-	-	-	N	Y
Involved activities in Figure 6	a	b	c	b	d

Table 1: Five possible cases for loading an object.

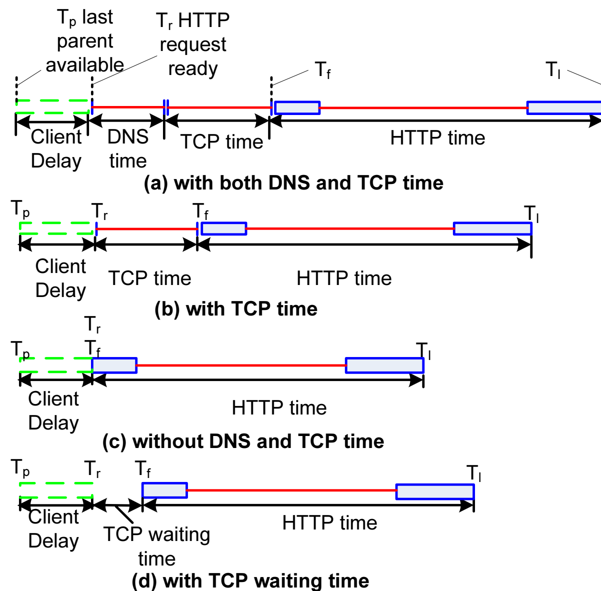


Figure 6: Four cases of activities for loading an object.

ing has not been widely supported by proxies and may have head-of-line blocking with the presence of dynamic contents [1], *e.g.*, one slow response in the pipeline will block other subsequent responses. Given the fact, we do not consider the effect of pipelining in this paper. More sophisticated techniques might be needed to model pipelining if it becomes widely-used in the future. Without pipelining, HTTP request-reply pairs do not overlap with each other within the same TCP connection.

In HTTP/1.1, a browser uses persistent TCP connections which can be reused for multiple HTTP requests and replies. The browser attempts to keep the number of parallel connections small. It opens a new connection only when it needs to send a request but all the existing connections are occupied by other requests or replies. A browser is configured with an internal parameter to limit the maximum number of parallel connections with a particular host. Note that the limit is applied to a host instead of to an IP address. If multiple hosts map to the same IP address, the number of parallel connections with that IP address can exceed the limit.

Loading an object may trigger multiple activities including looking up a DNS name, establishing a new TCP connection, waiting for an existing TCP connection, and/or issuing an HTTP request. Table 1 lists the five possible cases and the conditions of each of these cases. A "-" in the table means the corresponding condition does not matter. The activities involved in each case are illustrated in Figure 6. For instance, in Case V, a browser needs to load an object from a domain with which it already has TCP connections. Because all the existing TCP connections are occupied and the number of parallel connections has reached the maximum limit, the browser has to wait for the availability of an existing connection to issue the new HTTP request (Figure 6(d)).

```
PredictPLT(ObjectTimingInfo, PDG)
Insert root objects into CandidateQ
While (CandidateQ not empty)
  1) Get earliest candidate C from CandidateQ
  2) Load C according to conditions in Table 1
  3) Find new candidates whose parents
     are available
  4) Adjust timings of new candidates
  5) Insert new candidates into CandidateQ
Endwhile
```

Simulating page load: Given a webpage, Algorithm `PredictPLT` takes the timing information of each object and the PDG as input and simulates the page load process. The PLT is estimated as the time when all the objects are loaded. For each object X, the algorithm keeps track of four time variables: i) T_p: when X's last parent is available; ii) T_r: when the HTTP request for X is ready to be sent; iii) T_f: when the first byte of the HTTP request is sent; and iv) T_l: when the last byte of the HTTP reply is received. Figure 6 illustrates the position of these time variables in four different scenarios. In addition, the algorithm maintains a priority queue $CandidateQ$ which contains the objects that can be requested. The objects in $CandidateQ$ are sorted based on T_r.

5 Implementation

As illustrated in Figure 2, the implementation of WebProphet comprises three major components. In this section, we will describe each of them in more detail. The whole system is implemented with roughly 11,000 lines of code in Python, Perl, Javascript and Bro's policy language [18].

5.1 Measurement engine

The measurement engine includes a set of *web agents* which are currently deployed at multiple PlanetLab sites. These web agents allow us to measure the PLT of a webpage under diverse client, server, and network conditions. A centralized *controller* is used to maintain the

continual operation of the agents and perform upgrade when a new version of agent software becomes available. The controller can upload script snippets to an agent to control the interaction between the agent and a webpage. It also retrieves and stores the packet traces from the agents logged by `tcpdump`. The controller is written in Perl and Python with 1,300 lines of code.

The web agent needs to meet a few requirements. First, it should be able to interact with a webpage automatically. As mentioned in §2, WebProphet requires a potentially large number of page load traces in a baseline scenario to predict the statistical properties (*e.g.*, median or 95th-percentile) of the PLT distribution in a new scenario. Second, it should behave like a full-featured web browser in order to simulate user interaction with sophisticated web 2.0 applications, *e.g.*, Google Maps. This is especially important for correctly measuring the user-perceived PLT of these complex applications. Third, it should provide support for setting object and DNS cache, which will affect the PLT (§4.1). We need to control the web agent to cache the same set of objects and DNS names in the baseline and new scenarios. Fourth, it should be able to adjust the parallel TCP connection limit, *e.g.*, to a large number, to eliminate the impact of connection waiting time (§4.1).

Existing web measurement tools cannot meet our requirements. Simple web clients (*e.g.*, wget, curl, and lynx) do not execute the Javascripts in the pages. Web form automation tools [3, 5, 7] and KITE [4] (an automated web measurement tool developed by Keynote) do not provide control for object/DNS cache or TCP connection limit. This prompts us to develop our own web agent, which uses Jssh plug-in to take full control of the Firefox 3 browser. Through the XPCOM [8] interfaces of Firefox, we use Javascript to call the internal APIs of Firefox. These internal APIs supports object and DNS cache control, TCP connection limit adjustment, and user input simulation. The user inputs from mouse and keyboard can be simulated as DOM [6] events.

We developed a set of script snippets to automate the interactions with multiple complex web services, such as Google/Yahoo Search, Google/Yahoo Maps, Gmail, Hotmail, Flickr, *etc.* The script snippet for each web service comprises only about 10 to 150 lines of code, depending on the complexity of the service. We believe it is quite easy to create new script snippets for other services too. The web agent, excluding the service-specific script snippets, is implemented in Javascript and Python with 1,100 lines of code. The whole automation part of the web agent has no measurable effects on PLT since it incurs very little overhead.

5.2 Dependency extractor

To extract the PDG of a web page, we setup a web agent to go through a web proxy running on the same host. The web proxy is modified from a simple proxy written in Python. We extended the proxy with the support of delaying the download of any specified object, which is required for discovering the descendants of the object (§3.2). We also added the functionality of controlling the download speed of a stream object, which is required for discovering stream parent and dependency offset (§3.2).

Given a webpage, we first obtain the list of its embedded objects by loading it once. The proxy will cache all the objects observed in the first round for future use. This reduces the measurement overhead imposed on the origin servers. We then subsequently control the download of one object during each page reload and record the timing information of object download. Finally, we extract the PDG according to the procedure described in §3.2. The dependency extractor and web proxy include 2,800 lines of code in Python.

5.3 Performance predictor

The performance predictor comprises a *trace analyzer* and a *page simulator*. The trace analyzer extracts basic object timing information (described in §4.1) from packet traces in pcap format. It leverages Bro [18], a network intrusion detection system, to parse DNS, TCP, and HTTP protocol information. We write programs using Bro's policy language to recover timing information, *e.g.*, DNS lookup time, TCP handshake time, and HTTP transfer and response times.

We estimate the RTT of a TCP connection using the time between the SYN and SYN/ACK packets. This is because many web services have relatively short TCP connections (*e.g.*, a few seconds) and the RTT is usually quite stable in such time scale. We could use other existing techniques [12, 24] to estimate RTT for web services that involve long TCP connections. We infer the number of round trips involved in an HTTP transfer based on the TCP self-clocking behavior [24] — the packets in the same TCP send window are very close to each other while the packets in two adjacent send windows are at least one RTT apart. We compute the server delay by subtracting one RTT from the time interval between two adjacent send windows. Using this method, we find Google and Yahoo Search process user query in a streamlined fashion. The server will return partial results to users while additional results are being generated. This causes some extra delay to the reply packets in multiple different round trips. Our method can handle such cases well, leading to high prediction accuracy reported in §7.

The page simulator combines the basic object timing information and the PDG to infer the client delay of each

object. It then adjusts the timing of each object according to the specifications in the new scenario. Finally, it simulates the page load process (§4.2) and outputs the predicted PLT. The trace analyzer and page simulator include 6,200 lines of code written in Python and Bro's policy language.

6 Results of Dependency Extraction

We now characterize and validate the PDGs of Google/Yahoo Search and Maps extracted by WebProphet. Google/Yahoo Search are two of the most popular web services and their PDGs are relatively easy to validate. In contrast, the PDGs of Google/Yahoo Maps are much more complex. In fact, Yahoo Maps has the most complicated PDG in terms of the number of objects and dependencies among all the web services we studied (including Amazon.com, Flickr, and Google Docs).

In this paper, we only present the results on the cases where there is no cached object. This is common for accessing webpages in which most contents are dynamic. For instance, Yahoo performance team found 40 to 60% of Yahoo users have an empty cache when visiting Yahoo [22]. Our approach also works when some objects are cached and we omit those results here due to lack of space.

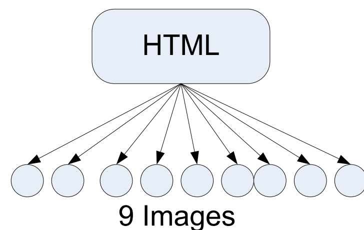


Figure 7: The PDG of Google Search.

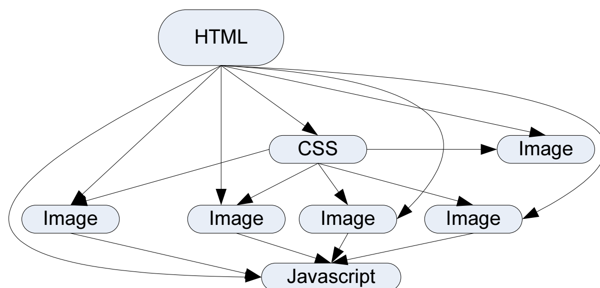


Figure 8: The PDG of Yahoo Search.

Google/Yahoo Search PDG: Figure 8 illustrates the inferred PDGs of Google/Yahoo Search for the search keyword "mapouka". The Google Search page has one HTML and several images while the Yahoo Search page has one HTML, one CSS, and a few Javascripts and images. The Google Search page is simpler than that of Yahoo. The former has not only a fewer number but also fewer levels of dependencies. This could be one of the factors that cause the PLT of Google Search to be smaller (§7). The PDGs for different keywords have similar structure. Some keywords may lead to an extra Javascript object or a different number of images in the PDG of Google Search. Because the Search pages are not very complicated, we are able to verify that the PDGs produced by WebProphet are the same as those extracted through manual code analysis.

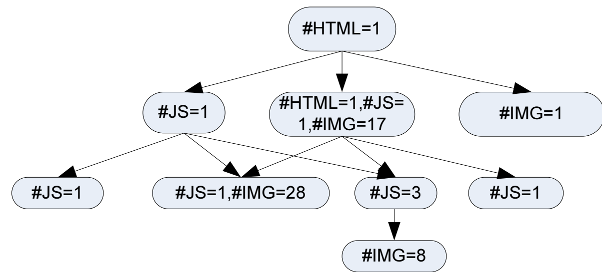


Figure 9: The simplified PDG of Google Maps.

Google/Yahoo Maps PDG: We study the PDG of the driving direction pages of Google/Yahoo Maps. We use a pair of addresses of the Whole Foods stores in Arkansas, USA. The full PDGs of Google/Yahoo Maps are too complex to read, *e.g.*, Yahoo Maps has a total number of 110 objects and 172 dependencies. Instead of showing the full PDGs, we simplify them by merging the objects that share the same sets of parents and children into a single node. The two simplified PDGs are shown in Figure 9 and 10. Each node carries a label which describes the number of objects of certain type. For instance, label "#HTML=1,#JS=1,#IMG=17" means this node corresponds to 1 HTML, 1 Javascript, and 17 image objects in the full PDG.

Apparently, the PDGs of Maps are significantly more sophisticated than those of Search. They include more Javascripts for user interactions and more images for map tiles. The PDG of Yahoo Maps is even more complex than that of Google Maps, as the former comprises a larger number and more levels of dependencies. The PDGs for different address inputs are quite similar. The main differences are in the map tile images.

The Javascripts of Google/Yahoo Maps are not only large (536KB and 670KB respectively) but also obfuscated. It is difficult for us to validate the extracted PDGs via manual code analysis. Instead, we verify their correctness in an indirect manner. First, we use our dependency extractor to obtain the "approximate" PDGs of Google/Yahoo Maps. Then we construct our own web-

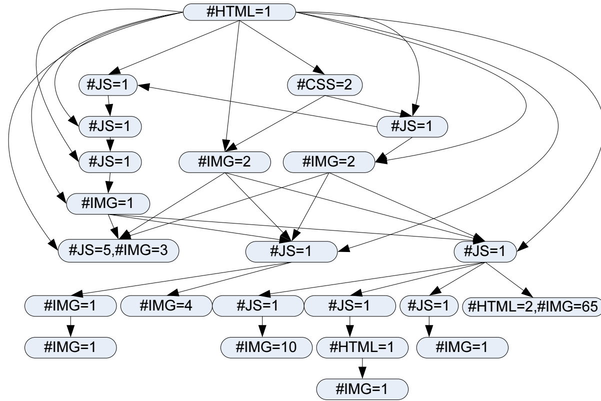


Figure 10: The simplified PDG of Yahoo Maps.

pages which exactly match the "approximate" PDGs in the number of objects, the types of objects, and the dependencies between objects. After that, we attempt to infer the PDGs of the constructed pages as if we know nothing about the "approximate" PDGs. We find the inferred PDGs exactly match the "approximate" PDGs. Although this does not prove that we have extracted the real PDGs of Google/Yahoo Maps, it at least suggests that we can correctly handle webpages as complex as Google/Yahoo Maps. Moreover, in §7, we will show that WebProphet can accurately predict the PLT of Google/Yahoo Maps under various hypothetical scenarios using the "approximate" PDGs.

7 Prediction Accuracy

In this section, we evaluate the PLT prediction accuracy of WebProphet for Google/Yahoo Search and Maps. We first conduct controlled experiments by manipulating the DNS delay, RTT, and server delay for all the objects or a subset of the objects in a webpage. We then conduct real-world experiments in which every delay factor of every object changes simultaneously. We find that ignoring object dependencies may lead to significant errors in the PLT prediction for complex webpages. Finally, we show that identifying stream parent and dependency offset is particularly important for accurate PLT prediction for simple webpages.

Suppose t_b and t_n are the PLT in the baseline and new scenarios and t_p is the PLT predicted by WebProphet. We could have used $err_r = \frac{abs(t_p - t_n)}{t_n}$, the relative error between t_p and t_n, to evaluate the prediction accuracy. However, we find err_r may not be the right metric because it tends to be small when $abs(t_b - t_n) << t_n$. Therefore, we choose $err_c = abs(1 - \frac{t_p - t_b}{t_n - t_b})$ as our evaluation metric. It represents the relative error of predicted PLT change compared to the actual PLT change between the baseline and new scenarios. For instance, suppose $t_b = 5$, $t_n = 4$, and $t_p = 4.2$ seconds. The prediction error will be 5% measured in err_r *vs.* 20% measured in err_c. As mentioned in §2, the PLT of a webpage may not be a constant under a given scenario. In this paper, we focus on err_c^{50} and err_c^{95} which are computed based on the median and 95th-percentile in the baseline, new, and predicted PLT distributions. These two metrics help to quantify whether WebProphet can make accurate prediction both for the normal case and for the extreme case.

In the following experiments, we consider the scenarios where a web service provider is interested in predicting the PLT reductions as a result of certain optimizations to the service. To evaluate the prediction accuracy in each of the experiments, we collect two set of page load traces in the baseline scenario. The first set, collected with normal TCP connection limit, is for producing the baseline PLT distribution. The second set, collected with a large TCP connection limit, is for inferring the object timing information in the baseline scenario (§4.1). Thereafter, this object timing information is used to generate the predicted PLT distribution in the new scenario. We also collect one set of traces in the new scenario, from which we can extract the actual new PLT distribution for validation purpose. Each of the three sets contains 500 page load traces, which provides enough samples for computing err_c^{95} and err_c^{50}. We only present the results based on one random keyword for the Search services and one random pair of addresses for the Maps services. The results of using other keywords or pairs of addresses are similar.

Before presenting the results, we discuss two problems that may cause the predicted PLT to deviate from the actual PLT. First, there could be slight differences between the times when the three set of traces are collected. These time differences may lead to differences in the client, server, and network conditions under which the traces are collected. The resulting "prediction error" is actually due to the limitations of our validation methodology rather than due to the limitations of our approach. Second, as mentioned in §2, we currently do not explicitly consider packet loss in our model. Any loss behavior differences between the baseline and new scenarios may also lead to prediction error. As shown in the following results, WebProphet can attain high prediction accuracy in spite of these two problems.

7.1 Controlled experiments

In the controlled experiments, we evaluate the accuracy of our performance prediction under various RTTs, DNS lookup times, and server response times. Figure 11 depicts the setup of our experiments located at Northwestern University. We run a web agent to collect the packet

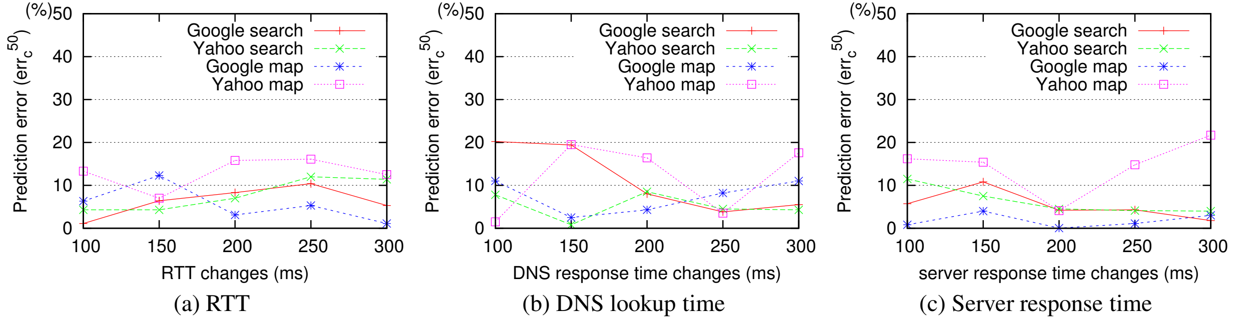


Figure 12: The prediction errors under different injected delays of RTT, DNS lookup time, and server response time.

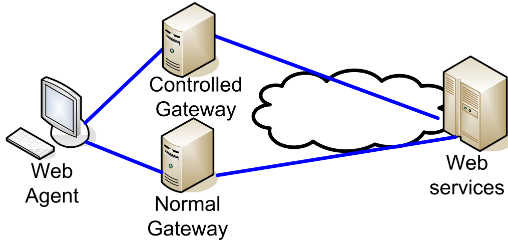


Figure 11: Setup of controlled experiments.

Service	t_b^{50}	t_n^{50}	t_p^{50}	err_c^{50}	err_c^{95}	Indep err_c^{50}
Gsearch	0.74	0.21	0.23	3.8%	15.9%	1.2%
Ysearch	1.04	0.26	0.24	3.2%	2.2%	13.0%
Gmap	4.10	2.12	2.01	5.5%	11.0%	49.7%
Ymap	6.19	3.99	4.03	1.7%	1.2%	85.5%

Table 2: Changing RTT and DNS lookup time together.

traces of page loads. The controlled gateway is used to inject extra delays during page loads. The normal gateway does not manipulate any traffic. We configure the routing table on the web agent to forward the traffic to the controlled gateway or to the normal gateway to create the baseline and new scenarios respectively. Since we currently do not have a precise way to inject client delays, we simply keep the web agent lightly-loaded all the time. This ensures client delays are roughly the same in all the controlled experiments. In the next section, we will show that WebProphet can achieve high prediction accuracy even when client delays change.

We use `netem` to inject extra RTT and DNS lookup times on the controlled gateway. `netem` is a network emulator in Linux which can add queuing delay to each traversing packet. To inflate RTT, we simply forward web traffic to the controlled gateway while forwarding the DNS traffic to the normal gateway. We may inflate DNS lookup time in a similar way. Unfortunately, `netem` cannot be used to inject extra server response time because it can only add queuing delay to every packet. Instead, we develop our own tool based on `libpcap` and `libdnet` to inflate server response time. Our tool can identify and delay the packets that correspond to the HTTP requests from the agent to the web server for certain amount of time. In effect, this extra delay will be considered as part of the server response time (§4.1). Note that this may trigger TCP timeout on the web agent. Our tool will intercept and drop all the related retransmitted packets.

Manipulating one delay factor at a time: In the first group of experiments, we evaluate the prediction accuracy when we change one delay factor for all the objects across a certain range. We inject five different delay values (100, 150, 200, 250, and 300 ms) to create the baseline scenarios. These values reflect the real delay differences observed from different PlanetLab sites (*e.g.*, those in Asia *vs.* those in the US) to our server at Northwestern University. We use the scenario without any injected delay as the new scenario that we aim to predict. Figure 12 (a)-(c) illustrate the err_c^{50} for the four web services as we change RTT, DNS delay, and server delay. Among the total of 60 experiments, 50% of them have $err_c^{50} \leq 6.1\%$, 90% $err_c^{50} \leq 16.2\%$. The maximum err_c^{50} is 21.7%.

Manipulating RTT and DNS delay together: Next, we evaluate the accuracy of performance prediction when multiple delay factors change simultaneously. We inflate both RTT and DNS delay by 100 ms for all the objects to create the baseline scenario. We still use the scenario without injected delay as the new scenario. Table 2 shows the prediction error for the four applications. t_b^{50}, t_n^{50} and t_p^{50} are the median PLT of the baseline, new, and predicted scenarios. WebProphet can accurately predict not only the median PLT but also the 95th-percentile PLT. The maximum err_c^{50} and err_c^{95} are only 5.5% and 15.9% respectively.

Manipulating only a subset of objects: So far, we have changed the delay factors for all the objects simultaneously. In fact, WebProphet can make accurate predic-

DC	err_c^{50}	err_c^{95}
Akamai	16.0%	11.8%
YDC_1	6.5%	9.7%
YDC_2	14.8%	6.0%

Table 3: Inflating the RTT to different DCs.

tion when we change the delay factors for any subset of objects. When visiting Yahoo Maps from Northwestern University, the web agent will download objects from Akamai CDN and two Yahoo data centers (YDC_1 and YDC_2). In the following experiments, we create three baseline scenarios by injecting 100 ms extra RTT to the objects from Akamai CDN, YDC_1, or YDC_2 respectively. We still use the scenario without injected delay as the new scenario. Our setup is to simulate the case in which the owners of Yahoo Maps want to predict the new PLT if they could reduce the RTT from users to one of their DCs or Akamai CDN. The results in Table 3 show that the prediction errors are reasonably small in all the three experiments.

7.2 Real-world experiments

In the controlled experiments, we changed the delay factors by the same amount for a set of objects. In this section, we conduct experiments on PlanetLab to demonstrate the effectiveness of WebProphet even when each delay factor of each object changes by a different amount simultaneously. The PlanetLab nodes in the US normally experience smaller PLT when accessing Google/Yahoo Search and Maps than those in Asia and Europe. For each of the four web services, we pick one international node as the baseline scenario. We use a node at Northwestern University as the new scenario. This is to simulate the case where the service owners want to predict the new PLT if they could optimize their services for international users in certain way. To predict the new PLT, we replace the timing information of each object in the baseline scenario with the timing information of the same object in the new scenario and then run the page load simulation.

Table 4 shows the locations of the baseline scenario and the prediction errors. Both err_c^{50} and err_c^{95} are within 10.7% for all the four services. We find the prediction errors in the PlanetLab experiments are generally smaller than those in the controlled experiments. Since we directly use the object timing information in the new scenario for PLT prediction in the PlanetLab experiments, the prediction results are no longer affected by the trace collection time differences between the baseline and new scenarios. This suggests our model does capture sufficient level of detail for accurate PLT prediction.

Service	Baseline	New	err_c^{50}	err_c^{95}	No-stream err_c^{50}
Gsearch	Singapore	US	2.0%	10.7%	21.2%
Ysearch	Japan	US	6.1%	0.3%	258.7%
Gmap	Sweden	US	1.2%	1.8%	1.2%
Ymap	Poland	US	0.7%	1.3%	0.1%

Table 4: The results in the real-world experiments.

7.3 Importance of modeling object dependencies

One alternate approach for performance prediction is to measure the PLT under a range of values for each delay factor of each object and then make predictions by extrapolating from these measured PLTs through some form of regression. This may not be feasible for complex webpages with many embedded objects. For instance, even if we measure the PLT only under two different values for each delay factor of each object in Yahoo Maps, we will end up measuring the PLT in 2^{440} scenarios, when considering all the possible combinations of four factors each for 110 objects. Without detailed domain knowledge, it is difficult to decide how many distinct scenarios indeed need to be measured for accurate prediction.

One way to reduce the number of measured scenarios required for prediction is by assuming independence among every delay factor of every object. Let x_i ($i = 1...n$) be the delay factors of all the objects that impact the PLT of a webpage. Under this assumption:

$$f(d_{x1}, d_{x2}, ..., d_{xn}) = \sum_{1}^{n} f_i(d_{xi})$$

Here, d_{xi} denotes the change of delay factor x_i. $f_i(d_{xi})$ is a function that describes the PLT change when only x_i changes. $f(d_{x1}, d_{x2}, ..., d_{xn})$ is a function that describes the PLT change when all the x_i's change simultaneously. In essence, this equation says the PLT change caused by each x_i is independent from the PLT changes caused by other delay factors. If this assumption holds, the number of measured scenarios required for prediction will become linear to the number of delay factors, significantly reducing the measurement overheads. Recently, Chen *et al.* developed a latency prediction tool based on similar assumption [10].

In the following, we study to what extent such independence assumption affects the prediction accuracy. We use the same baseline scenario as that of Table 2 in §7.1, in which both RTT and DNS delay are inflated by 100 ms for all the objects. We still use the scenario without injected delay as the new scenario. For each web service, we divide the objects in the page into three groups (G_1, G_2, and G_3) and subsequently measure the PLT

change δ_i when we only change the delay factors for one group G_i at a time. We then predict the PLT change between the baseline and new scenarios by taking the sum of δ_i's. As shown in column "Indep err_c^{50}" of Table 2, the prediction errors are significantly higher than those of WebProphet for Google/Yahoo Maps. In particular for Yahoo Maps, err_c^{50} is as high as 85.5%. The prediction errors are smaller for Google Search because its webpage has very simple dependencies. Since each δ_i is directly measured instead of being predicted by any model, the prediction error should be close to zero when the delay factors are indeed independent. The results of the experiment highlight the importance of capturing object dependencies for accurate PLT prediction.

7.4 Importance of identifying stream parent

One of the key steps in our PDG extraction is to identify stream parents and dependency offsets (§3.2). We now evaluate the importance of identifying stream parents in prediction accuracy. We use the same baseline and new scenarios as those in the PlanetLab experiments in §7.2. The only difference is that we ignore all the dependencies on stream parents in the PDGs when we make predictions. As shown in column "No-Stream err_c^{50}" in Table 4, the prediction errors without stream parents are much higher than those with stream parents for the Search services. Nonetheless, the prediction errors are roughly the same for the Maps services. This is because the HTML objects account for a significant portion of the Search pages. In contrast, most of the objects in the Maps pages are non-stream ones, *e.g.*, Javascripts and images.

8 Usage Scenarios

As illustrated in the previous sections, the PLT of a complex webpage may depend on the delay factors of many objects. The owner of the page often faces the challenge of finding a cost-effective way to improve service performance from a huge number of possible optimization strategies. Since WebProphet can make fast and accurate prediction under the changes of any delay factor and/or object, it provides the service owner an easy way to narrow down the strategies that could bring the most benefit.

Because Yahoo Maps has the most complex webpage and the largest median PLT (measured from Northwestern University) among all the four services, we use it as an example to demonstrate the power of WebProphet. Though we cannot directly validate the effect of these changes, the experiments described in §7 provide a basis for trust in the predictions. Suppose the owners of Yahoo Maps are considering three methods to optimize the median PLT: i) OPT_{rtt}: reducing the RTT of certain static objects by moving them from Yahoo data centers to the Akamai CDN; ii) OPT_{server}: reducing the server response time by half for certain dynamic objects; and iii) OPT_{client}: reducing the client execution time by half for certain objects. Since the Yahoo Maps page contains about 110 objects including roughly 74 static objects and 36 dynamic ones, it could be too costly to optimize for all of them. Hence, we seek to identify a small set of *candidate objects* whose optimization would lead to significant PLT reduction.

In this paper, we use a simple greedy-based algorithm to search for those candidate objects. In the future, we could also leverage other more sophisticated search algorithms (such as simulated annealing) to obtain better results. Our search algorithm considers one of the optimization methods (OPT_{rtt}, OPT_{server}, or OPT_{client}) at a time. It starts with a list of all the objects and the original object timing information extracted from the page load trace that corresponds to the median of the baseline PLT distribution. At each step, it greedily picks the candidate object whose optimization will lead to the largest PLT reduction among all the remaining objects. It then removes the new candidate object from the list and updates its timing information according to the optimization method. This process terminates when the PLT reduction resulting from the optimization of a new candidate object becomes negligibly small.

After evaluating 2,176 hypothetical scenarios, we identify 5 candidate objects for OPT_{server} and OPT_{rtt} respectively. We also identify 14 candidate objects for OPT_{client}. The predicted PLT reductions by applying OPT_{rtt}, OPT_{client}, and OPT_{server} are 14.8%, 26.6%, and 1.6% respectively. Apparently, OPT_{server} does not seem to be promising, since it can only reduce PLT slightly. The PLT can be further reduced by 40.1% by combining OPT_{rtt} and OPT_{client} together. Therefore, by simply optimizing the client execution time of 14 objects and moving 5 static objects to Akamai CDN, we predict that the median PLT of Yahoo Maps can be cut from 3.99 to 2.39 seconds.

9 Systems Evaluation

We now evaluate the systems overhead of dependency extraction and performance prediction. The dependency extraction process includes two steps: 1) subsequently control the download of each object during a page load; and 2) extract the PDG from the recorded timing information of object download (§3.2). Step 2 is relatively simple. The running time is dominated by step 1 because we need to reload a page many times and artificially delay the download of one object during each page load. Given a page with n objects and m stream objects, we

need to reload the page n times to discover all the descendants and at most $m \times n$ times to discover all the stream parents and dependency offsets (§3.2). All the webpages we have studied so far have only a few stream (HTML) objects. Thus, the running time is roughly linear to the total number of objects in the page. Even for Yahoo Maps which has the most complex PDG, the running time is only two hours. Note that since each controlled page load is independent, we can easily run dependency extraction on multiple machines in parallel to speed up the process.

To predict the PLT of a page, we first need to parse page load traces to extract object timing information and then to simulate page load process under new scenarios (§4). We evaluate the performance predictor on a commodity server with two 2.5 GHz Xeon processors and 16 GB memory running Linux 2.6.18. We use Yahoo Maps as an example because its page incurs the largest prediction time among the four services. The trace parsing time depends on the size of the traces. For Yahoo Maps, it takes 317 seconds to parse 500 page load traces of one scenario with a total size of 455 MB. The page load simulation time depends on the complexity of the PDG. For Yahoo Maps, it takes about 9 ms to run one page load simulation under our current implementation in Python. This translates to a total of 20 seconds simulation time to evaluate all of the 2,176 hypothetical scenarios in §8. We could further optimize the running time by rewriting the simulation code in C/C++.

10 Related Work

There is a large body of prior work on web performance measurement and modeling. For instance, Smith *et al.* leveraged TCP/IP headers in packet traces to characterize the nature of web traffic and the structure of webpages [21]. Nahum *et al.* built an emulator to study the impact of network delay and packet loss on web server performance [17]. Olshefski *et al.* developed techniques for inferring client response time from server-side logs. These works either treat a webpage as a single object or treat each web object independently while ignoring the dependencies between different objects.

Web performance measurement tools, such as Firebug and IBM Page Detailer [2], can provide detailed object timing information of a page load. Nonetheless, they can neither extract object dependencies nor perform PLT predictions.

CPRT [19] used client-side Javascript code to measure user-perceived response times. AjaxScope [15] provided more detailed Javascript code instrumentations to debug client-side errors. Due to the limited information exposed by the browser and OS to the Javascript layer, these approaches cannot reason about the impact of network-layer conditions, such as DNS delay or RTT, on web service performance.

Several existing systems, *e.g.*, Orion [11], eXpose [13], NetMedic [14], and Sherlock [9], employed various techniques to automatically infer causalities between hosts, processes, and network flows. They leveraged these causalities to diagnose performance problems in network applications. In contrast, WebProphet focuses on extracting dependencies between web objects and predicting the performance of web applications.

Wischik used a manually constructed dependency graph of Gmail to study the effects of TCP parameter settings on web performance [25]. In this paper, we formally define the object dependencies of a webpage including ancestors, stream and non-stream parents, and dependency offsets. We further develop an automated system to extract the PDG of a webpage.

11 Conclusion

We built WebProphet, a system that automates performance prediction for web services. The key idea of WebProphet is the use of PDG to encapsulate web object dependencies for accurate and scalable predictions. WebProphet leverages a novel technique based on timing perturbation to extract object dependencies of complex webpages. It implements a simple and yet effective model to simulate the page load process of a web browser, which enables accurate PLT prediction under changes to any web objects. It can also predict the statistical properties of a PLT distribution under a hypothetical scenario. Applying WebProphet to the Search and Maps services of Google and Yahoo, we successfully extract their PDGs and keep the PLT prediction error rates under 16% in most cases. Our results show WebProphet provides a solid foundation for web service providers to quickly find cost-effective optimization strategies for real applications.

References

[1] HTTP pipelining. http://en.wikipedia.org/wiki/HTTP_pipelining.

[2] IBM Page Detailer. http://www.alphaworks.ibm.com/tech/pagedetailer.

[3] IMacros for Firefox - Scripts your Firefox Web Browser. http://www.iopus.com/iMacros/.

[4] Kenote Internet Testing Environment. http://kite.keynote.com/.

[5] Selenium web application testing system. http://seleniumhq.org/.

[6] W3C Document Object Model. http://www.w3.org/DOM/.

[7] Watir - We App Testing in Ruby. http://wtr.rubyforge.org/.

[8] XPCOM Mozilla. `https://developer.mozilla.org/en/XPCOM`.

[9] BAHL, P., CHANDRA, R., GREENBERG, A., KANDULA, S., MALTZ, D. A., AND ZHANG, M. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *ACM SIGCOMM* (2007).

[10] CHEN, S., JOSHI, K., HILTUNEN, M., SCHLICHTING, R., AND SANDERS, W. Link gradients: Predicting the impact of network latency on multi-tier applications. In *IEEE INFOCOM* (2009).

[11] CHEN, X., ZHANG, M., MAO, Z. M., AND BAHL, P. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *OSDI* (2008).

[12] JAISWAL, S., IANNACCONE, G., DIOT, C., , KUROSE, J., AND TOWSLEY, D. Inferring tcp connection characteristics through passive measurements. In *IEEE INFOCOM* (2004).

[13] KANDULA, S., CHANDRA, R., AND KATABI, D. What's going on? extracting communication rules in edge networks. In *ACM SIGCOMM* (2008).

[14] KANDULA, S., MAHAJAN, R., VERKAIK, P., AGARWAL, S., PADHYE, J., AND BAHL, P. Detailed diagnosis in enterprise networks. In *ACM SIGCOMM* (2009).

[15] KICIMAN, E., AND LISHITS, B. Ajaxscope: a platform for remotely monitoring the client-side behavior of web 2.0 applications. In *ACM SOSP* (2007).

[16] KOHAVI, R., HENNE, R. M., AND SOMMERFIELD, D. Practical guide to controlled experiments on the web: Listen to your customers not to the HiPPO. In *ACM KDD* (2007).

[17] NAHUM, E. M., ROSU, M., SESHAN, S., AND ALMEIDA, J. The effects of wide-area conditions on www server performance. In *ACM Sigmetrics* (2001).

[18] PAXSON, V. Bro: A system for detecting network intruders in real-time. *Computer Networks 31*, 23-24 (1999), 2435–2463.

[19] RAJAMONY, R., AND ELNOZAHY, M. Measuring client-perceived response times on the www. In *USENIX Symposium on Internet Technologies and Systems (USITS)* (2001).

[20] SHAH, G., MOLINA, A., AND BLAZE, M. Keyboards and covert channels. In *USENIX Security* (2006).

[21] SMITH, F., HERNANDEZ, F., JEFFAY, K., AND OTT, D. What tcp/ip protocol headers can tell us about the web. In *ACM Sigmetrics* (2001).

[22] SOUDERS, S. *High Performance Web Sites: Essential Knowledge for Front-End Engineers*. O'Reilly Media, 2007.

[23] TARIQ, M., ZEITOUN, A., VALANCIUS, V., FEAMSTER, N., AND AMMAR, M. Answering "what-if" deployment and configuration questions with WISE. In *ACM SIGCOMM* (2008).

[24] VEAL, B., LI, K., AND LOWENTHAL, D. New methods for passive estimation of tcp round-trip times. In *PAM* (2005).

[25] WISCHIK, D. J. Short messages. *Journal Philosophical Transactions of the Royal Society 366*, 1872 (2008).

Mugshot: Deterministic Capture and Replay for JavaScript Applications

James Mickens, Jeremy Elson, and Jon Howell
Microsoft Research
mickens,jelson,jonh@microsoft.com

Abstract

Mugshot is a system that captures every event in an executing JavaScript program, allowing developers to deterministically replay past executions of web applications. Replay is useful for a variety of reasons: failure analysis using debugging tools, performance evaluation, and even usability analysis of a GUI. Because Mugshot can replay every execution step that led to a failure, it is far more useful for performing root-cause analysis than today's commonly deployed client-based error reporting systems—core dumps and stack traces can only give developers a snapshot of the system after a failure has occurred.

Many logging systems require a specially instrumented execution environment like a virtual machine or a custom program interpreter. In contrast, Mugshot's client-side component is implemented entirely in standard JavaScript, providing event capture on *unmodified* client browsers. Mugshot imposes low overhead in terms of storage (20-80KB/minute) and computation (slowdowns of about 7% for games with high event rates). This combination of features—a low-overhead library that runs in unmodified browers—makes Mugshot one of the first capture systems that is practical to deploy to every client and run in the common case. With Mugshot, developers can collect widespread traces from programs in the field, gaining a visibility into application execution that is typically only available in a controlled development environment.

1 Introduction

Despite developers' best efforts to release high quality code, deployed software inevitably contains bugs. When failures are encountered in the field, many programs record their state at the point of failure, e.g., in the form of a core dump, stack trace, or error log. That snapshot is then sent back to the developers for analysis. Perhaps the best known example is the Windows Error Reporting framework, which has collected over a billion error reports from user programs and the kernel [14].

Unfortunately, isolated snapshots only tell part of the story. The root cause of a bug is often difficult to determine based solely on the program's state after a problem was detected. Accurate diagnosis often hinges on an understanding of the events that *preceded* the failure. For this reason, systems like Flight Data Recorder [29], DejaVu [5], and liblog [13] have implemented *deterministic program replay*. These frameworks log enough information about a program's execution to replay it later under the watchful eye of a debugging tool. With a few notable exceptions, these systems require a specially instrumented execution environment like a custom kernel to capture a program's execution. This makes them unsuitable for field deployment to unmodified end-user machines. Furthermore, no existing capture and replay system is specifically targeted for the unique needs of the web environment.

Mugshot's goal is to provide low-overhead, "always-on" capture and replay for web-deployed JavaScript programs. Our key insight is that JavaScript provides sufficient reflection capabilities to log client-side non-determinism *in standard JavaScript* running on *unmodified browsers*. As a user interacts with an application, Mugshot's JavaScript library logs explicit user activity like mouse clicks and "behind-the-scenes" activity like the firing of timer callbacks and random number generation. When the application fetches external objects like images, a server-side Mugshot proxy stores the binary data so that at replay-time, requests for the objects can access the log-time versions.

The client-side Mugshot log is sent to the developer in response to a trigger like an unexpected exception being caught. Once the developer has the log, he uses Mugshot's replay mode to recreate the original JavaScript execution on his unmodified browser. Like the logging library, the replay driver is implemented in

standard JavaScript. The driver provides a "VCR" interface which allows the execution to be run in near-real time, paused, or advanced one event at a time. The internal state of the replaying application can be inspected using unmodified JavaScript debugging tools like Firebug [17]. Developers can also analyze a script's performance or evaluate the usability of a graphical interface by examining how real end-users interacted with it.

At first glance, Mugshot's logging and replay capabilities may seem to introduce a fundamentally new threat to user privacy. However, Mugshot does not create new techniques for logging previously untrackable events—instead, it leverages the preexisting introspective capabilities found in browsers' standard JavaScript engines. Furthermore, the preexisting security policies which prevent cross-site data exchange also prevent Mugshot's event logs from leaking across domains. Thus, the Mugshot log for a particular page can only be accessed by that page's domain.

The rest of the paper is organized as follows. In Section 2, we review related work in capture and replay. We then describe the architecture of Mugshot in Section 3. Section 4 contains our evaluation, which describes microbenchmarks (§4.1) and Mugshot's performance inside several complex, real-world applications (§4.2). Our evaluation shows that Mugshot is unobtrusive at logging time and faithful at replay time, recreating several bugs that we found in our evaluation applications. We consider the privacy implications of Mugshot in Section 5 and then conclude in Section 6.

2 Related Work

Error Reporting from the Field

There are many frameworks for collecting information about crashed programs and sending the data back to the developers. Perhaps the best known is Windows Error Reporting (WER), which has been included in Windows since 1999 and has collected billions of error reports [14]. When a crash, hang, installation failure, or other error is detected, WER creates a *minidump*. Minidumps are snapshots of the system's essential state: register values, thread stacks, lists of loaded modules, a portion of the text segment surrounding the instruction pointer, and other information. With the user's permission, this data is sent back to Microsoft, where it is bucketed according to likely root-cause for later analysis. WER only records the state of the application at the moment of the crash; developers must infer the sequence of events that led to it. Other deployed systems have similar capabilities and constraints, including Firefox's Breakpad [4] and the iPhone OS [1].

Capture and Replay

As described by the survey papers [8] and [10], deterministic replay has been studied for many years in a variety of contexts. Some of the prior systems are machine-wide replay frameworks intended to debug operating systems or cache coherence in distributed shared memory systems [11, 22, 29]. They create high-fidelity execution logs, but with significant cost: the target software must run atop custom hardware, a modified kernel, or a special virtual machine. This limits opportunities for widespread deployment. Furthermore, these systems produce log data at a high rate; generally, these systems can only report the last few seconds of system state before an error.

Moving up the stack, a number of application- or language-specific tools allow deterministic capture and replay. By restricting themselves to high-level interfaces and a single-threaded execution model, they obviate the need to log data at the instruction level. This dramatically reduces the overhead of logging, both in processor time and storage requirements. The liblog system[13] is perhaps closest to our work. liblog provides a C-library interposition layer that records the input and output of all interactions between the application and the C library (and hence, the operating system) below. As with Mugshot, one of liblog's goals was to make logging sufficiently lightweight that it can be run in the common case. Other application-specific logging environments include DejaVu [5] for Java programs and Retrospect [3] for parallel programs written using the MPI interface.

Ripley [20] is a framework for preserving the computational integrity of AJAX applications. Client and server code is written in `.NET`, but Ripley automatically translates the client-side portion into JavaScript for execution in a browser. The instrumented JavaScript sends an event stream to the web server, which replays the events to a server-side replica of the client. The server only executes a client RPC if it is also generated by the replica.

Mugshot differs from Ripley in three important ways. First, Mugshot works on arbitrary JavaScript applications and does not require applications to be developed in a special environment. Second, Ripley's current implementation does not capture all sources of nondeterminism. For example, Ripley does not handle calls to `Date()`. It could treat `Date()` as an RPC and have the client synchronously fetch a value from the server (a value which would also be fed to the server-side client replica). However, this incurs a round trip for each time request, making it infeasible for applications like games that rapidly generate events. Third, for performance reasons, Ripley's server-side client replicas are not actual web browsers—they are lightweight browser emulators that track DOM state (§3.1.3) but do not perform layout

or rendering. In contrast, Mugshot replays events inside the same browser type used at logging time. This greatly increases the likelihood that a buggy execution can be recreated.

Debugging for Web Applications

There are a variety of tools for debugging web applications. For example, Fiddler [21] is a web proxy that allows the local user to inspect, modify, and replay HTTP messages. Firebug [17], an extension to Firefox, is an advanced JavaScript debugger that supports breakpoints, arbitrary expression evaluation, and performance profiling. Internet Explorer 8 has a built-in debugger with similar features. All of these tools provide rich introspection upon the local execution environment. However, none of them provide a way to capture remote bugs in the wild and explore the execution paths that led to faulty behavior.

AjaxScope [18] uses a web proxy to dynamically instrument JavaScript code before sending it to remote clients. Developers express their debugging intent through functions inserted at specific places in the code's abstract syntax tree. For example, to check for infinite loops, a programmer can attach a diagnostic function to each `for`, `while`, and `do-while` statement. Whereas AjaxScope's goal is to let developers express specific debugging policies, Mugshot focuses on recreating entire remote execution contexts.

The commercial Selenium [27] Firefox extension records user activity for later playback. Recording can only be done in Firefox, but playback is portable across browsers using synthetic JavaScript events. Because Selenium does not log the full set of nondeterministic events, it is suitable for automating tests, but it cannot reproduce many nondeterministic bugs.

The commercial products ClickTale [7] and CS SessionReplay [12] capture mouse and keyboard events in browser-based applications. However, neither of these products expose a full, browser-neutral environment for logging *all* sources of browser nondeterminism, including both client-side nondeterminism like timer interrupts and server-side nondeterminism like dynamic image generation. The services provide click analytics and a movie of client-visible interactions, but not the underlying internal state of the JavaScript heap and the browser DOM tree.

3 Design and Implementation

Mugshot's goal is to record the execution of a web application on an end user's machine, then recreate that execution on a developer's machine. To capture application activity, one could exhaustively record every intermediate state of the program. Mugshot instead takes the approach of many other systems: recording all sources of *nondeterminism*. If an application is run again and injected with the same nondeterministic events, the program will follow the same execution path that was observed at logging time.

Past systems have recorded nondeterminism at the instruction level [11] or the `libc` level [13]. However, the former may introduce prohibitive logging overheads, and both require users to modify standard browsers or operating systems. Both approaches also record nondeterminism at a granularity that is unnecessarily fine for JavaScript-driven web applications. JavaScript programs are *non-preemptively single threaded* and *event driven*. Applications register callback functions for events like key strokes or the completion of an asynchronous HTTP request. When the browser detects such an event, it invokes the appropriate application handlers. The browser will never interrupt the execution of one handler to run another. Thus, the execution path of the application is solely determined by the event interleavings encountered during a particular run. This means that logging the *content* and the *ordering* of events provides sufficient information for replay.

Logging nondeterminism at the level of JavaScript events would be easy if we could insert logging code directly into the browser. However, this solution is unappealing to developers since it requires users to download a special browser or install a logging plug-in. Many users will not opt into such a scheme, dramatically reducing the size and diversity of the developer's logging demographic.

To avoid these problems, we implemented the client portion of Mugshot entirely in JavaScript. Compared to an in-browser solution, a JavaScript implementation is more complex and more difficult to make performant. However, it has the enormous advantage of being transparent to users and hence much easier to deploy. As we will see in the sections that follow, JavaScript offers sufficient introspection and self-modification capabilities to enable insertion of shims that log most sources of nondeterminism.

In Section 3.1, we enumerate the sources of nondeterminism in web applications and describe how Mugshot captures them in Firefox and IE. Although conceptually straightforward, the logging process is complicated by various browser incompatibilities and implementation deficiencies, particularly with respect to keyboard events. In Section 3.2, we describe how Mugshot replays an execution by dispatching synthetic events from its log.

3.1 Capturing Nondeterministic Events

To add Mugshot recording to an application, the developer delivers an application through a server-side web proxy. The proxy's first job is to insert a single tag

at the beginning of the application's `<head>` block:

```
<script src='Mugshot.js'></script>
```

When the page loads, the Mugshot library runs before the rest of the application code has a chance to execute. Mugshot interposes on the sources of nondeterminism that we describe below and begins to write to an in-memory log. Event recording continues until the page is closed.

If the application contains multiple frames, the proxy injects the Mugshot `<script>` tag into each frame. Child frames report all events to the Mugshot library running in the topmost frame; this frame is responsible for collating the aggregate event log and sending it back to the developer.

The developer controls when the application uploads event logs. For example, the application may post logs at predefined intervals, or only if an unexpected exception is thrown. Alternatively, the developer may add an explicit "Send error report" button to the application which triggers a log post.

Figure 1 lists the various sources of nondeterminism in web applications. In the sections below, we discuss each of the broad categories and describe how we capture them on Firefox and IE. Our discussion proceeds in ascending order of the technical difficulty of logging each event category.

Importantly, Mugshot does *not* log events for media objects that are opaque to JavaScript code. For example, Mugshot does not record when users pause a Flash movie or click a region inside a Java applet. Since these objects do not expose a JavaScript-accessible event interface, Mugshot can make no claims about their state. The current implementation of Mugshot also does not capture nondeterministic events arriving from opaque containers like Flash's `ExternalInterface`; such events are rarely used in practice.

For each new event that it does capture, Mugshot creates a log entry containing a sequence number and the wall clock time. The entry also contains the event type and enough type-specific data to recreate the event at replay time. For example, for keyboard events, Mugshot records the GUI element that received the event, the character code for the relevant key, and whether any of the shift, alt, control, or meta keys were simultaneously pressed.

3.1.1 Nondeterministic Function Calls

Applications call `new Date()` to get the current time and `Math.random()` to get a random number. To log time queries, Mugshot wraps the original constructor for the `Date` object with one that logs the returned time. To log random number generation, Mugshot replaces the built-in `Math.random()` with a simple linear congruential generator [23]. Mugshot uses the application's load date to seed the generator, and it writes this seed to the log. Given this seed, subsequent calls to the random number generator are deterministic and do not require subsequent log entries.

Our initial implementation of Mugshot did not define a custom random number generator—instead, it simply wrapped `Math.random()` in the same way that it wrapped `Date()`. However, we found that games often made frequent requests for random numbers, e.g., to determine whether a space invader should move up or down. The resulting logs were filled with random numbers and did not compress well (which was important, since uncompressed logs can be large (§ 4.2.1)). Thus, we decided to use the logging scheme described above.

3.1.2 Interrupts

JavaScript interrupts allow applications to schedule callbacks for later invocation. Callbacks can be scheduled for one-time execution using `setTimeout(callback, waitTime)`. A callback can be scheduled for periodic execution using `setInterval(callback, period)`. JavaScript is cooperatively single threaded, so interrupt callbacks (and event handlers in general) execute atomically and do not preempt each other.

Mugshot logs interrupts by wrapping the standard versions of `setTimeout()` and `setInterval()`. The wrapped registration functions take an application-provided callback, wrap it with logging code, and register the wrapped callback with the native interrupt scheduler. Mugshot also assigns the callback a unique id; since JavaScript functions are first class objects, Mugshot stores this id as a property of the callback object. Later, when the browser invokes the callback, the wrapper code logs the fact that a callback with that id executed at the current wall clock time.

Although simple in concept, IE does not support this straightforward interposition on `setTimeout()` and `setInterval()`. Mugshot's modified `setTimeout()` must hold a reference to the browser's original `setTimeout()` function; however, IE sometimes garbage collects this reference, leading to a "function not defined" error from the Mugshot wrapper. To mitigate this problem, Mugshot creates an invisible `<iframe>` tag, which comes with its own namespace and hence its own references to `setTimeout()` and `setInterval()`. The Mugshot wrapper invokes copies of these references when it needs to schedule a wrapped application callback.

Although this trick gives Mugshot references to the native scheduling functions, it prevents Mugshot from actually scheduling callbacks until the hidden frame

	Event type	Examples	Captured by Mugshot
DOM Events §3.1.3	Mouse	click, mouseover	Yes
	Key	keyup, keydown	Yes
	Loads	load	Yes
	Form	focus, blur, select, change	Yes
	Body	scroll, resize	Yes
Interrupts §3.1.2	Timers	setTimeout(f, 50)	Yes
	AJAX	req.onreadystatechange = f	Yes
Nondeterministic functions §3.1.1	Time query	(new Date()).getTime()	Yes
	Random number query	Math.random()	Yes
Text selection §3.1.8	Firefox: window.getSelection()	Highlighting text w/mouse	Yes
	IE: document.selection	Highlighting text w/mouse	Partially
Opaque browser objects §3.1	Flash movie	User pauses movie	No
	Java applet	Applet updates the screen	No

Figure 1: Sources of nondeterminism in browsers.

is loaded. This problem has three cascading consequences. First, since JavaScript is single-threaded, Mugshot cannot block until the hidden frame is loaded without hanging the application. Instead, it must queue application timer requests and install them once the hidden frame loads. Second, `setTimeout()` and `setInterval()` return opaque scheduling identifiers that the application can use to cancel the callback via `clearTimeout()` and `clearInterval()`. For interrupt registrations issued before the hidden frame loads, Mugshot cannot call the real registration functions to get cancellation ids. So, Mugshot generates synthetic identifiers and maintains a map to the real identifiers it acquires later. Third, an application may cancel an interrupt before the hidden frame loads; Mugshot responds by simply removing the callback from its queue of requests.

AJAX requests allow JavaScript applications to issue asynchronous web fetches. The browser represents each request as an *XMLHttpRequest* object. To receive notifications about the status of the request, applications assign a callback function to the object's `onreadystatechange` property. The browser invokes this function whenever new data arrives or the entire transmission is complete. Upon success or failure, the various properties of the object contains the status code for the transfer (e.g., `200 OK`) and the fetched data.

Mugshot must employ different techniques to wrap AJAX callbacks on different browsers. On Firefox, Mugshot's wrapped `XMLHttpRequest` constructor registers a DOM Level 2 handler (§3.1.3) for the object's `onreadystatechange` event. IE does not support DOM Level 2 handlers on AJAX objects, so Mugshot interposes on the object's `send` method to wrap the application handler in logging code before the browser issues the request. We describe DOM Level 2 handlers in more detail in the next section.

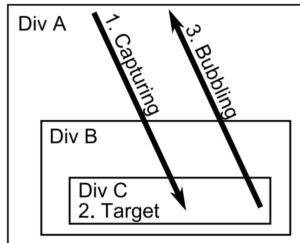
For each AJAX event, Mugshot logs the current state of the request (e.g., "waiting for data"), the HTTP headers, and any incremental data that has already returned. Once the request has completed, Mugshot logs the HTTP status codes. We also log the raw request data on the server-side replay proxy (§3.2.1). For the purposes of replay, this data only needs to be logged on one side. Thus, our current implementation consumes more space than strictly necessary. However, it makes the client-side and proxy-side logs more understandable to human debuggers, since AJAX activity in one log does not have to be collated with data from the other log.

3.1.3 DOM Events

The Document Object Model (or DOM) is the interface between JavaScript applications and the browser's user interface [28]. Using DOM calls, JavaScript applications register handlers for user events like mouse clicks. DOM methods also allow the application to dynamically modify page content and layout.

The browser binds every element in a page's HTML to an application-accessible JavaScript object. Applications attach event handlers to these DOM objects, informing the browser of the application code that should run when a DOM node generates a particular event. In the simplest handler registration scheme, applications simply assign functions to specially-named DOM node properties. For example, to execute code whenever the user clicks on a `<div>` element, the application assigns a function to the `onclick` property of the corresponding JavaScript DOM node.

This simple model, called DOM Level 0 registration, only allows a single handler to be assigned to each DOM node/event pair. Modern browsers also implement the DOM 2 model, which allows an application to register multiple handlers for a particular DOM node/event pair.




```
<div onclick='handlerA()'>
   <div onclick='handlerB()'>
      <div onclick='handlerC()'>
      </div>
   </div>
</div>
```

Figure 2: Event handling after a user clicks within Div C. In the W3C model's capturing phase, handlerA() is called if it is a capturing handler, followed by handlerB() if it is a capturing handler. After the target's handlerC() is called, W3C mandates a bubbling phase in which handlers marked as bubbling are called from the inside out.

An application calls the node's attachEvent() (IE) or addEventListener() (Firefox) method, passing an event name like "click", a callback, and in Firefox, a useCapture flag to be discussed shortly.

The World Wide Web Consortium's DOM Level 2 specification [28] defines a three-phase dispatch process for each event (Figure 2). In the *capturing* phase, the browser hands the event to the special window and document JavaScript objects. The event then traces a path down the DOM tree, starting at the top-level <html> DOM node and eventually reaching the DOM node that actually generated the event. The boolean parameter in addEventListener() allows a handler to be specified as capturing. Capturing handlers are only executed in the capturing phase; they allow a DOM node to execute code when a child element has generated an event. Importantly, the ancestor's handler will be called *before* any handlers on the child run.

In the *target* phase, the event is handed to the DOM node that generated it. The browser executes the appropriate handlers at the target, and then sends the event along the reverse capturing path. In this final *bubbling* phase, ancestors of the target can run event handlers marked as bubbling, allowing them to process the event after it has been handled by descendant nodes.

In the DOM 2 model, some event types are cancelable, i.e., an event handler can prevent the event from continuing through the three phase process. Also, although all events capture, some do not bubble. For example, load events, which are triggered when an image has completely downloaded, do not bubble. Form events also do not bubble. Examples of form events include focus and blur, which are triggered when a GUI element like a text box gains or loses input focus.

3.1.4 DOM Events and Firefox

Firefox supports the W3C model for DOM events. Thus, Mugshot can record these events in a straightforward way—it simply attaches capturing logging handlers to the window object. Since the window object is the highest ancestor in the DOM event hierarchy, Mugshot's logging code is guaranteed to catch every event before it has an opportunity to be canceled by other nodes in the capture, target, or bubble phases. Note that canceled events still need to be logged, since they caused at least one application handler to run!

Mugshot must ensure that the application does not accidentally delete or overwrite Mugshot's logging handlers. To accomplish this, Mugshot registers the logging handlers as DOM 2 callbacks, exploiting the fact that applications cannot iterate over the DOM 2 handlers for a node, and they cannot deregister a DOM 2 handler via domNode.detachEvent(eventName, callback) without knowing the callback's function pointer.

Mugshot must also ensure that its DOM 2 window handlers run before any application-installed window handlers execute and potentially cancel an event. Firefox invokes a node's DOM 2 callbacks in the order that they were registered; since the Mugshot library registers its handlers before any application code has run, its logging callbacks are guaranteed to run before any application-provided DOM 2 window handlers.

Unfortunately, Firefox invokes any DOM 0 handler on the node before invoking the DOM 2 handlers. To ensure that Mugshot's DOM 2 handler runs before any application-provided DOM 0 callback, Mugshot uses JavaScript setters and getters to interpose on assignments to DOM 0 event properties. Setters and getters define code that is bound to a particular property on a JavaScript object. The setter is invoked on read accesses to the property, and the getter is invoked on writes.

Ideally, Mugshot would define a DOM 2 logging handler for each event type e, and create setter code for the window.e property which wrapped the the user-specified handler with a Mugshot-provided logging function. If the application provided no DOM 0 handler, Mugshot's DOM 2 callback would log the event; otherwise, the wrapped DOM 0 handler would log the event and set a special flag on the event object indicating that Mugshot's DOM 2 handler should not duplicate the log entry. Unfortunately, this scheme will not work because Firefox's getter/setter implementation is buggy. Mugshot can create a getter/setter pair for a DOM node event prop-

erty, and application writes to the property will properly invoke the setter. However, when an actual event of type `e` is generated, the browser will not invoke the associated function. In other words, the setter code, which works perfectly at the application level, hides the event handler from the internal browser code.

Luckily, the setter code does not prevent the browser from invoking DOM 2 handlers. Thus, Mugshot's setter also registers the application-provided handler as a DOM 2 callback. The setter code ensures that when the application overwrites the DOM 0 property name, Mugshot deregisters the shadow DOM 2 version of the old DOM 0 handler.

When Mugshot logs a DOM event, it records an identifier for the DOM node target. If the target has an HTML id, e.g., `<div id='foo'>`, Mugshot tags the log entry with that id. Otherwise, it identifies the target by specifying the capturing path from the root `<html>` tag. For example, the id $(1, 5)$ specifies that the target can be reached by following the first child of the `<html>` tag and then the fifth child of that node. Since many JavaScript applications use dynamic HTML, the path for a particular node may change throughout a program's execution. Thus, the determination of a target's path id must be done at the time the event is seen—it cannot be deferred until (say) the time that the log is posted to the developer.

3.1.5 DOM Events and IE

IE's event model is only partially compatible with the W3C one. The most important difference is that IE does not support the capturing phase of event propagation. This introduces two complications. First, an event may never have an opportunity to bubble up to a `window`-level logging handler—the event might be canceled by a lower-level handler, or it may be a non-bubbling event like a load. Second, even if an event bubbles up to a `window`-level logger, the event may have triggered lower-level event handlers and generated loggable nondeterministic events. For example, a mouse click may trigger a target-level callback that invokes `new Date()`. The mouse click is temporally and causally antecedent to the time query. However, the mouse click would be logged *after* the time query, since the time query is logged at its actual generation time, whereas the mouse click is logged after it has bubbled up to the `window`-level handler.

Mugshot addresses these problems using several techniques. To log non-bubbling events, Mugshot exploits IE's facility for extending the object prototypes for DOM nodes. For DOM types like `Images` and `Inputs` which support non-bubbling events, Mugshot modifies their class definitions to define custom setters for DOM 0 event properties. Mugshot also redefines `attachEvent()` and `detachEvent()`, the mechanisms by which applications register DOM 2 handlers for these nodes. The DOM 0 setters and the wrapped DOM 2 registration methods collaborate to ensure that if an application defines at least one handler for a DOM node/event pair, Mugshot will log relevant events precisely once, and before any application-specified handler can cancel the event.

Ideally, Mugshot could use the same techniques to capture bubbling events at the target phase. Unfortunately, IE's DOM extension facility is fragile: redefining certain combinations of DOM 0 properties can cause unpredictable behavior. Therefore, Mugshot uses `window`-level handlers to log bubbling events; this is the problematic technique described above that may lead to temporal violations in the log. Fortunately, Mugshot can mitigate this problem in IE, because IE stores the current DOM event in a global variable `window.event`. Whenever Mugshot needs to log a source of nondeterminism, it first checks whether `window.event` is defined and refers to a not-yet-logged event. If so, Mugshot logs the event before examining the causally dependent event.

An application may cancel a bubbling event before it reaches Mugshot's `window`-level handler by setting its `Event.cancelBubble` property to `true`. The event still must be logged, so Mugshot extends the class prototype for the `Event` object, overriding its `cancelBubble` setter to log the event before its cancellation.

In summary, Mugshot on IE logs all bubbling DOM events, but only the non-bubbling events for which the application has installed handlers. This differs from Mugshot's behavior on Firefox, where it logs all DOM events regardless of whether the application cares about them. Recording these "spurious" events does not affect correctness at replay time, but it does increase log size. Fortunately, as we show in Section 4.2.1, Mugshot's compressed logs are small enough that the storage penalty for spurious events is small. Thus, we were not motivated to implement an IE-style logging solution for Firefox—it was technically feasible, but comparatively much more difficult to implement correctly than our capturing-handler solution.

3.1.6 Handling Load Events on IE

In IE, `load` events do not capture or bubble. Using the techniques described in the previous section, Mugshot can capture these events for elements with application-installed `load` handlers. However, Mugshot actually needs to capture *all* load events so that at replay time, it can render images in the proper order and ensure that

the page layout unfolds in the same fashion observed at logging time. Otherwise, an application that introspects the document layout may see different intermediate results at replay time.

Ideally, Mugshot could modify the prototype for `Image` objects such that whenever the browser created an `Image` node, the node would automatically install a logging handler for `load` events. Unfortunately, prototype extension only works for properties and methods accessed by application-level code—the browser's native code creation of the DOM node cannot be modified by extending the JavaScript-level prototype. So, Mugshot uses a hack: whenever it logs an event, it schedules a timeout to check whether that event has created new `Image` nodes; if so, Mugshot explicitly adds a DOM 2 logging handler which records load events for the image. Mugshot specifies this callback by invoking a non-logged `setTimeout(imageCheck, 0)`. The 0 value for the timeout period makes the browser invoke the `imageCheck` callback "as soon as possible." Since the timeout is set from the context of an event dispatch, the browser will invoke the callback immediately after the dispatch has finished, but before the dispatch of other queued events (such as the `load` of an image that we want to log). Mugshot also performs this image check at the end of the initial page parse to catch the loading of the page's initial set of images.

3.1.7 Synthetic Events

Applications call `DOMnode.fireEvent()` on IE and `DOMnode.dispatchEvent()` on Firefox to generate synthetic events. Mugshot uses these functions at replay time to simulate DOM activity from the log. However, the application being logged can also call these functions. These synthetic events are handled synchronously by the browser; thus, from Mugshot's perspective, they are deterministic program outputs which do not need to be logged. However, in terms of the event dispatching path, the browser treats the fake events just like real ones, so they will be delivered to Mugshot's logging handlers.

To prevent these events from getting logged on Firefox, Mugshot interposes on `document.createEvent()`, which applications must call to create the fake event that will be passed to `dispatchEvent()`. The interposed `document.createEvent()` assigns a special `doNotLog` property to the event before returning it to the application. Mugshot's logging code will ignore events that define this property.

This technique does not work on IE, which prohibits the addition of new properties to the Event object. Thus, Mugshot uses prototype extension to interpose on `fireEvent()`. Inside the interposed version, Mugshot pushes an item onto a stack before calling the native `fireEvent()`. After the call returns, Mugshot pops an item from the stack. In this fashion, if Mugshot's logging code for DOM events notices a non-empty stack, it knows that the current DOM event is a synthetic one and should not be logged.

3.1.8 Annotation Events

At replay time, Mugshot dispatches synthetic DOM events to simulate user GUI activity. These events are indistinguishable from the real ones with respect to the dispatch cycle—given a particular application state, a synthetic event will cause the exact same handlers to execute in exactly the same order as a semantically equivalent real event. However, we noticed that synthetic events did not always update the visible browser state in the expected way. In particular, we found the following problems on both Firefox and IE:

- According to the DOM specification, when a `keypress` event has finished the dispatch cycle, the target text input or content-editable DOM node should be updated with the appropriate key stroke. Our replay experiments showed that this did not happen reliably. For example, synthetic key events could be dispatched to a text entry box, but the value of the box would not change, despite the fact that the browser invoked all of the appropriate event handlers.
- `<select>` tags implement drop-down selection lists. Each selectable item is represented by an `<option>` tag. Dispatching synthetic mouse clicks to `<option>` nodes should cause changes in the selected item property of the parent `<select>` tag. Neither Firefox nor IE provided this behavior.
- Users can select text or images on a web page by dragging the mouse cursor or holding down the shift key while tapping a directional key. The browser visibly represents the selection by highlighting the appropriate text and/or images. The browser internally represents the selected items as a range of underlying DOM nodes. Applications access this range by calling `window.getSelection()` on Firefox and inspecting the `document.selection` object on IE. We found that dispatching synthetic key and mouse events did not reliably update the browser's internal selection range, and it did not reliably recreate the appropriate visual highlighting.

To properly replay these DOM events, Mugshot defines special *annotation events*. Annotation events are "helpers" for events which, if replayed by themselves, would not produce a faithful recreation of the logging-

time application state. Mugshot inserts an annotation event into the log immediately after a DOM event which has low fidelity replay. At replay time, Mugshot dispatches the low fidelity synthetic event, causing the appropriate event handlers to run. Mugshot then executes the associated annotation event, finishing the activity induced by the prior DOM event. Annotation events are not real events, so they do not trigger application-defined event handlers. They merely describe work that Mugshot must perform at replay time to provide faithful emulation of logging-time behavior.

To fix low-fidelity `keypress` events on text inputs, Mugshot's `keypress` logger schedules a timeout interrupt with an expiration time of 0. The browser executes the callback immediately after the end of the dispatch cycle for the `keypress`, allowing Mugshot to log the value of the text input. At replay time, after dispatching the synthetic `keypress`, Mugshot reads the value annotation from the log and programmatically assigns the value to the target DOM node's `value` property.

To ensure that clicks on `<option>` elements actually update the chosen item for the parent `<select>` tag, Mugshot's `mouseup` logger checks whether the event target is an `<option>` tag. If so, this indicates that the user has selected a new choice. Mugshot generates an annotation indicating which of the `<select>` tag's children was clicked upon. At replay time, Mugshot uses the annotation to directly set the `selectedIndex` property of the `<select>` tag.

Mugshot generates annotation events for selection ranges after logging `keyup` and `mouseup` events. On Firefox, the selection object conveniently defines a starting DOM node, a starting position within that node, an ending DOM node, and an ending position within that node. Mugshot simply adds the relevant DOM node identifiers and integer offsets to the annotation record. Abstractly speaking, Mugshot includes the same information for an annotation record on IE. However, IE does not provide a straightforward way to determine the exact extent of a selection range. So, Mugshot must cobble together several IE range primitives to deduce the current range. Mugshot first determines the highest enclosing parent tag for the currently selected range. Then, Mugshot creates a range which covers all of the parent tag's children, and progressively shrinks the number of HTML characters it contains, using IE's `Range.inRange()` to determine whether the actual selection region resides within the shrinking range. At some point, Mugshot will determine the exact amounts by which it must shrink the left and right margins of the parent range to precisely cover the actual selected region. Mugshot logs the DOM identifier for the parent node and the left and right pinch margins.

IE's selection semantics are extremely complex, and we have not produced a complete formal specification for them. Since Mugshot cannot currently reproduce these semantics in all applications, Figure 1 lists Mugshot's support for IE selection events as partial.

3.1.9 Performance Optimizations

Both Firefox and IE support the W3C `mousemove` event, which is fired whenever the user moves the mouse. Mugshot can log this event like any other mouse action, but this can lead to unnecessary log growth in Firefox if the application does not care about this event (remember that Mugshot on Firefox logs all DOM events, regardless of application interest in them). Mugshot logs `mousemove` by default, but since few applications use `mousemove` handlers, the developer can disable its logging to reduce log size. In Section 4.2, we evaluate Mugshot's log size for a drawing application that *does* use `mousemove` events.

Games which have high rates of keyboard or mouse activity may generate many content selection annotation events. Generating these annotations is expensive on IE since Mugshot has to experimentally determine the selection range (see Section 3.1.8). Furthermore, games do not typically care about the selection zones that their GUI inputs may or may not have created. Thus, for games with high event rates like Spacius [16], we disable annotations for content selection.

3.2 Replay

Compared to the logging process, replay is straightforward. The most complexity arises from replaying load events, since JavaScript code cannot modify the network stack and stall data transmissions to recreate logging-time load orderings. Thus, Mugshot coordinates load events with a transparent caching proxy that the developer inserts between his web server and the outside world.

In addition, Mugshot must also shield the replaying execution from *new* events that arise on the developer machine, e.g., because the developer accidentally clicks on a GUI element in the replaying application. Without a barrier for such new events, the replaying program may diverge from the execution path seen at logging time.

3.2.1 Caching Web Content at Logging Time

When a user fetches a page logged by Mugshot, the fetch is mediated by a transparent Mugshot proxy. The proxy assigns a session ID to each page fetch; this ID is stored in a cookie and later written to the Mugshot log. As the proxy returns content to the user, it updates a

per-session cache which maps content URLs to the data that was served for those URLs during that particular session. Optionally, the proxy can rewrite static `<html>` and `<frame>` declarations to include the Mugshot library's `<script>` tag.

3.2.2 Replaying Load Events

At replay time, the developer switches the proxy into replay mode, sets the session ID in his local Mugshot cookie to the appropriate value, and directs his web browser to the URL of the page to replay. The proxy extracts the session ID from the cookie, determining the cache it should use to serve data. The proxy then begins to serve the application page, replacing any static `<script>` references to the logging Mugshot library to references to the Mugshot replay library.

During the HTML parsing process, browsers load and execute `<script>` tags synchronously. The Mugshot replay library is the first JavaScript code that the browser runs, so Mugshot can coordinate load interleavings with the proxy before any load requests have actually been generated. During its initialization sequence, Mugshot fetches the replay log from the developer's log server and then sends an AJAX request to the proxy indicating that the proxy should only complete the loads for subsequent non-`<script>` objects in response to explicit "release load" messages from Mugshot.

The rest of the page loads, with any `<scripts>` loading synchronously. The browser may also launch asynchronous requests for images, frame source, etc. These asynchronous requests queue at the proxy. Later, as the developer rolls forward the replay, Mugshot encounters load events for which the corresponding browser requests are queued at the server. Before signaling the proxy to transmit the relevant bytes, Mugshot installs a custom DOM 2 `load` handler for the target DOM node so that it can determine when the load has finished (and thus when it is safe to replay the next event).

3.2.3 The Replay Interface

At replay initialization time, Mugshot places a semi-transparent `<iframe>` overlaying the application page. This frame acts as a barrier for keyboard and mouse events, preventing the developer from issuing events to the replaying application that did not emerge from the log. We embed a VCR-like control interface in the barrier frame which allows the developer to start or stop replay (see Figure 3). The developer can single-step through events or have Mugshot dispatch them at fixed intervals. Mugshot can also try to dispatch the events in real time, although "real-time" playback of applications with high event rates may have a slowdown factor of 2 to 4 times (see Section 4.2.2).

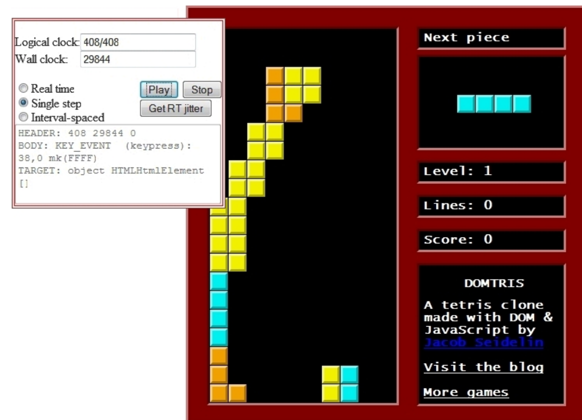


Figure 3: Replaying Tetris (VCR control on left)

Whenever Mugshot replays an event, it can optionally place a small, semi-translucent square above the target DOM node. These squares are color-coded by event type and fade over time. They allow the developer to visually track the event dispatch process, and are particularly useful for understanding mouse movements.

3.2.4 Replaying Non-`load` Events

Replaying events is much simpler than logging them. To replay a non-`load` DOM event, Mugshot locates the target DOM node and dispatches the appropriate synthetic event. For low-fidelity events (§3.1.8), Mugshot also performs the appropriate fix-ups using annotation records. To replay text selection events, Mugshot recreates the appropriate range object and then uses a browser-specific call to activate the selection.

To replay timeout and interval callbacks, Mugshot's initialization code interposes on `setTimeout()` and `setInterval()`. The interposed versions tag each application-provided callback with an interrupt ID and add the callback to a function cache. This cache is built in the same order that IDs were assigned at logging time, so replay-time interrupt IDs are guaranteed to be faithful. Mugshot does not register the application-provided callback with the native interrupt scheduler. Instead, when the log indicates that an interrupt should fire, Mugshot simply retrieves the appropriate function from its cache and executes it. Mugshot interposes on the cancellation functions `clearTimeout()` and `clearInterval()`, but the interposed version are no-ops—once the application cancels an interrupt at logging time, Mugshot will never encounter it again in the log.

Mugshot also interposes on the `XMLHttpRequest` constructor. Much like interrupt replay, Mugshot stores AJAX callbacks in a function cache and executes them at the appropriate time. Mugshot updates each synthetic

AJAX object with the appropriate log data before invoking the application AJAX handler.

By interposing on the `Date()` constructor, Mugshot forces time queries to read values from the log. The log also contains the initialization seed used by the random number generator at capture time. Mugshot uses this value to seed the replay-time generator. This is sufficient to ensure that subsequent calls to `Math.random()` replay faithfully.

3.3 Limitations

Mugshot uses a caching proxy to reproduce the load events in the log. If an application fetches external content that does not pass through the proxy, Mugshot cannot guarantee faithful replay of its data or its load time. Thus, these ill-defined loads can ruin the fidelity of the entire replay.

As described in Section 3.1.8, Mugshot must use annotation records to properly replay GUI events involving drop-down boxes. Although the replay is correct from the perspective of a `<select>` tag's internal JavaScript state, both IE and Firefox refuse to visually drop-down a drop-down list in response to synthetic events. However, after Mugshot applies the annotation event, the visual display of the tag adjusts to indicate the appropriate selection.

Web applications typically fail because of unexpected interactions between HTML, CSS, and event-driven JavaScript code. Mugshot logs all of these application inputs and the associated event streams. In many cases, this is sufficient to recreate a bug; the log-time browser need not be the exact same type and version as the replay-time browser. However, some bugs arise from an interaction between the application and a *specific* browser type and version. In these cases, it is crucial for the log-time and replay-time browsers to be the same. Mugshot's client-side component records the identity of its log-time browser (e.g., Firefox 3.5) so that the developer can run the same browser at debug time. However, even this may be insufficient to recreate some bugs—users can install browser plug-ins or change local configuration state, and the existence of that particular local state may be the root cause of a bug. Since this type of client state cannot be introspected by JavaScript code, Mugshot cannot reliably reproduce these kinds of bugs.

If a web page contains multiple frames, proper logging requires each frame to contain the Mugshot logging `<script>` tag. Similarly, at replay time, each frame must contain Mugshot's replay script. The replay proxy can automatically instrument statically declared frames. However, if a page dynamically creates frames, e.g., using JavaScript, the developer is responsible for inserting the appropriate Mugshot tags.

4 Evaluation

For Mugshot to be useful, it must be unobtrusive at logging time and faithful to the original program execution at replay time. If event logging makes programs sluggish, users will reject Mugshot-enabled applications; if Mugshot cannot reproduce real bugs, it provides no utility to application developers. In this section, we run Mugshot on a variety of microbenchmarks and real JavaScript programs, demonstrating that user-perceived logging overhead is no worse than 6.8% for applications with high event rates. We provide two examples of bugs that Mugshot can log and then reproduce at replay time. We also demonstrate that replay speed is not unacceptably slow, and that events logs grow no faster than 100 KB per minute in applications with high event rates.

All experiments ran on an HP xw46000 workstation with a dual-core 3GHz CPU and 4 GB of RAM. We tested Mugshot performance inside two browsers, Firefox v3.5.3 and IE8 v8.0.6001. When stripped of extraneous white space and comments, Mugshot's logging code was 46 KB and its replay code was 35 KB. Note that only the logging code must be shipped to end users, and only the replay code must be shipped to debugger machines.

4.1 Microbenchmarks

To explore the basic computational overheads of logging and replay, we inserted Mugshot into several microbenchmark applications. For each microbenchmark, we compared its run time without Mugshot support to its run time during logging and replay. We used the following test suite:

- DeltaBlue is a constraint solver from Google's V8 JavaScript benchmark suite [15]. The benchmark is computationally intensive, but it has no user interface, and it does not internally generate loggable events. Thus, DeltaBlue's Mugshot-enabled running time reflects any penalty that Mugshot imposes on straightline computational workloads.
- The `Date` benchmark simply called `new Date()` 5000 times.
- In baseline and logging mode, the `click` benchmark dispatched 5000 synthetic mouse events as quickly as possible. As explained in Section 3.1.7, Mugshot normally does not log synthetic GUI events since they are deterministic. However, for the logging part of the benchmark, we forced Mugshot to log the synthetic mouse clicks. At replay time, Mugshot simply tried to dispatch these logged events as quickly as possible.
- The `setTimeout` benchmark issued 25 calls to `setTimeout(function(){}, 0)`. If the computational overheads of logging and replaying a null function are high, fewer interrupts can be issued per unit time.

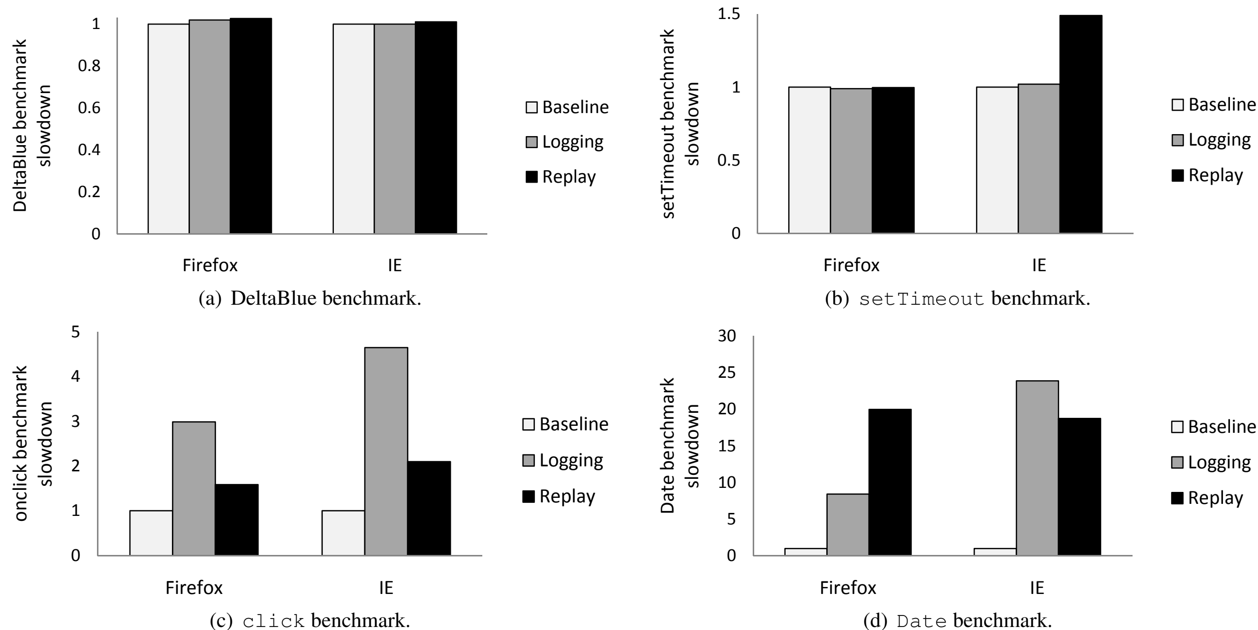


Figure 4: Microbenchmark slowdown due to logging and replay.

Figure 4 shows the Mugshot-induced slowdowns for the test suite. Each graph depicts a benchmark's execution time in baseline, logging, and replay scenarios; performance is normalized with respect to baseline performance. Each result represents the average of 10 trials, and in all cases, standard deviations were less than 5%.

As expected, Figure 4(a) shows that Mugshot introduced no overhead for a purely computational workload. Figure 4(b) demonstrates that logging activity did not delay interrupt scheduling in Firefox and IE. However, replay did introduce a 50% slowdown on IE; we are still investigating the reasons for this behavior.

As shown in Figure 4(c), logging penalties slowed the `click` benchmark by a factor of 3 on Firefox and 4.6 on IE. The overhead primarily arose from the complex logic needed to properly log mouse events on `<select>` elements (Section 3.1.8). During replay, the `click` benchmark slowed by a factor of 1.6 on Firefox and 2.1 on IE. Most of the slowdown was caused by regular expression computations during the parsing of the each `click` log entry. An optimized version of Mugshot would parse the entire log at replay initialization time. However, as we show in Section 4.2, our unoptimized Mugshot can already replay real applications at a tolerable rate.

Figure 4(d) shows the Mugshot penalties for the `Date` microbenchmark. The slowdown factors are large, ranging from 8.4 to 23.9. The reason is that fetching the current date in the baseline case is extremely fast— it merely requires a read of a native browser variable. User-level JavaScript code is much slower than native code, so Mugshot's `Date()` logging introduces high relative overheads. Fortunately, Section 4.2 shows that real applications do not issue time queries at a high enough rate to expose Mugshot's logging overhead.

4.2 Application Examples

To evaluate Mugshot's performance in more realistic conditions, we examined its logging and replay overheads for seven applications. Three of the applications were games with varying rates of event generation.

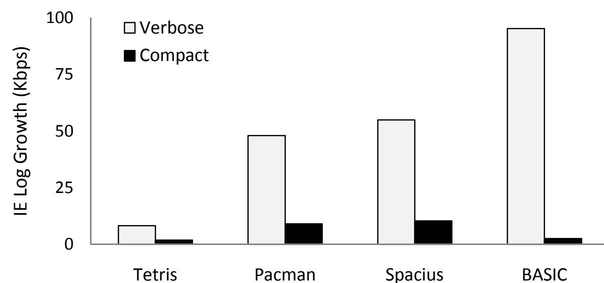
- DOMTRIS [26] is a JavaScript implementation of the classic Tetris game.
- Pacman [6] is unsurprisingly a Pacman clone.
- Spacius [16] is a 2D side-scrolling space shooter.

These games implement many of their animations using interrupt callbacks, so frame rates (and the user experience) will suffer if Mugshot introduces too much latency to the critical path of interrupt dispatch.

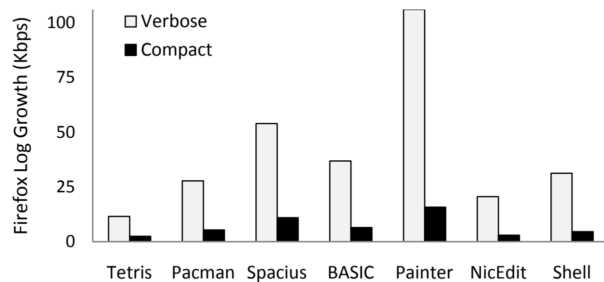
We also evaluated Mugshot's performance on four non-games:

- The Applesoft BASIC interpreter [2] parses and runs BASIC programs, providing an emulated joystick and graphical display.
- NicEdit [19] is a WYSIWYG text and HTML editor.
- Painter [24] is a simple drawing program.
- The JavaScript shell [25] provides a command-line interface for manipulating the DOM and application-defined JavaScript state.

These programs stress Mugshot's handling of form, key, and selection events. Painter also makes use of the

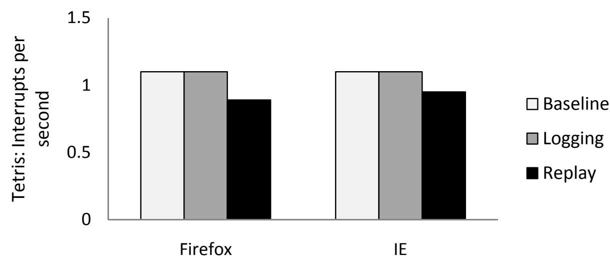


(a) IE applications.

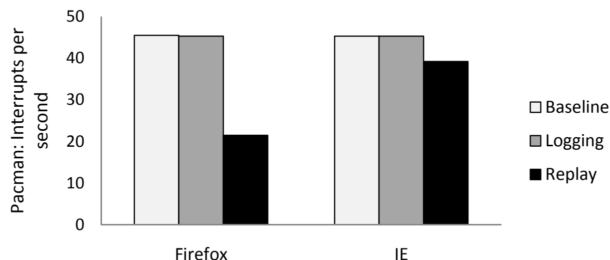


(b) Firefox applications.

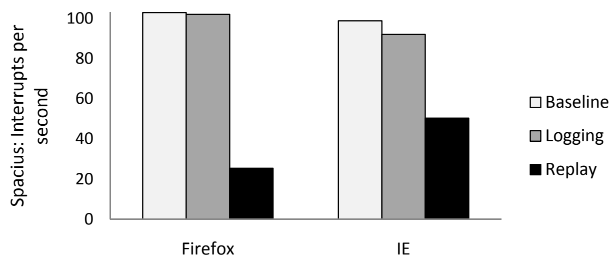
Figure 5: Growth rate of logs (kilobits per second).



(a) Tetris.



(b) Pac-man.



(c) Spacius.

Figure 6: Interrupt rates.

`mousemove` event, so we configured Mugshot to log those events for this application.

We evaluated Mugshot's performance in the Firefox browser for each of the seven applications. However, we only evaluated Mugshot's IE performance for the first four applications. The latter three applications do not replay correctly in IE; they trigger quirks in IE's form/selection event model that Mugshot does not currently handle.

4.2.1 Log Sizes

Figure 5 depicts the growth rate of Mugshot's log for each application, showing the size of the verbose log and the compact log. The verbose log has a human-friendly format; among other things, it contains a dump of the page's HTML at load time, and it explicitly tags each event with an easy-to-understand string representing the event type and its parameters. In our experiences, just reading the verbose log can provide a human debugger with invaluable insights about program operation. The compact log discards the beautifications of the verbose log and represents events and their parameters using short status codes. The compact log is also compressed using the LZW algorithm with a window size of 200.

Figure 5 shows that the rates of uncompressed log growth varied widely, from roughly 10 Kbps (Tetris) to 106 Kbps for Painter on Firefox and 95 Kbps for the BASIC interpreter on IE. Figure 5 also shows that Mugshot's log compression is effective, with a worst case compressed growth rate of 15.7 Kbps for the Painter application. Painter's logs were comparatively large because Mugshot had to log frequent `mousemove` events. Spacius generated the second highest growth rates (10.9 Kbps KB on Firefox and 10.4 Kbps on IE). As we discuss in the next section, this was due to Spacius' high rate of interrupt events.

4.2.2 Logging and Replay Overheads

For Mugshot to be practical, its logging overhead must have a minimal impact on the user experience. It is less important for Mugshot to be able to replay application traces in real time. However, replaying should not be so slow that debugging is painful for a developer.

Games update the screen in response to GUI events like mouse clicks. However, for graphically intense games, most of the screen updates are driven by timer interrupt callbacks. The dispatch rate of these callbacks provides a natural metric for Mugshot-induced slowdowns—the more overhead that Mugshot creates, the slower these callbacks execute, and the more sluggish the application appears.

Figure 6 shows the interrupt dispatch rate for the three games on IE and Firefox. As with the microbenchmark results, we show dispatch rates for baseline, logging, and replay scenarios. To measure these rates, we manually identified all of the interrupt handlers in each game, adding a single line of code to each handler which incremented a global counter. At the end of 30 seconds, we divided this counter by the elapsed wall time to get the number of interrupts dispatched per second.

Figures 6(a) and 6(b) show that for applications with low to moderate interrupt rates, the interrupt dispatch rate was unchanged, i.e., Mugshot logging introduced no overhead. Compared to Tetris and Pacman, Spacius had a very high interrupt rate, executing about 100 callbacks per second. Figure 6(c) shows that for this application, Mugshot's logging overhead reduced dispatch rates by 0.8% on Firefox and 6.8% on IE. However, Spacius gameplay did not seem qualitatively degraded during logging on either browser.

Figure 6 shows that dispatch rates at replay time can decrease by as much as 75% in the case of a Spacius replay on Firefox. This time dilation is certainly tolerable, but as mentioned in Section 4.1, we could improve the replay rate by optimizing our log parsing.

4.2.3 Capturing Real Bugs

Mugshot's goal is to capture application runs and replay them on developer machines. An important application of replay mode is the recreation of buggy application states. Armed with the event interleavings that generate a program fault, a developer can use powerful localhost debuggers to step through the log and inspect the application's state after each event.

Since we lacked detailed changelogs for the seven applications described in Section 4.2, we could not intentionally undo a bug fix and then see whether Mugshot could successfully log and replay a problematic event sequence. However, while performing the experiments in Section 4.2, we did encounter bugs in two of the applications, both of which Mugshot could log and replay.

The first bug involved a display glitch in the Tetris program which we were able to characterize in detail using Mugshot. A Tetris game terminates if a falling block nestles amongst the static blocks in a way that causes the overall block structure to exceed a maximum allowable height. When this happens, the game should render the bottom part of the most recent piece but leave the top part clipped, since this part extends above the playable area. However, depending on the shape of the most recent piece and the preexisting block structure, the Tetris implementation we tested would incorrectly render the final block structure, scattering the constituent blocks of the final piece in arbitrary positions, sometimes overwriting preexisting blocks. Figure 3 shows an example of this bug. The final piece is 1 block by 4 blocks, but when its stacking causes the game to end, two of its blocks mysteriously materialize in the square formation at the bottom of the screen.

The second bug involved the Painter application. To draw a rectangle in this program, the user clicks on the "Rectangle Tool", then drags the pointer across the canvas with the mouse button down. If the user selects the "Rectangle Tool" and just single-clicks on the drawing area, no rectangle should be drawn. However, after single clicking, an expanding rectangle will appear as the user moves the mouse. This makes the user think that he is, in fact, drawing a rectangle. However, when the mouse is clicked again, the rectangle suddenly disappears.

We captured, replayed, and diagnosed both bugs using Mugshot. For both applications, the compressed log which captured the bug was under 11 KB in size. Transmitting such an error report to developers would be extremely fast, even on a slow connection.

5 Privacy

Mugshot provides developers with an extremely detailed log of user behavior. Some might worry that this leads to an unacceptable violation of user privacy. Such privacy concerns are valid. However, web sites can (and should) provide an "opt-in" policy for Mugshot logging, similar to how Windows users must willingly decide to send performance data to Microsoft [14].

We also emphasize that Mugshot is not a fundamentally new threat to online privacy. Web developers already have the ability to snoop on users to the extent allowed by JavaScript, and to send the resulting data back to their own web servers. Indeed, many web sites already perform a crude version of event logging using web analysis services like CrazyEgg [9] that build heat maps of click activity on a particular page. In all cases, the scope of JavaScript-based snooping is limited by the browser's cross-site scripting policies. From the browser's perspective, Mugshot is not an exception: it is subject to exactly the same restrictions designed to thwart malware. These restrictions prevent all programs—including Mugshot—from snooping on a frame owned by one domain and sending that data to a different domain.

6 Conclusions

As web applications have grown in popularity, browsers have shipped with increasingly powerful JavaScript debuggers. These tools are extremely useful for introspecting applications that are running on a local

development machine. However, they cannot be used to examine program contexts which reside on remote machines. When regular end users encounter application bugs, they will not inspect the application using their browser's advanced debugger. At best, they will send a bug report which describes their problem using natural language. At worst, they will do nothing and simply be frustrated. Ideally, users would have a convenient way to give developers the precise event sequence that led to a buggy application state. The developer could then recreate the execution run and use his knowledge of the code to diagnose the problem.

To address these issues, we created Mugshot, a lightweight framework for capturing JavaScript application runs and replaying them on different machines. Experiments show that Mugshot introduces little overhead at logging time. For applications like games which generate many events, Mugshot slows execution speeds by 6.8% in the worst case. Mugshot event logs grow at a reasonable rate, requiring 20–80 KB per minute of application activity. Using Mugshot's replay mode, we have successfully recreated bugs in two real applications. Mugshot's logs also support usability investigations and traditional click analytics.

References

[1] APPLE COMPUTER. Crash Reporting for iPhone OS Applications, 2009. Technical Note TN2151.

[2] BELL, J. Applesoft BASIC Interpreter in Javascript. http://www.calormen.com/applesoft/.

[3] BOUTEILLER, A., BOSILCA, G., AND DONGARRA, J. Retrospect: Deterministic Replay of MPI Applications for Interactive Distributed Debugging. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (2007), vol. 4757 of *Lecture Notes in Computer Science*, Springer, pp. 297–306.

[4] BREAKPAD. http://kb.mozillazine.org/Breakpad. MozillaZine Knowledge Base.

[5] CHOI, J.-D., AND SRINIVASAN, H. Deterministic replay of Java multithreaded applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools* (1998), pp. 48–59.

[6] CIESLAK, K. PacMan! http://www.digitalinsane.com/api/yahoo/pacman/.

[7] CLICKTALE LTD. ClickTale: Record, Watch, Understand. http://www.clicktale.com, 2009.

[8] CORNELIS, F., GEORGES, A., CHRISTIAENS, M., RONSSE, M., GHESQUIERE, T., AND BOSSCHERE, K. D. A taxonomy of execution replay systems. In *Proceedings of International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet* (2003).

[9] CRAZY EGG, INC. CrazyEgg: Visualize Your Visitors. http://crazyegg.com/, 2010.

[10] DIONNE, C., FEELEY, M., AND DESBIENS, J. A Taxonomy of Distributed Debuggers Based on Execution Replay. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications* (1996), pp. 203–214.

[11] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of OSDI* (2002), pp. 211–224.

[12] FORESEE RESULTS, INC. CS SessionReplay. http://www.4cresults.com/CSSessionReplay.html, 2009.

[13] GEELS, D., ALTEKAR, G., SHENKER, S., AND STOICA, I. Replay debugging for distributed applications. In *Proceedings of USENIX Technical* (2006), pp. 289–300.

[14] GLERUM, K., KINSHUMANN, K., GREENBERG, S., AUL, G., ORGOVAN, V., NICHOLS, G., GRANT, D., LOIHLE, G., AND HUNT, G. Debugging in the (Very) Large: Ten Years of Implementation and Experience. In *Proceedings of SOSP* (2009), pp. 103–116.

[15] GOOGLE. V8 JavaScript benchmarks. http://code.google.com/apis/v8/benchmarks.html.

[16] HACKETT, M., AND MORSE, J. Spacius. http://scriptnode.com/lab/spacius/.

[17] JON HEWITT. Ajax Debugging with Firebug. http://http://www.ddj.com/architect/196802787, January 10, 2007.

[18] KICIMAN, E., AND LIVSHITS, B. AjaxScope: A Platform for Remotely Monitoring the Client-side Behavior of Web 2.0 Applications. In *Proceedings of SOSP* (2007), ACM, pp. 17–30.

[19] KIRCHOFF, B. NicEdit. http://nicedit.com/.

[20] K.VIKRAM, PRATEEK, A., AND LIVSHITS, B. RIPLEY: Automatically Securing Web 2.0 Applications Through Replicated Execution. In *Proceedings of CCS* (2009), pp. 173–186.

[21] LAWRENCE, E. Fiddler: Web Debugging Proxy. http://www.fiddler2.com/fiddler2, 2009. Microsoft Corporation.

[22] NARAYANASAMY, S., POKAM, G., AND CALDER, B. Bugnet: Recording application-level execution for deterministic replay debugging. *IEEE Micro 26*, 1 (2006), 100–109.

[23] PARK, S., AND MILLER, K. Random Number Generators: Good Ones are Hard to Find. *Communications of the ACM 31* (1988), 1192–1201.

[24] ROBAYNA, R. Canvas Painter. http://caimansys.com/painter/.

[25] RUDERMAN, J., MIELCZAREK, T., LEE, E., AND RONN-JENSEN, J. JavaScript Shell. http://www.squarefree.com/shell/.

[26] SEIDELIN, J. DOMTRIS. http://www.nihilogic.dk/labs/tetris/.

[27] SELENIUM WEB APPLICATION TESTING SYSTEM. http://seleniumhq.org/, 2009.

[28] WOOD, L., NICOL, G., BYRNE, S., CHAMPION, M., HORS, A. L., HÉGARET, P. L., AND ROBIE, J. Document object model (DOM) level 2 core specification, Nov. 2000. http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113.

[29] XU, M., BODÍK, R., AND HILL, M. D. A "Flight Data Recorder" for Enabling Full-System Multiprocessor Deterministic Replay. In *Proceedings of ISCA* (2003), pp. 122–133.

AccuRate: Constellation Based Rate Estimation in Wireless Networks

Souvik Sen
Duke University

Naveen Santhapuri
Duke University

Romit Roy Choudhury
Duke University

Srihari Nelakuditi
University of South Carolina

Abstract

This paper proposes to exploit physical layer information towards improved rate selection in wireless networks. While existing schemes pick good transmission rates, this paper takes a step further towards computing the *optimal* bit rate. The main idea is to capture the channel behavior through symbol level dispersions, and "replay" these dispersions on different rate encodings of the same packet. The "replay" action can be emulated at the receiver without requiring the transmitter to send the packet at every other rate. The maximum successful rate is likely to be the optimal rate of the received packet, and assuming that the channel remains coherent, the same rate can be prescribed for the next transmission. We design, implement, and evaluate this idea over a small testbed of USRP hardware and GNURadio software. Our proposal, called *AccuRate*, predicts a packet's optimal rate 95% of times when the packet is received correctly. When the packet is received in error, AccuRate computes its optimal rate with 93% accuracy. In terms of throughput, we show that AccuRate improves over the state-of-the-art scheme SoftRate by around 10%, and is reasonably close to the optimal.

1 Introduction

Rate estimation is an important problem because it directly translates to throughput. The difficulty in rate estimation stems from channel fluctuations – the optimal rate quickly becomes stale, requiring a fresh round of estimation [1–3]. WiFi rate control is performed at the link layer, and hence, must operate on the granularity of packets. Approaches such as ARF [4], RRAA [5], and SampleRate [2] continuously track the success/failure of packets, and employ statistical prediction methods to select the appropriate rate. To improve responsiveness to channel fluctuations, alternate schemes have explored the use of SNR for rate selection. RBAR [6] and OAR [7] were the first-generation schemes that utilized RTS/CTS to exchange SNR values. However, with recent consensus to turn off RTS/CTS, new schemes are recording historical SNRs and deriving a rate-versus-SNR relationship from it [8, 9]. While this improves performance, continuously refreshing the SNR for every rate is often difficult [9]. Moreover, the rate-vs-SNR relationship changes with different propagation environments, especially when the channel changes quickly over time [1]. Therefore, although practical SNR-based schemes are reasonably good at slower time-scales, they lack the agility to achieve per packet rate adaptaion in dynamic wireless environments.

This paper proposes to exploit physical layer information (such as symbol level dispersion on a constellation space) to improve the accuracy of rate selection. We show that such PHY layer information can be derived from a received packet, and then used to compute the optimal rate at which that packet should have been transmitted. Although the optimal rate is computed in retrospect, it can be valuable for guiding the transmission rate of subsequent packets. Moreover, symbol level information can discriminate between losses due to fading and interference, further assisting in link layer retransmission strategies. Our ideas are consolidated into a constellation based rate estimation scheme, called *AccuRate*. We show that the improvements from AccuRate are consistent over diverse wireless environments.

AccuRate's main idea is intuitive. Given that the PHY layer encodes a sequence of bits into a symbol on the constellation space, AccuRate looks at the *dispersion* between the transmitted and received symbol positions. Small *dispersions* indicate that the communication link is strong, and perhaps capable of supporting higher rates than the one used. By comparing these *dispersions* to the permissible dispersions at different bit rates, AccuRate can precisely derive the maximum rate the packet could have been transmitted at. Even when the packet fails, AccuRate extracts known parts of the packet (preamble and postamble [10]), and estimates the appropriate rate from them. Of course, this is a retrospective analysis of

a just-concluded transmission. However, as argued earlier, knowing the optimal rate of a received packet is a valuable primitive for rate control algorithms. The AccuRate receiver prescribes this rate to the transmitter, which in turn uses it for the next transmission. So long as the channel remains coherent between two consecutive packets, AccuRate achieves a near-optimal rate selection accuracy.

This paper is not the first to use PHY layer information towards rate estimation. Recently, authors in [1] proposed SoftRate, a scheme that uses PHY layer confidence values to estimate a packet's bit error rate (BER). By comparing the BER against an empirically generated lookup table, the transmitter picks a "good" bit rate for subsequent transmissions. While SoftRate makes a valuable contribution, we show that there is room for improvement. Specifically, we show that by directly operating on symbol constellations, AccuRate can "jump" to the optimal rate in one step, while eliminating the reliance on empirical measurements. Moreover, AccuRate's approach scales to arbitrarily high bit rates, and does not require large gaps between the consecutive rates. Experiments performed in a wireless channel simulator [11] (where the channel conditions can be repeated for fair comparison) demonstrates that AccuRate reliably selects the optimal rate. Similar experiments on a prototype USRP testbed show consistent throughput gains under various wireless environments. Together, these results confirm that AccuRate advances the state of the art through PHY-aware rate estimation. AccuRate's key contributions can be summarized as follows.

- **Identify the opportunity of rate estimation using symbol dispersion at the PHY layer.** We verify our ideas through measurements on the USRP/GNURadio platform. The findings offer new insights for further research at the link layer.

- **Develop a constellation based rate estimation scheme (AccuRate) that "jumps" to the appropriate rate.** The wireless channel manifests itself through symbol level dispersions. By "replaying" the dispersions on packets at different rates, AccuRate is able to identify the best bit rate of a packet. This bit rate is prescribed for future transmissions.

- **Implement and evaluate AccuRate on a USRP testbed, and on a emulation platform composed of USRPs and a wireless channel simulator.** Results from 25 hours of testbed experimentation shows consistent improvement in performance over existing schemes. Emulation results (enabling experiments under controllable and repeatable channel conditions) exhibit similar trends.

2 Related Work

Perfect bit rate selection in wireless networks is an eluding problem that has been researched extensively in the past [1, 2, 4–9, 12–18]. Existing schemes have been broadly classified as frame-level or SNR-based, and has been well surveyed in [1]. Here, we touch upon only the recent works relevant to AccuRate.

History based: SampleRate [2] by Bicket adapts transmission rate by periodically probing the channel with packets at various bit rates. The idea is to adapt to changing channel conditions and minimize the overall transmission time for the packets. In RRAA [5] the authors propose faster rate estimation than SampleRate by using loss information from short frame windows. Frame error history based schemes like SampleRate and RRAA do not distinguish between fading and collision which is significant for rate estimation. This class of schemes are also slow to converge, and may not converge at all, if channel conditions change frequently. AccuRate distinguishes between fading and collisions and has a one-packet convergence-time to estimate the best rate supported by the channel.

SNR-based: Two recent SNR-based schemes take a cross layer approach to perform rate estimation. In [9], Camp and Knightly show that SNR-BER relationships change with the operating environment and therefore need training to operate in a particular environment. They also compare existing SNR-based schemes with SNR-trained schemes to show that trained SNR schemes perform considerably better. In [17], the authors demonstrate the utility of adaptive modulation per frequency band. The variation of channel characteristics across frequency sub-bands accentuates the effect in ultra wide band regimes which will benefit the most from such schemes. To perform well these schemes need in-situ training for each environment. AccuRate does not need any training or information about the environment.

Collision vs. Fading: Collision detection has been an area of active research and lately several schemes have been proposed [19–23]. The scheme in COLLIE [20] allows a transmitter to distinguish between a fading and a collision loss by having the receiver send back the erroneously received packet. This allows the sender to identify the corrupt bits (via comparison with the original packet), and then analyze the cause of failure by analyzing the corruption patterns. Of course, the scheme depends on proper packet reception from the receiver in a timely manner. In [22], the authors propose a way to distinguish between collisions and fading, and adapt rate based only on the errors due to fading. This scheme is still history based and suffers from the same pathologies associated with other similar schemes. The use of OFDM symbol dispersions was shown in [23] as a

technique to distinguish between collision and fading. Our work goes beyond making this distinction by using known dispersions to select the correct rate.

SoftRate: The closest proposal to AccuRate is SoftRate [1], which was the first to exploit PHY layer information for rate estimation. We therefore focus on explaining the differences between SoftRate and AccuRate. SoftRate achieves high quality rate estimation using a cross layer approach, but we believe there is room for improvement. Specifically, SoftRate estimates the rate supported by the channel based on the BER of the received packet. The BER is an average of SoftPHY confidence values, computed from the dispersion of the received symbols from their nearest constellation symbols. SoftRate employs a heuristic to predict the BER at other bit-rates using the BER estimate at a given bit rate[1]. While this heuristic can effectively indicate when the rate must decrease to the next-lower bit rate (or increase to the next-higher bit rate), the ability to jump directly to the best rate is limited. In contrast, AccuRate's ability to replay the channel distortion on all possible rates facilitates selection of the best rate in one step. The replaying mechanism is expected to scale to bit rates at potentially finer granularity. However, unlike SoftRate, the hardware cost and implementation complexity may be excessive. To balance performance and complexity, one may envision a combination of AccuRate and SoftRate – a topic of future research.

3 Background and Observations

We present some background material on PHY layer encoding/decoding of bits with different modulation schemes. Building on this understanding, we observe that the extent of signal distortion due to channel fading is independent of the modulation scheme. We validate this through USRP/GNURadio measurements, and use it as a pivot for subsequently proposed ideas.

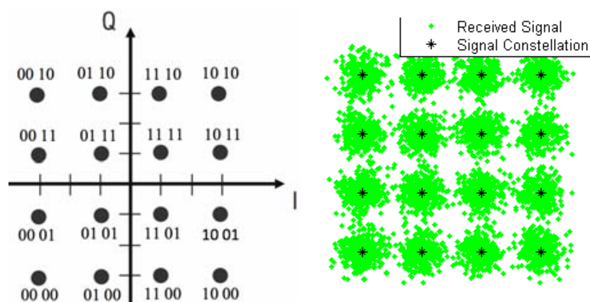


Figure 1: Symbol constellation for 16QAM: (a) Each symbol corresponds to a 4-bit sequence. (b) Symbols received after suffering channel-induced dispersions.

[1]The heuristic exploits the empirical observation that, under a given SNR, adjacent bit rates experience a factor of 10 difference in BER.

3.1 PHY Layer Symbol Constellations

The PHY layer encodes a sequence of bits into a PHY symbol which is represented by a position on a 2D complex plane called the *constellation* diagram. Figure 1(a) shows an ideal constellation diagram from 16-ary quadrature amplitude modulation (16QAM). If the transmitter wishes to send a bit sequence "0000", it sets the *In-Phase* (x-axis) and *Quadrature* (y-axis) to a value of $<-3,-3>$. The receiver recovers the I and Q values after demodulation, and plots each symbol on the IQ plane. Since the channel distorts the transmitted signals, the received symbol positions get dispersed from their ideal positions. Let $\vec{r_i}$ be the received symbol position and $\vec{s_i}$ be its ideal symbol position. We define its dispersion as $\vec{d_i} = \frac{\vec{r_i}-\vec{s_i}}{\vec{s_i}}$. This is essentially the Error Vector Magnitude (EVM) [24, 25], but for ease of understanding, we refer to it as dispersion. Figure 1(b) shows an example of the dispersed symbols at the receiver.

To decode the symbols, for each received symbol position, $\vec{r_i}$, the receiver guesses the corresponding ideal symbol position, $\vec{s_i}$. A simple method is to pick the symbol that is closest to the received symbol position $\vec{r_i}$. In other words, there is a *tile* associated with each symbol in the constellation. When a symbol $\vec{s_i}$ is received correctly, its received position falls within the symbol's tile, i.e., $\vec{r_i} \in \texttt{tile}(\vec{s_i})$. When all the symbols in a packet are received correctly, the corresponding bits will pass the CRC check and the packet is handed to the upper layer.

With channel fading or interference from nearby transmissions, the received symbol position $\vec{r_i}$ may be quite far away from the transmitted symbol position $\vec{s_i}$. The received position $\vec{r_i}$ may even fall outside the tile of $\vec{s_i}$, i.e., $\vec{r_i} \notin \texttt{tile}(\vec{s_i})$. Then, $\vec{r_i}$ will be closer to another symbol position than $\vec{s_i}$, misguiding the receiver to believe that some other symbol was transmitted instead of $\vec{s_i}$. This error will be caught later when the CRC check on the packet fails. Of course, channel coding techniques, such as forward error correction (FEC), may be effective in correcting some errors in demodulation. If the number of errors are large, even channel coding may not be adequate to recover the packet.

3.2 Relation between Transmission Rate and Symbol Constellation Density

For ease of explanation, let us ignore channel coding for now and assume that each symbol encodes the actual bits from the packet. Observe that a higher transmission rate is realized by encoding a longer bit-sequence on a symbol. Thus, if one increases the length of the sequence from 2 to 4 bits per-symbol, the constellation diagram must also accommodate a greater number of symbols (from 4QAM to 16QAM). In other words, the den-

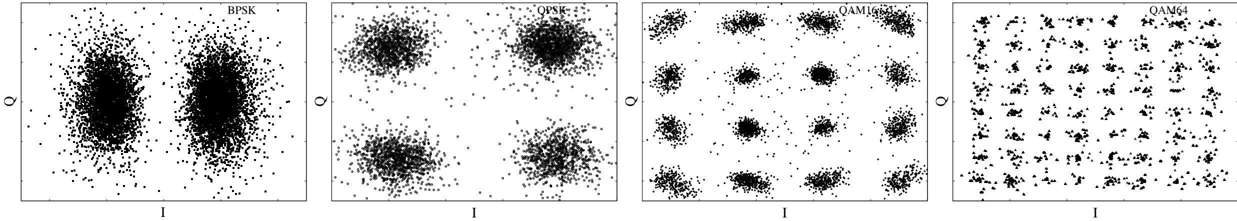


Figure 2: Symbol density increases with increasing data rates (BPSK, QPSK (or 4QAM), 16QAM, 64QAM).

sity of symbols in the constellation diagram increases at higher rates as shown in Fig. 2. Since increased density implies shorter distance between neighboring symbols, the received packet is more susceptible to errors at higher rates when the channel is weak. Figure 3 confirms this by showing that the maximum tolerable BPSK error ($|s_i - r_i|$) can be twice that of QPSK (or 4QAM), and four times that of 16QAM. This well-known observation will underlie the design of AccuRate.

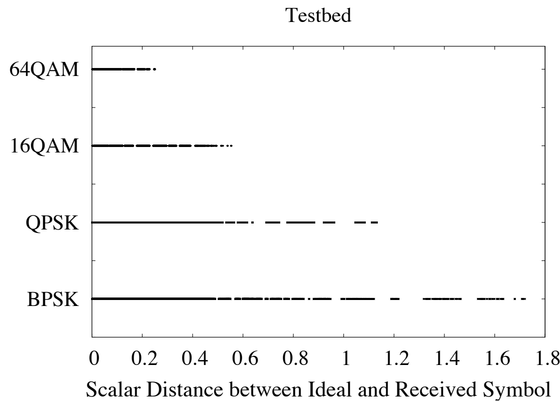


Figure 3: Lower rates can tolerate higher magnitude of symbol dispersion

3.3 Relation between Dispersion due to Channel Fading and Bit Rate

We now demonstrate that symbol dispersion is not influenced by the modulation scheme (or transmission rate), and is only a function of the channel. We transmit data from a static USRP sender to a static USRP receiver using 2, 4, 16, and 64 QAM. We maintain as much coherence in the channel as possible (by keeping the physical environment static), and transmit small packets repeatedly using different modulation schemes in a round robin manner. For every received symbol, we calculate its dispersion from the correct constellation symbol[2]. Figure 4 plots the CDF of symbol dispersion magnitude for each modulation scheme for packets transmitted in one round.

[2]The correct constellation symbol is known because the transmitted packet is known in our experiments. Thus, even when a packet fails, we can still compute the correct dispersions.

Almost-identical curves provide evidence that *the dispersions are independent of the symbol constellation, and therefore the transmission rate.* More detailed experimental evidence is presented in [25].

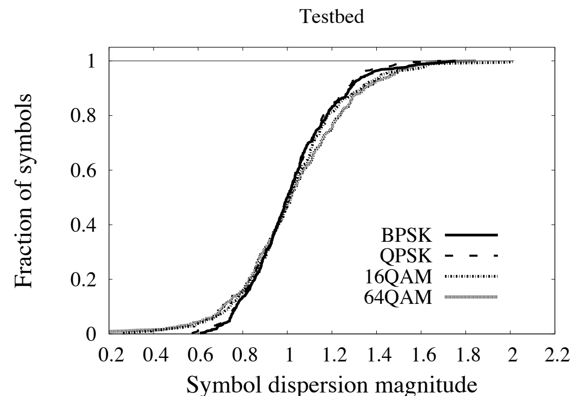


Figure 4: CDF of symbol dispersion magnitude for packets transmitted with different modulation schemes. Not all packets were received correctly, but their dispersions could be computed offline using the (known) transmitted packet.

These observations enable us to model the channel behavior based on the dispersion of known symbols at the receiver. The receiver can then conduct a *what-if* analysis by "replaying" the channel on a packet encoded at different rates. For instance, Fig. 5(a) shows the dispersion of symbols when a packet was transmitted using 4QAM. Given that the dispersion is independent of the modulation, the receiver can check whether a higher modulation such as 16QAM with denser constellation could have tolerated the same level of dispersion. In other words, 16QAM is feasible if all the received symbols in each 4QAM quadrant can be accommodated in a smaller 16QAM tile (drawn with dashed grids) as in Fig. 5(b). The original 16QAM grid, as shown in Fig. 7 has been shifted and superimposed on Fig. 5(b) for the purpose of demonstration. In this example, 16QAM is not feasible since some received symbols spill out of their correct tile. More generally, this shows that the outcome of a 16QAM transmission may be predicted without actually transmitting the packet over the air. Repeating this over all possible bit rates will reveal the best possible rate for

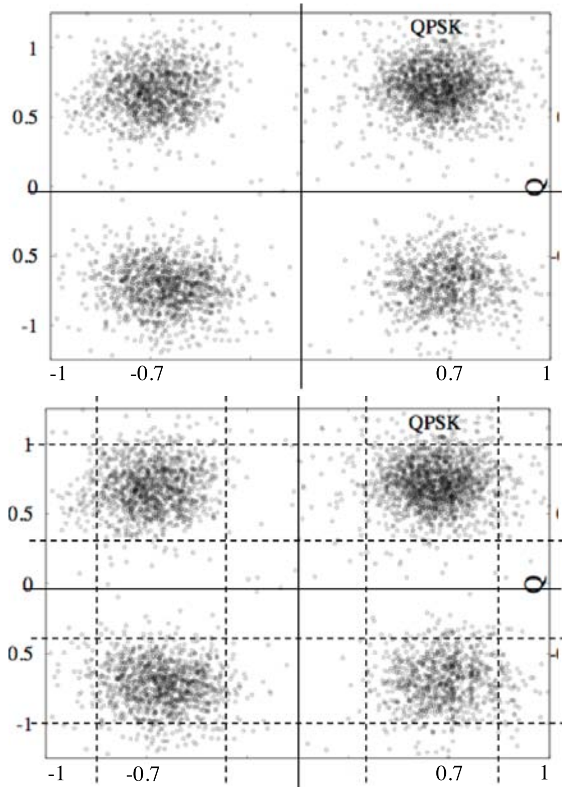


Figure 5: Receiver checks if QPSK (4QAM) packet could have sustained higher bit rate (16QAM).

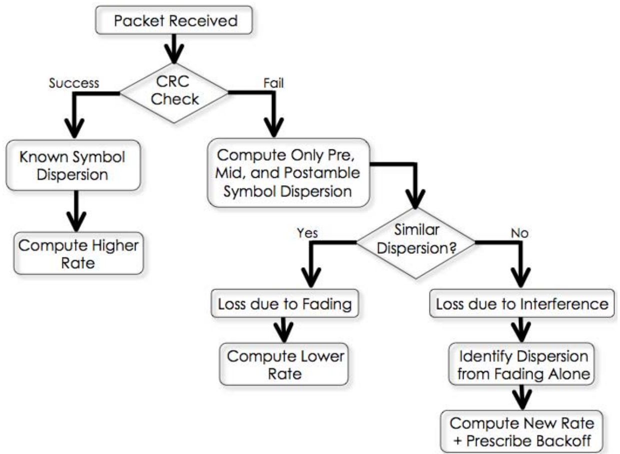


Figure 6: Flowchart of determining the optimal rate in retrospect.

this just-received packet[3]. Such a retrospective analysis can guide us in subsequent rate control decisions. The details on how this hindsight is leveraged is described in the following sections.

4 Determining The Optimal Transmission Rate in Retrospect

We now explain how a receiver can determine from a received packet, what could have been the optimal rate, for transmitting that packet. A high level schematic of the procedure is depicted in Fig. 6. We present the rate computation method for three cases: (1) when the packet is received successfully, (2) when the packet fails due to fading, and (3) when the packet fails due to interference. We support our basic claims with measurements from the USRP testbed.

4.1 In Case of Successful Packet Reception

Let us first consider the case where a packet is successfully received. Since all the bits are decoded correctly, the receiver is aware of all the transmitted symbols. Hence, it can compute the dispersion $\vec{d_i}$, between each transmitted symbol position $\vec{s_i}$ and received symbol position $\vec{r_i}$. Assuming N symbols in the packet, the channel can then be characterized by $\vec{D}$, a sequence of dispersions, i.e., $\vec{D} = \{\vec{d_1}, \vec{d_1}, \cdots \vec{d_N}\}$. Now, suppose the packet was transmitted at a bit rate of R. Given that it was received successfully, it is clear that the symbol-constellation density corresponding to R can tolerate the dispersion $\vec{D}$. Now the question is what is the highest rate, R^* ($\geq R$), at which the transmission would have been successful over a channel with dispersion sequence $\vec{D}$.

Note that, as argued before, $\vec{D}$ is independent of the modulation used by the transmitter, i.e., the i'th symbol gets dispersed by $\vec{d_i}$ regardless of whether that symbol is from the constellation of BPSK, QPSK, 16QAM, or 64QAM. Consequently, *the receiver can analyze the outcome of different modulations without requiring the transmitter to explicitly send the packet once per each modulation*. The procedure to check whether a transmission at a higher modulation would be successful is as follows. For each symbol i, we apply the dispersion vector $\vec{d_i}$ on its ideal position $\vec{s_i}$ in the constellation space and check if the resulting symbol position would still be correctly decoded. If that position happens to be closer to some other constellation point (i.e., in some other tile), this constellation is too dense for this dispersion. In this manner, the most-dense constellation is chosen in which, for each symbol i, $\vec{d_i}$ is completely contained in the same tile. Figure 7 illustrates this checking operation – a symbol received through BPSK modulation is being tested against a 4QAM and 16QAM constellation. In this example, the channel-induced dispersion can be tolerated by a 4QAM symbol, whereas a 16QAM symbol will not be decoded correctly as it's received position falls in the wrong tile. Ignoring error coding, this implies that 16QAM is an inappropriate rate for transmitting this

[3]The what-if analysis with "replay" operation is applicable even with channel coding as discussed later in Section 4.1.1

packet. However, 4QAM may prove to be suitable, provided all the symbols in the packet passes this test successfully.

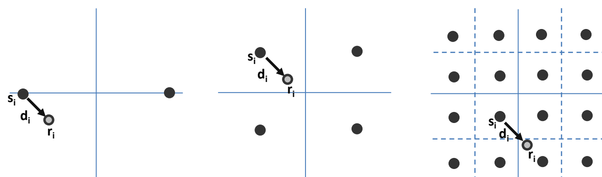


Figure 7: Computing the appropriate rate at which this packet reception would have been successful. In this case BPSK and QPSK will be successful where as 16QAM will not.

4.1.1 Error Correction with Channel Coding

We now introduce the role of channel coding in rate computation. Briefly, channel coding helps in error correction by including redundant bits in the packet. Some symbols may fall in the incorrect tile on the constellation, but channel coding may still be able to correct them. This implies that coding can allow for a denser constellation, at the expense of a larger packet size. The net result is a new intermediate rate between the sparse and the dense constellation. To clarify with an example from 802.11g, 4QAM results in 24Mbps. However, data rate 18Mbps can be achieved if the 4QAM modulation is combined with a 3/4 coding scheme. Thus, channel coding allows for additional data rates, offering finer-grained choices to the rate selection algorithm.

With coding, our goal then is to precisely identify the best <modulation, coding> tuple at which this packet would have been successful. For this, the receiver considers every higher modulation scheme, and computes the fraction of symbols that would have been in error. Note that the higher the modulation, the more the number of errors, and the larger the number of redundant bits. Suppose a modulation M_1 needs 3/4 coding to correct errors and a higher modulation M_2 requires more redundant 1/2 coding. Let R_1 and R_2 be the rates corresponding to M_1 and M_2 respectively. Then, the effective rates (after accounting for the overhead due to coding) would be $\frac{3}{4}R_1$ and $\frac{1}{2}R_2$. The higher effective rate is then chosen as the best rate in retrospect, i.e., $R^* = \max(\frac{3}{4}R_1, \frac{1}{2}R_2)$.

Besides offering bit rates at finer granularity, channel coding also allows for precisely computing the symbol dispersions. If a packet is known to pass the CRC check, the exact dispersion can be computed for all the symbols in that packet. These dispersions can then be recorded and replayed on higher-rate packet encodings to estimate the best bit rate. Observe that a packet may be successful even if some replayed dispersion causes the symbol to fall in an incorrect tile – channel coding may absorb these errors, similar to over-the-air reception. In other words, so long as the dispersions are precisely known, the replaying operation is no different from an actual transmit-receive operation. As will be clear from Section 5.2, even the same hardware chain may be reused for both the actual reception and the replayed operation. This implies that, as long as a packet is received correctly, AccuRate can retrospectively compute its optimal bit rate, and use it for the subsequent transmission.

4.2 In Case of Packet Loss due to Fading

Let us now consider the case where a packet is not received correctly, and the receiver has to find the smallest rate reduction that would have resulted in a successful reception. This is more challenging because, unlike the above case, the receiver does not know the actual transmitted symbol $\vec{s}_i$ for every received symbol position $\vec{r}_i$. Since the packet failed the CRC check, some $\vec{r}_i$ must have been outside the tile of the correct symbol $\vec{s}_i$ and inside the tile of some other symbol. The receiver does not know which of the symbols are incorrectly decoded and so it can not precisely compute for each symbol the dispersion $\vec{d}_i$ caused by the channel.

Fortunately, each packet starts with a preamble, a *globally known sequence* of bits, that the receiver uses to detect and synchronize onto a newly arriving signal. The receiver can utilize the preamble to estimate dispersion [26]. Suppose the preamble consists of k symbols and their computed dispersions are $\vec{d}_1^{\text{pre}}, \vec{d}_2^{\text{pre}}, \cdots, \vec{d}_k^{\text{pre}}$. We subject this sequence of k dispersions to the whole packet, i.e., we compute the dispersion for i'th symbol in the packet as $\vec{d}_i = \vec{d}_{i\%k}^{\text{pre}}$. Given this set of $\vec{d}_i$ vectors, we try to estimate the optimal rate R^* using the same approach as described earlier.

The preamble's symbol dispersions will only capture the channel behavior in the earlier parts of the packet. If the channel changes over time, the later changes will remain unquantified. Therefore, to better cope with channel variations, a postamble [10] may be inserted at the end of the packet. Suppose the postamble also consists of k symbols and their dispersions are $\vec{d}_1^{\text{post}}, \vec{d}_2^{\text{post}}, \cdots, \vec{d}_k^{\text{post}}$. Given the original packet size of N symbols, and a randomly generated packet of N or more symbols, we subject the ith symbol in the random packet to a dispersion $\vec{d}_i$, computed as follows: If $i < \frac{N}{2}$, then $\vec{d}_i = \vec{d}_{i\%k}^{\text{pre}}$, else $\vec{d}_i = \vec{d}_{i\%k}^{\text{post}}$. The rationale is that the preamble is a better representative of the first half of the original packet duration, while the postamble is better for the rest of the duration. Also, when the random packet is encoded at a lower rate, the number of symbols increase. The postamble is likely to be a better estimate of the channel for these symbols as well. Once again, based on $\vec{d}_i$ vectors obtained thus, we try to estimate the optimal rate of the

packet, R^*. The overhead of postamble would be justifiable if that leads to throughput improvements due to better rate estimation.

4.3 In Case of Packet Loss due to Interference and Fading

Rate selection must be approached somewhat differently when interference is the cause of packet failure. Under interference only, the transmitter should ideally backoff and transmit at the same rate. Under both interference and fading, the ideal approach is to backoff but transmit at a rate that accounts for the channel's fading component. We approach this problem by looking at both the preamble and the postamble. The presence of a preamble and postamble in a packet offers multiple "glimpses" into how the channel varied during packet reception. Because both preamble and postamble are known, the receiver computes their respective dispersion vector sequences $\vec{D}^{\text{pre}}$ and $\vec{D}^{\text{post}}$. Using statistical methods, we compute the similarity of these sequences (detailed in Section 6). If there was no interference, $\vec{D}^{\text{pre}}$ and $\vec{D}^{\text{post}}$ would be similar and the loss is attributed to fading. Otherwise, depending on whether the interference overlapped with the preamble or postamble, $\vec{D}^{\text{pre}}$ or $\vec{D}^{\text{post}}$ would exhibit a higher dispersion than the other. If rate must be selected only in response to channel fading, then we must select R^* based on $\min(\vec{D}^{\text{pre}}, \vec{D}^{\text{post}})$. Of course, when the interference overlaps with both the preamble and the postamble, $\vec{D}^{\text{pre}}$ and $\vec{D}^{\text{post}}$ will be similar, and our approach will incorrectly select a lower-than-optimal rate. Also, if the interfering packet is small enough to fit within the preamble and postamble of a transmission, AccuRate will fail to prescribe backoff although it will still estimate the rate induced by fading alone. One way to alleviate this problem is to insert known "midambles" in different parts of the packet thereby allowing for multiple glimpses into the channel behavior. We discuss these possibilities in section 7.

4.4 Experimental Validation

The above approach, AccuRate, raises a few obvious questions about its feasibility and performance. How accurately can a receiver determine the optimal rate in retrospect? Is the preamble sufficient or the postamble also necessary for estimating the rate in case of packet loss due to fading? To answer these questions, we conducted experiments on a Rayleigh fading channel simulator [11] and a real testbed. The evaluation setting is described in detail in Section 6. Briefly, in the simulator we froze the channel parameters for a Rayleigh fading model in GNURadio and computed the ideal rate $\hat{R}$ (by transmitting at all rates). Next, we allowed the receiver to determine the optimal rate R^* from the received packet under identical channel conditions. We repeated this experiment with different channel parameters and transmit powers. Overall, we compared $\hat{R}$ and R^* for more than 2000 packets. Fig. 8(a) shows the comparison by plotting the difference between R^* and $\hat{R}$ rate levels (e.g., successive 802.11 bit rates such as 24 Mbps and 18 Mbps are separated by 1 rate level). These results indicate that when the packets are received correctly, $R^* = \hat{R}$ for every instance. Even with preamble alone, AccuRate can determine the rate correctly in 80% of the cases, and the addition of postamble improves the accuracy to 95%. The postamble samples help AccuRate better estimate wireless channel coefficients using symbol dispersion. We observed similar results over a real wireless channel between a USRP/GNURadio transmitter and receiver pair – Fig. 8(b) shows these results.

The above description and supporting results offer reason to believe that symbol dispersion information gathered from a received packet can be used to estimate the packet's optimal rate. Considering that the channel coherence time is expected to be in the order of multiple packets, the receiver can prescribe the same rate for the subsequent transmission. One could argue that the optimal rate for the previous packet may not be optimal for the next packet or may even cause packet loss. But note that every rate adaptation scheme has to speculate the future channel conditions based on the past measurements. Any scheme that is not overly conservative and attempts to extract the best throughput from the channel runs the risk of packet loss. However, since our approach is based on fine grain information about the channel, the next packet has a reasonable chance of succeeding at the prescribed rate. In the following section, we gather our ideas into a single Constellation Based Rate estimation protocol, called AccuRate.

5 AccuRate Protocol and Implementation

5.1 Protocol

The AccuRate module is located at the boundary of the PHY and MAC layer. For every outgoing frame, AccuRate concatenates it with a postamble. Upon reception of this packet, the AccuRate receiver performs the following checks and reacts accordingly. If the packet is correctly received, AccuRate estimates the best transmission rate, and piggybacks it in the acknowledgment (ACK). If the packet is incorrectly received (meaning that the preamble was decoded but the CRC check failed), AccuRate triggers an interference-detection operation. Learning that the failure was not due to interference, AccuRate estimates the appropriate rate using only the pre/postambles [1], and conveys this back through a negative acknowledgment (NACK). However, if interference was the cause of failure, AccuRate performs rate estimation using either the interference-free preamble or postamble, depending on which exhibits lower symbol

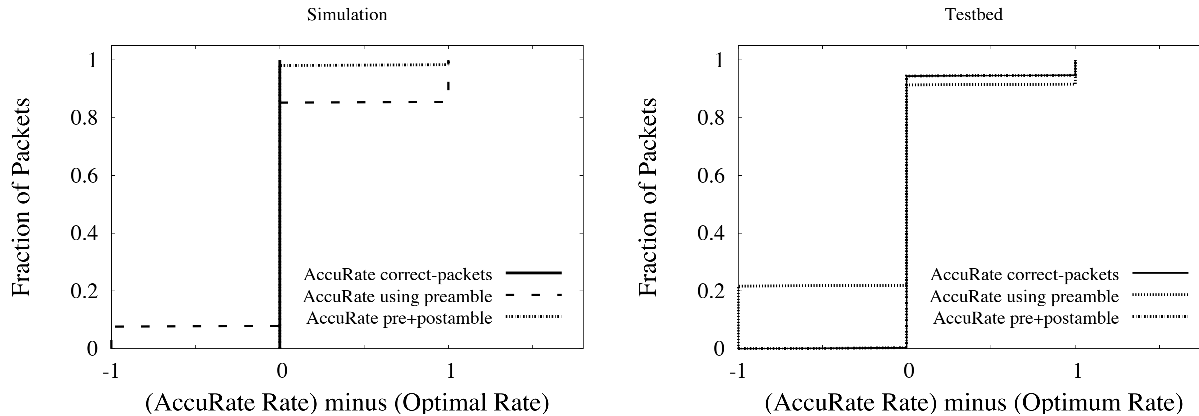


Figure 8: Accuracy of determining the optimal rate in retrospect: (a) simulation; (b) test bed. The x-axis is expressed in rate levels, where two successive 802.11 rates are assumed to have a rate level difference of one. The *AccuRate-correct-packets* and *AccuRate-pre+postamble* curves overlap. These results correspond to channel conditions with fading but without interference (impact of interference on AccuRate is evaluated in Section 6). The optimal rate in the testbed scenario is determined by sending a train of small packets at all possible rates. (details in Section 6.1)

.

dispersion. AccuRate conveys this fading-induced rate in the NACK, but also instructs the transmitter to back-off according to regular 802.11. In the worst case, if the packet's preamble itself is non-decodable, AccuRate cannot perform any rate prediction. The transmitter does not receive any ACK/NACK, and retransmits the packet as per the 802.11 specifications. In all other cases, the transmitter adopts AccuRate's rate and backoff prescriptions, and prepares accordingly for the next transmission to the same receiver.

5.2 Implementation

AccuRate builds on the OFDM codebase for the USRP/GNU-Radio platform. We adopt the publicly available building blocks of SoftRate (like the BCJR decoder [27]) for building AccuRate. This facilitates a platform for fair comparison between the two. 802.11a/g specified modulation schemes and channel coding rates are used (Table 1) in an attempt to emulate 802.11 like scenarios. The transmitter encodes the data using a standard rate-1/2 convolutional encoder, and applies puncturing to achieve varying code rates. The bandwidth is fixed at 20 MHz for GNURadio simulations and at 2 MHz for testbed experiments. We have incorporated a Rayleigh fading channel simulator [11] into the GNURadio codebase. The OFDM implementation uses an FFT length of 1024, with 394 occupied tones, 8 pilot tones and a cyclic prefix of length 256.

Figure 9 presents the block diagram for AccuRate's implementation in GNURadio. SoftRate is also shown as a comparison point, especially because the two schemes use very similar modules. In SoftRate, an incoming packet is demodulated and passed through the BCJR decoder. The output of the BCJR decoder comprises the data bits and their respective confidence values. These are passed through a BER computation module, resulting in the actual packet and its single BER. The SoftRate estimation algorithm runs in the "Select Rate" module, which picks the packet's rate by comparing the BER against a BER-Rate relationship curve. The final output, $R^*_{SoftRate}$, is SoftRate's prescribed rate.

We note that AccuRate uses a similar module chain with a few augmentations. When the over-the-air packet arrives, AccuRate measures the symbol level dispersions from the demodulator and stores it in the *Build Dispersion Model* module. This module uses the correctly received packet to calculate the accurate per-symbol dispersion[4]. In addition, a random packet is generated and encoded at different rates $(R_1, R_2, ...R_n)$. Symbols from each rate-encoded packet are then subjected to the recorded dispersions, and the output is passed through the demodulator. Although Figure 9 shows the operations in parallel (incurring an additional hardware cost), we use a single chain in our GNURadio implementation and iterate over all possible rate encodings (imposing a higher processing latency). The output of the demodulator, denoted *Demod*, is fed into the BCJR decoder. The output bits are collected into a frame and checked for CRC (the confidence values are not used in AccuRate). If the CRC check passes at that rate, AccuRate deems the corresponding rate to be successful. AccuRate picks the maximum of all successful rates, and prescribes it for the subsequent transmission.

[4] If the over-the-air packet failed, the dispersion sequence is suitably built from the preamble and postamble only.

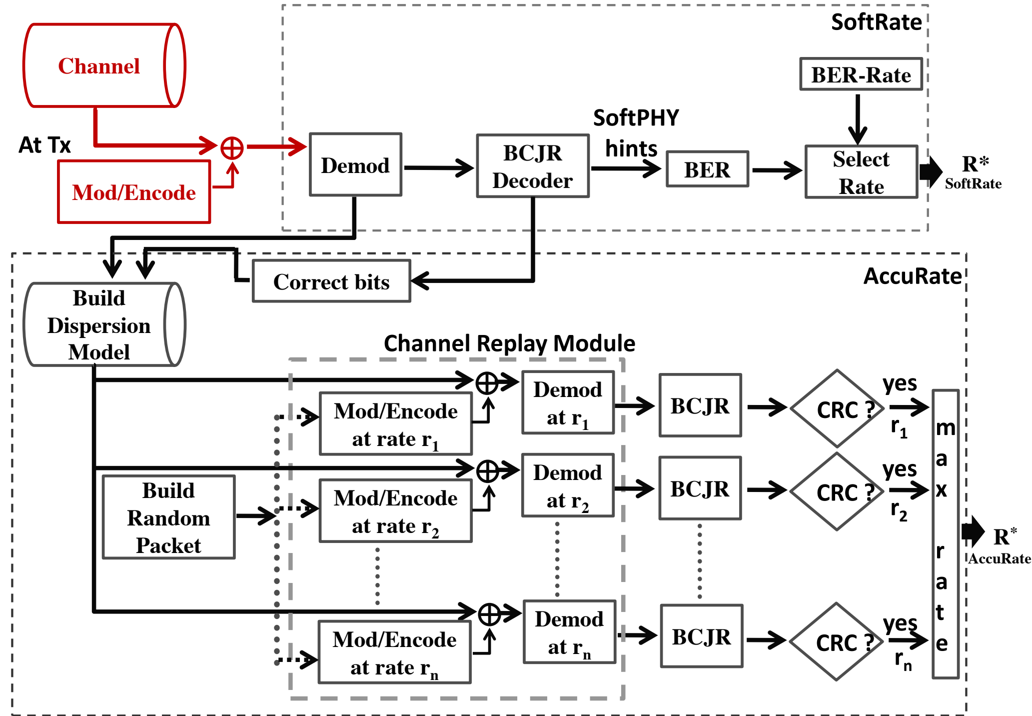


Figure 9: Block diagram of AccuRate

Table 1: 802.11 Modulation and coding used in AccuRate (20Mhz channel)

Modulation	Coding	802.11 rate (implemented?)
BPSK	1/2	6 Mbps (yes)
BPSK	3/4	9 Mbps (yes)
QPSK	1/2	12 Mbps (yes)
QPSK	3/4	18 Mbps (yes)
16QAM	1/2	24 Mbps (yes)
16QAM	3/4	36 Mbps (yes)
64QAM	2/3	48 Mbps (no)
64QAM	3/4	54 Mbps (yes)

6 Evaluation

6.1 Methodology

We faced two challenges while evaluating AccuRate. (1) The wireless channel changes over time making it difficult to determine what could have been the optimal rate for a given transmission. (2) The high latency incurred in procuring RF samples from the USRP front-end makes it impractical to evaluate AccuRate in realtime. In view of these, we make two approximations in our experimentation methodology. First, we incorporate a Rayleigh fading simulator into the GNURadio codebase. The simulator [11] employs the same USRP/GNURadio transmitter and receiver, only connects them through a loopback configuration. Packets flow out of the transmitter, and instead of advancing through the wireless channel, they are made to flow through simulated channel conditions. The output of the channel simulator is presented to the receiver which then executes regular demodulation/decoding. Since the simulated channel conditions can be forced to remain unchanged, we are able to compare the optimal bit rate for a given transmission against those prescribed by AccuRate and other schemes.

Our second approximation is designed to test the performance of AccuRate over real wireless channels. To this end, we repeatedly transmitted trains of packets, each train comprising of 7 short packets (each 200 bytes) at increasing bit rates. Assuming that the channel is coherent for the duration of the packet train, we determine the optimal transmission rate R^* by recording the highest bit rate successful in that train. Now, AccuRate picks a random packet in the first train, predicts the optimal rate for that packet $R^*_{AccuRate}$. The operation is performed offline – the difference between R^* and $R^*_{AccuRate}$ characterizes AccuRate's rate selection accuracy. Moreover, the packet corresponding to $R^*_{AccuRate}$ in the next train is also selected, as if that's the transmission that AccuRate would have executed. This packet's transmission at $R^*_{AccuRate}$ is then used to predict the subsequent transmission rate, and so on. The throughput is computed based on the success/failure of the packets selected in each train. SoftRate's performance is also compared in

these settings. Thus, while simulators provide faithful comparisons under approximate channel models, packet-train based evaluations attempt to achieve the converse. We believe that together, these experiments provide a fair comparison between AccuRate, SoftRate, and the Optimal rate selection algorithms.

6.2 Performance Results

We have designed experiments to answer the following key questions about the performance of AccuRate. (1) What is AccuRate's rate estimation accuracy compared to the optimal rate and other existing schemes? (2) How does the accuracy vary under different channel conditions? (3) How well does AccuRate discriminate between fading and interference? How does interference affect rate selection? (4) What is the accuracy of rate estimation based on preamble and postamble dispersions?

To understand AccuRate's performance against existing schemes, we also evaluate SoftRate and SNR-based rate estimation. SNR-based rate uses the SNR feedback to pick the transmission bit rate. The SNR-rate relationship is derived a priori from a wide range of empirical measurements on USRP/GNURadios.

Rate Selection Accuracy

We evaluated the accuracy of rate estimation by AccuRate and other schemes in both slow fading (walking) and fast fading (driving) scenarios as described below.

Slow Fading: We induced slow fading by moving the USRP transmitter on a wheeled chair, while it is transmitting to a fixed USRP receiver. We transmitted 500 packet trains where each train has one packet per rate. We repeat this experiment 10 different times and thus resulting in a total of 5000 packet transmissions per each bit rate. Figure 10(a) shows the results of these testbed experiments by plotting the CDF of the estimation accuracy. A negative value of the difference between estimated rate and optimal rate indicates underselection and a positive value indicates overselection. AccuRate selects the optimal rate nearly 95% of the time, which is around 10% and 20% better than SoftRate and SNR-based scheme respectively. We also conducted simulations by setting the channel parameters to reflect slow fading. In this case, as shown in Figure 10(b), AccuRate is always optimal and again performs better than the other two schemes. Based on these results, we conclude that AccuRate estimates the rate with high accuracy under slow fading.

To get a sense of how well the predicted rate by AccuRate tracks the optimal rate in a time varying channel, we take a closer look at AccuRate rate selection. We plot the AccuRate rate and optimal rate at each point for a 300 train snapshot in Figure 11. Clearly, AccuRate

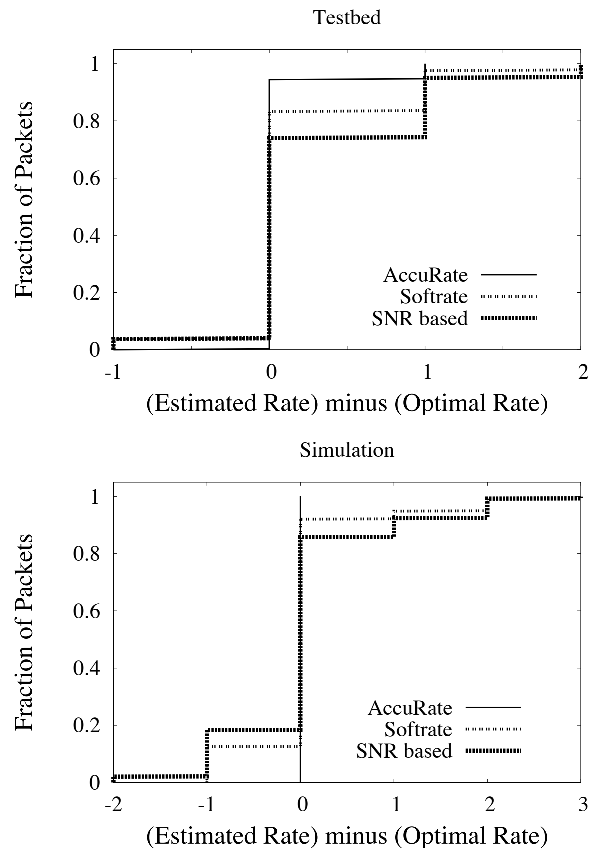


Figure 10: Rate selection accuracy under slow-fading mobility: a) Testbed vs (b) Simulation results. The X-axis shows the difference in discrete rate levels.

tracks the optimal rate curve reasonably well.

Fast Fading: Doppler effects at vehicular speeds cause fast fading in wireless channels. We examine AccuRate's performance by simulating such conditions in the Raleigh Fading channel simulator for GNURadio [11]. This simulator implements detailed channel models including multipath. The inputs (and outputs) to this simulator are drawn from (and sent to) GNURadio. The system parameters are configured to emulate various channel coherence conditions. Doppler Shift is varied between 400Hz to 4KHz, translating to channel coherence time of 1ms to 100 μs. This captures the range of mobile channel conditions. We sent 25000 packets of size 700 bytes for each Doppler Shift. The channel is replayed for every scheme for performance comparison.

We present the rate selection accuracy for each scheme under varying channel coherence time in Table 3. We show only the accurate and over-selection percentages and omit the under-selection percentages (which can be inferred as they total 100%) for clarity. SNR-based schemes underestimate or overestimate the rate in

	Coherence Time							
	1ms		500μs		200μs		100μs	
Scheme	Accuracy	Over-Select	Accuracy	Over-Select	Accuracy	Over-Select	Accuracy	Over-Select
AccuRate	98%	1%	98%	1%	97%	2%	95%	3.1%
SoftRate	83%	0%	86%	6%	78%	4%	80%	14%
SNR	79%	14%	57%	21%	60%	24%	54%	18%

Table 2: Rate selection accuracy under various fading conditions (simulated).

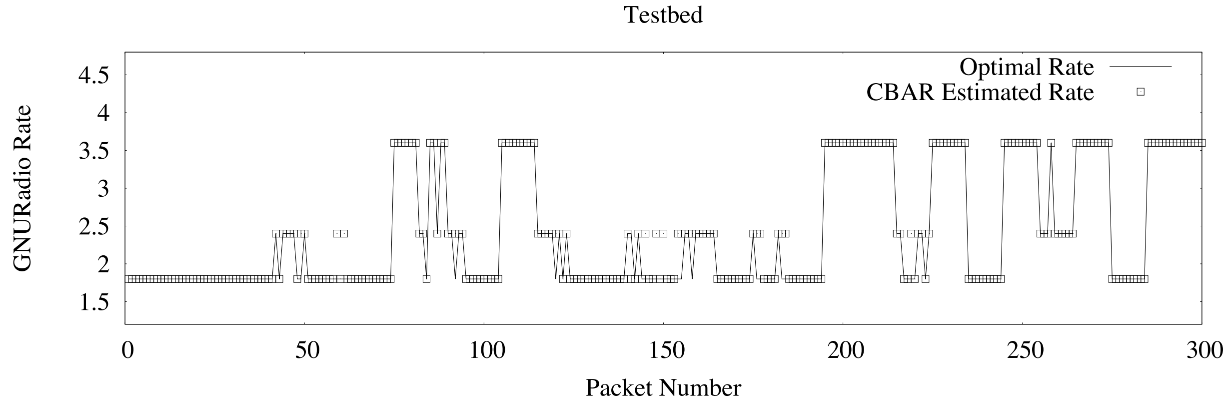


Figure 11: Close-up of AccuRate rate selection under time varying channel.

around 40% of the cases when the coherence time is less than 1ms. This is due to the changes in SNR-BER relationship with the change in coherence time. Also, SNR is calculated only during the preamble which does not capture the entire packet duration. The accuracy of AccuRate and SoftRate remains relatively consistent across different coherence times, though AccuRate still outperforms SoftRate by around 12%. This is an effect of fast-changing channel conditions, requiring a rate estimation scheme to jump multiple levels in one step. AccuRate executes these jumps effectively.

Interference Detection and Rate Selection: Rate estimation under interference is a challenging problem. A receiver must first detect that there is interference. It should then estimate the dispersion due to fading alone to determine the best rate for the packet under fading. Existing schemes have focused on discriminating between fading and interference, and have proposed to backoff when losses are due to interference. AccuRate tries to characterize fading even in case of interference losses [1], and account for fading alone in rate prescription. To evaluate these capabilities, we first evaluate AccuRate's accuracy in detecting interference, followed by rate selection under both interference and fading.

Interference Detection Accuracy

In our experiment, we varied the position and power of the transmitter and interferer to obtain various realistic topologies. As a result, the SINR varies from 0 to 12 dB. We ensured that the primary link has high packet delivery probability ($\geq$ 0.9) in the absence of interference. Now, under interference, we considered

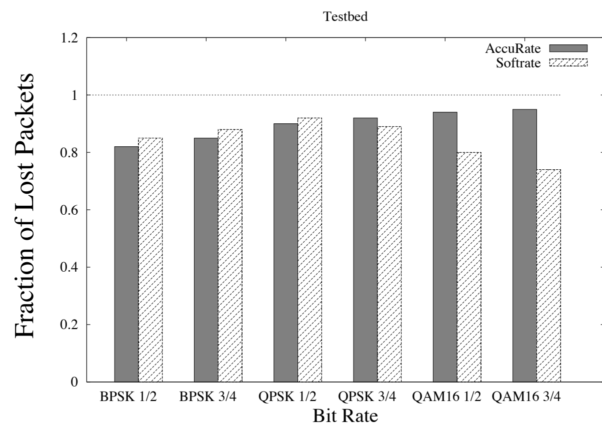


Figure 12: Interference detection accuracy for different transmission bit-rates

the packets that failed the CRC check. We compute the symbol dispersion distributions D^{pre} and D^{post} for the preamble and postamble of the CRC-failed packet. We attribute the loss to interference if these distributions are not "similar". Two distributions are declared similar if more than 50% of samples of one distribution falls within three sigma limits of the other distribution (we model the dispersions with a Gaussian distribution). Figure 12 presents the detection accuracy results for varying transmission rates. AccuRate's interference detection accuracy is slightly lower than SoftRate for low bit rates. This is because low bit rates can tolerate high dispersion and therefore dispersions of preamble

and postamble tend to be similar even in case of a loss with interference. On the other hand, AccuRate performs much better than SoftRate at higher rates and accurately diagnoses loss in more than 95% cases.

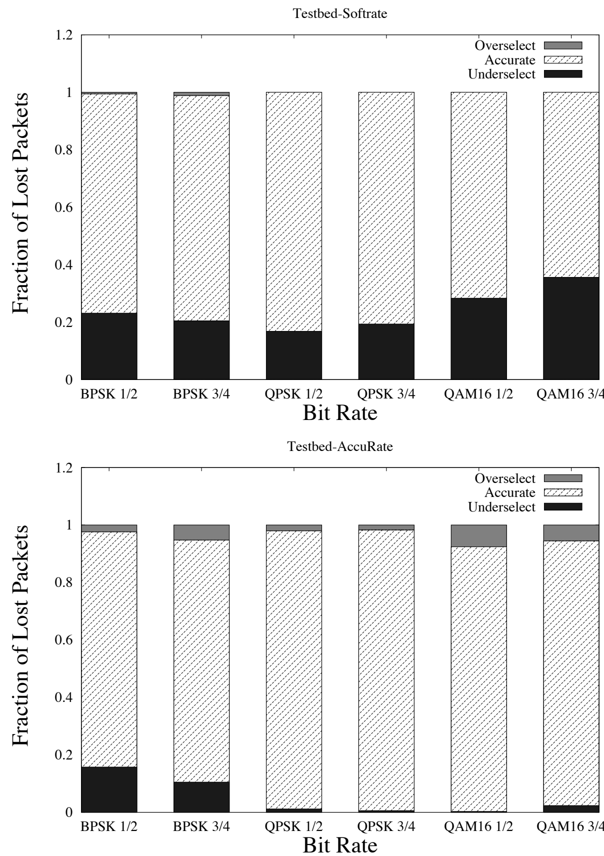


Figure 13: Rate prescription accuracy in the presence of interference: (a) SoftRate; (b) AccuRate.

Rate Estimation Accuracy with Interference
When interference is detected by AccuRate, it estimates the rate supported by the channel under fading alone based on the lower of the dispersion values among preamble and postamble. Fig. 13 compares the rates prescribed by SoftRate and AccuRate with the optimal rates. In the presence of interference, SoftRate's estimation of the rate supported by the channel is not optimal in 20% to 30% of the cases. Whenever SoftRate fails to identify interference, it computes a conservative rate. AccuRate does not perform well at lower rates either (as explained above), but is still better than SoftRate. On the other hand, AccuRate performs quite well at higher transmission rates as it prescribes the best rate in above 92% cases. These results show that AccuRate is quite robust under varying channel conditions with slow/fast fading and with/without interference.

Throughput

The projection of effective rate selection on the link's throughput is of interest. Figure 14 compares the throughput between AccuRate, SoftRate, and SNR-based rate estimation. The simulation results in Figure 14(a) are obtained for varying channel coherence times. As expected, all schemes suffer performance degradation with shorter coherence times. However, AccuRate's ability to pick the optimal rate from correctly received packets permits the subsequent packet to succeed as well. This is a positive feedback that results in good performance, particularly because a large fraction of the packets are received correctly. SoftRate outperforms SNR-based rate selection, but still remains below the AccuRate throughput.

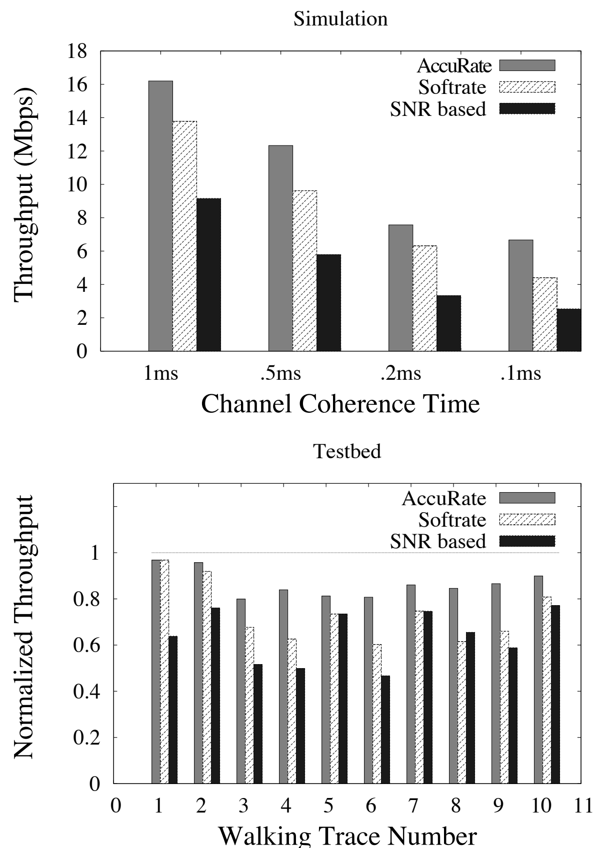


Figure 14: Throughput comparison under (a) simulated slow fading channels, (b) walking experiments on the USRP/GNURadio testbed.

Figure 14(b) shows the throughput comparison from testbed experiments (the receiver was moved with walking speeds). We briefly summarize the experiment methodology here. Recall that 7 short back-to-back packets (called a packet-train) are being repeatedly transmitted to determine the optimal rate during each train. For a packet-train T_i, say packet j's rate was estimated

by AccuRate, denoted as R^*_{ij}. Now, for packet-train T_{i+1}, the short packet that was transmitted at rate R^*_{ij} is selected. Observe that, when running as a full system, this is the packet that AccuRate would have transmitted. Now, if R^*_{ij} was an incorrect estimate, this packet in T_{i+1} would be received in error, implying that AccuRate would have to make the next estimate based on this erroneous packet. Continuing this process, we compute the number of packets successfully received at the receiver, and the total time incurred for their transmissions. The throughput for SoftRate and SNR-based-Rate are also computed as above. The Optimal throughput is computed based on the highest achievable rate in each packet-train. Evidently, AccuRate consistently outperforms SoftRate and SNR, except in a few occasions where the performances are comparable. On average, AccuRate achieves 87% of the optimal throughput possible while SoftRate accomplishes 75%. We summarize by observing that SoftRate leaves a small room for improvement, and AccuRate makes that room even smaller.

Efficacy of Pre/Postamble based Models

When packets are received erroneously, recall that AccuRate uses only the known preamble and postamble to model the channel-induced dispersion. This is clearly an approximation and will cause sub-optimal rate estimation. Moreover, the postamble is an additional overhead, and hence, reducing its size is of interest. Figure 15 illustrates the performance of rate estimation with preambles and two different-sized postambles. In the simulation results (Figure 15(a)), the performance with preamble alone achieves around 80% accuracy, 12% rate over-selection, and 8% under-selection. Including half the postamble improves the accuracy to around 89%, while the preamble and the postamble together can offer nearly 98% accuracy. Of course, in the testbed results (Figure 15(b)), the rate estimation accuracy degrades because the pre/postambles may not always capture the dynamism of the wireless channel. Thus, while estimating the rate of the received packet, AccuRate may be optimistic about the channel fluctuations, thereby selecting higher rates. However, we still find that the degradation is slight. When the preamble and postamble are both used, rate over-selection is around 5%, and accuracy is 94%. We note that the testbed experiments are performed for walking scenarios. We believe that in exchange for the postamble overhead, a 5% over-selection and a 94% rate estimation accuracy is a decent tradeoff. Note that with correct reception of the packet, rate estimation accuracy increases further.

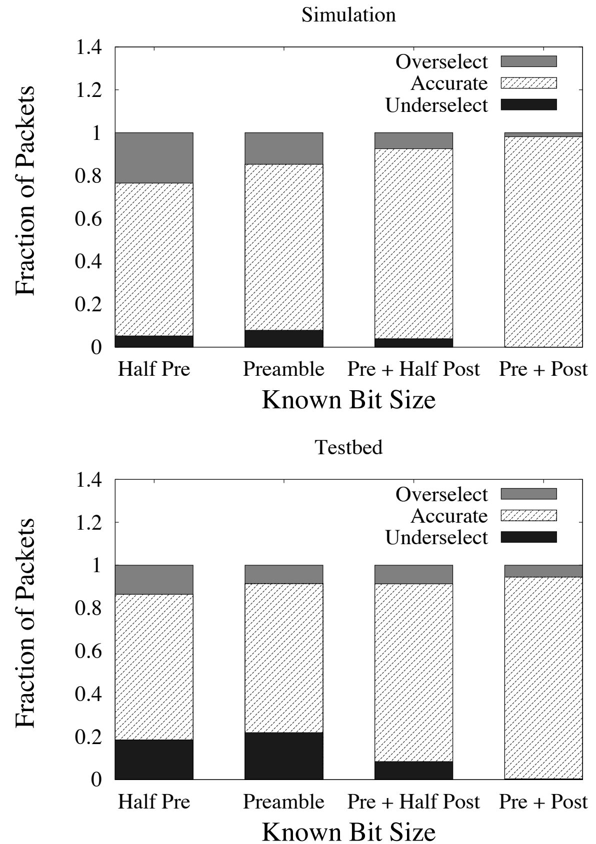


Figure 15: Efficacy of pre/postamble based dispersion models for (a) simulation, and (b) testbed. Simulations configured with slow fading channels, while testbed results are from walking experiments.

7 Deficiencies, Ongoing Work

AccuRate is promising but not yet ready for full-scale deployment. We discuss some of the deficiencies and directions of ongoing work.

(1) Pre/Postambles produce inaccurate channel modeling. When a received packet fails the CRC check, AccuRate extracts only the (BPSK modulated) preamble and postamble to model the channel-induced dispersions. These dispersions are replicated to form a packet-long dispersion sequence, and "replayed" on packets at other rates. Clearly, this is an approximation and does not capture channel variations that may have occurred between the preamble and the postamble. Naturally, AccuRate's selection accuracy deviates from the optimal. One way to address this issue could be to introduce midambles in the packets, i.e., known symbols that are interspersed with the actual data symbols. Midambles will offer additional "glimpses" into the channel's behavior, allowing for a better channel dispersion model. We are investigating the potential benefits of midambles as a part of our ongoing work.

(2) Overhead of ambles. Postambles and midambles are overheads introduced by AccuRate. This overhead can be viewed as the price of improved rate estimation

accuracy (for erroneously received packets). If this overhead is deemed unacceptable (perhaps for shorter packets), we plan to test a few other ideas. First, the 4 pilot tones used for equalization in each OFDM symbol may be used to replace the post/midambles. These tones are known, and may serve AccuRate well for estimating dispersions. Second, we observe that SoftRate does not rely on the post/midambles; instead they utilize the confidence values of all (correct or incorrect) symbols. We envisage that when the packet has failed, Softrate could be triggered to pick reasonably good rates. When the packet is received correctly, AccuRate could predict the optimal rate. Such a fusion of the two schemes is likely to be better than any one.

(3) Implementation complexity. Our primary focus while designing AccuRate has been on estimating the optimal bit rate, unconstrained by the complexity and cost of its implementation. In practice, the hardware cost and the implementation complexity will be high. We intend to optimize for these factors in our future work. However, even if cost and complexity can be sidestepped, AccuRate will still need to meet the latency constraints of IEEE 802.11 (i.e., the rate estimation process must complete with SIFS time window of 9us). This may be a concern even when implemented in hardware. However, we observe that several components of AccuRate, as organized in Figure 9, are amenable to pipelining and speculative operation. For instance, while the symbols are being received, one may speculate correct packet reception and form the dispersion vector from the already-received symbols. This dispersion vector can be "replayed" on a random packet, and hence, the "replay" operation can be pipelined with the actual over-the-air reception. Since this packet is assumed to be correct, replaying needs to be performed only for bit rates that are higher than the packet's actual transmission bit rate. The replay operation can easily be performed at least as fast as the actual reception (potentially using the same hardware), and hence, the SIFS constraint can be met for correctly received packets.

To account for the case of packet failure, the dispersions can be modeled from the preamble alone, and replayed in a parallel hardware pipeline. This will also meet the timing constraints, but at the expense of less accurate dispersion model. While using the postamble will improve this model, its implementation may violate timing constraints because the receiver will have to wait till the end of packet reception to perform the replay. To address this concern, we envisage trading off hardware cost, interference-detection accuracy, or per-packet overhead. (1) By incurring a greater cost, the receiver could incorporate a "bank" of replay chains. Once the postamble arrives at the end of the packet, the receiver could model the dispersion, and replay them concurrently on symbols from the tail of the packet. (2) If this hardware cost is unacceptable, an alternative could be to move the postamble earlier in the packet. An earlier postamble will model the dispersion in time, which can then be replayed on the symbols arriving over-the-air. At the risk of not detecting interference that arrives during the tail end of the packet, the early postamble may reduce hardware cost and meet the desired time constraints. (3) Another alternative could be to include a midamble in addition to the postamble, which elongates the packet but aids in both accurate and timely rate estimation, and interference detection.

In summary, even if network processors operate at the same speed as wireless reception, it may be possible to meet timing constraints through speculation and pipelining. Depending on the outcome of the final CRC check, the corresponding replay thread can be used. Of course, if processor speed exceeds that of wireless reception, the cost, complexity, and accuracy, is likely to improve in favor of AccuRate. A more careful investigation of this space is a topic of future research.

8 Conclusion

This paper asks the question, *for any received packet, can we determine the optimal rate at which the packet should have been transmitted.* This information is valuable because the optimal rate can help the link layer with impending rate selection. In an attempt to answer this question, we propose AccuRate, a constellation based rate estimation scheme. AccuRate exploits symbol level information to characterize the channel's distortion on the incoming packet, and then "replays" this distortion on other rate-encodings of the same packet. The maximum rate that succeeds is deemed as the optimal rate. When such a retrospective method is used to decide on impending transmission rates, we find that AccuRate achieves higher throughput than SoftRate. The performance is often close to the optimal when time-separation between packets are small and the transmissions are in static or slow-moving scenarios. Our ongoing work is simultaneously focussed on extending AccuRate to high mobility environments, while also making the protocol viable in terms of hardware cost and complexity.

9 Acknowledgement

We sincerely thank our shepherd Hari Balakrishnan, as well as the anonymous reviewers, for their valuable feedback on this paper. We also thank Mythili Vutukuru for her invaluable help with SoftRate implementation. We are also grateful to NSF for partially funding this research through the following grants – CNS-0448272, CNS-0917020, CNS-0916995, and CNS-0747206.

References

[1] Mythili Vutukuru, Hari Balakrishnan, and Kyle Jamieson, "Cross-Layer Wireless Bit Rate Adaptation," in *ACM SIGCOMM*, Barcelona, Spain, August 2009.

[2] J.C. Bicket, "Bit-rate Selection in Wireless Networks," *Massachusetts Institute of Technology*, 2005.

[3] T.S. Kim, H. Lim, and J.C. Hou, "Improving spatial reuse through tuning transmit power, carrier sense threshold, and data rate in multihop wireless networks," in *ACM MOBICOM, 2006*.

[4] A. Kamerman and L. Monteban, "WaveLAN®-II: a high-performance wireless LAN for the unlicensed band," *Bell Labs technical journal*, vol. 2, no. 3, pp. 118–133, 1997.

[5] S.H.Y. Wong, H. Yang, S. Lu, and V. Bharghavan, "Robust rate adaptation for 802.11 wireless networks," in *ACM MOBICOM*, 2006.

[6] G. Holland, N. Vaidya, and P. Bahl, "A rate-adaptive MAC protocol for multi-hop wireless networks," in *ACM MOBICOM*, 2001.

[7] B. Sadeghi, V. Kanodia, A. Sabharwal, and E. Knightly, "Opportunistic media access for multirate ad hoc networks," in *ACM MOBICOM*, 2002.

[8] J. Zhang, K. Tan, J. Zhao, H. Wu, and Y. Zhang, "A practical SNR-guided rate adaptation," in *IEEE INFOCOM*, 2008.

[9] J. Camp and E. Knightly, "Modulation rate adaptation in urban and vehicular environments: cross-layer implementation and experimental evaluation," in *ACM MOBICOM*, 2008.

[10] K. Jamieson and H. Balakrishnan, "PPR: Partial packet recovery for wireless networks," in *ACM SIGCOMM*, 2007.

[11] J. Hodel and R. Clarke, " USRP Fading Simulator," http://www.cgran.org/.

[12] D. Qiao and S. Choi, "Fast-responsive link adaptation for IEEE 802.11 WLANs," in *IEEE ICC*, 2005.

[13] J. Kim, S. Kim, S. Choi, and D. Qiao, "CARA: Collision-aware rate adaptation for IEEE 802.11 WLANs," in *IEEE INFOCOM*, 2006.

[14] madwifi, "Onoe rate control," http://madwifi.org/browser/trunk/ath_rate/onoe.

[15] G. Judd, X. Wang, and P. Steenkiste, "Efficient channel-aware rate adaptation in dynamic environments," in *ACM MOBISYS*, 2008.

[16] P. Acharya, A. Sharma, E.M. Belding, K.C. Almeroth, and K. Papagiannaki, "Congestion-aware rate adaptation in wireless networks: A measurement-driven approach," in *Proc. IEEE SECON Conf*, pp. 1–9.

[17] H. Rahul, F. Edalat, D. Katabi, and C.G. Sodini, "Frequency-aware rate adaptation and MAC protocols," in *ACM MOBICOM*, 2009.

[18] S. Sen, J. Xiong, R. Ghosh, and R.R. Choudhury, "Link layer multicasting with smart antennas: No client left behind," in *IEEE ICNP*, 2008.

[19] S. Sen, N. Santhapuri, R. R. Choudhury, and S. Nelakuditi, "Moving Away from Collision Avoidance: Towards Collision Detection in Wireless Networks," in *ACM HOTNETS*, 2009.

[20] S. Rayanchu, A. Mishra, D. Agrawal, S. Saha, and S. Banerjee, "Diagnosing wireless packet losses in 802.11: Separating collision from weak signal," in *IEEE Infocom*, 2008.

[21] S. Biaz and N. Vaidya, "Discriminating congestion losses from wireless losses using inter-arrival times at the receiver," in *IEEE Symposium ASSET*, 1999, vol. 99, pp. 10–17.

[22] M. Khan and D. Veitch, "Isolating Physical PER for Smart Rate Selection in 802.11," in *IEEE INFOCOM*, 2009.

[23] Z. Zeng and P.R. Kumar, "Towards Optimally Exploiting Physical Layer Information in OFDM Wireless Networks," in *IEEE WiCon*, 2008.

[24] A. Georgiadis, "Gain, phase imbalance, and phase noise effects on error vector magnitude," *IEEE Transactions on Vehicular Technology*, vol. 53, no. 2, pp. 443–449, 2004.

[25] M.D. McKinley, K.A. Remley, M. Myslinski, J.S. Kenney, D. Schreurs, and B. Nauwelaers, "EVM calculation for broadband modulated signals," in *64th ARFTG Conf. Dig*, 2004, pp. 45–52.

[26] E.G. Larsson, G. Liu, J. Li, and G.B. Giannakis, "Joint symbol timing and channel estimation for OFDM based WLANs," *IEEE Communications Letters*, vol. 5, no. 8, pp. 325–327, 2001.

[27] L.R. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Trans. Inform. Theory*, vol. 20, no. 2, pp. 284–287, 1974.

Scalable WiFi Media Delivery through Adaptive Broadcasts

Sayandeep Sen, Neel Kamal Madabhushi, Suman Banerjee
University of Wisconsin-Madison

Abstract

Current WiFi Access Points (APs) choose transmission parameters when emitting wireless packets based solely on channel conditions. In this work we explore the benefits of deciding packet transmission parameters in a content-dependent manner. We demonstrate the benefits specifically for media delivery applications in WiFi environments by designing, implementing and evaluating a system, called *Medusa*. In order to keep the APs relatively simple, we implement the *Medusa* functions in a media-aware proxy. More specifically, when forwarding our media traffic, *Medusa* requires that APs simply use the WiFi broadcast feature, and that they refrain from making decisions on which wireless packets to retransmit, or what PHY rates such packets should be transmitted at. Instead we combine these typical link layer functions with a few other content-specific choices, in the proxy. Through detailed experiments across diverse mobility and interference conditions we demonstrate the advantages of this scheme for both unicast and multicast media delivery applications. The advantages are particularly substantial in multicast scenarios, where *Medusa* was able to deliver a 20 Mbps HD video stream simultaneously to 25 clients, using a single 802.11 AP, with good to excellent PSNR.

1 Introduction

Robust delivery of rich media content over wireless links is an increasingly important service today. As more and more high quality media content becomes available through the Internet, the user expectation of accessing such content over their wireless enabled devices continue to grow. Examples include users watching on-demand shows and movies from sources such as Hulu and NetFlix, students in a university campus interested in following online lectures while sitting in the cafeteria, and employees in a company watching a company-wide presentation by their CEO, whether at home, at work, or while sitting in a coffee shop. While the widely deployed WiFi technology can provide adequate performance for relatively lower quality media streams, the user experience when watching high quality streams (e.g., HD quality content) leaves much to be desired. In this paper, we attempt to push the envelope of media delivery performance for WiFi systems, by exploiting some knowledge of media content in making transmission parameter selection at the wireless transmitter (APs). While our proposed approach applies equally to unicast as well as to multicast scenarios, the biggest advantage of the system arises in the multicast case where multiple users are interested in the same content.

A target campus application and challenges: The IT department of the UW-Madison campus is interested in providing high quality broadcast of specific educational content through its intranet. Such capability would allow students sitting in dormitory rooms, in union buildings, in cafeterias, and in libraries, to follow the classroom. Further, the system would also allow easy and convenient dissemination of live guest lectures from remote locations, without requiring the guest lecturer to visit the campus. While the wired backhaul has sufficient capacity to carry such media traffic, initial experiments by the IT department revealed obvious performance problems on the WiFi based last hop. In particular, they observed that even if 3 users connected to a single 802.11g AP attempted to watch the *same* HD video stream, the performance of the system was abysmally poor [1].

The poor performance of media delivery over WiFi for multiple users requesting the same content, is a combined effect of three factors: (i) HD quality video places a high bandwidth demand on the wireless medium — commercial HD encoders, such such as the Streambox SBT3-9200 [6], create content with data rates ranging from 512 Kbps to 30 Mbps, (ii) WiFi typically employs 802.11 unicast mode for sending similar content separately to each user, and (iii) the WiFi transmitter makes various configuration choices, e.g., PHY transmission rates, number of re-transmission attempts, etc., for each wireless packet, without any knowledge of the relevance of its(packet's) contents to end applications. In this paper, we design and implement a system called *Medusa* — Media delivery using adaptive (pseudo)-broadcasts, that can efficiently address issues (ii) and (iii).

1.1 *Medusa* approach

The 802.11 transmitter typically transmits packets in the FIFO order and makes multiple decisions for each wireless packet transmission. This include channel contention, i.e., when to attempt wireless transmission, selection of the PHY rate for the packet, and the number of re-transmission attempts to make in case of failures. In this paper we contend that many of these decisions, namely PHY rate selection, number of re-transmission

[1] An 802.11n AP can potentially scale performance to upto 10-12 users watching such HD content simultaneously. The typical number of users in busy parts of campus is often much higher.

attempts for each packet, and the order of packet transmissions, are better made by taking the "value" of the wireless packet to the application into account as well. Let us consider MPEG-encoded [4] video content that consists of I-, P-, and B- video frames. Given that a packet carrying I-bits is more important, the PHY rate of such a packet can be picked more conservatively, than value-unaware rate adaptation algorithms, e.g., SampleRate [8]. This would ensure that the loss probability of packets carrying I-bits are particularly low. Similarly, if the wireless channel capacity is scarce and packet errors are high, then it is more important to devote greater re-transmission effort for packets carrying I-bits, than packets carrying P- and B-bits.

Hence in *Medusa*, we offload these decisions for our media traffic to a media-aware proxy that can interpret the value of the data. More specifically, APs in *Medusa* no longer perform link-layer re-transmissions or PHY rate selections. Instead, the proxy examines the value of each packet to applications, and instructs APs to (re)transmit these packets in a certain order and at specified PHY rate.

Prior work, e.g., Trantor [21], has considered a model in which a centralized controller decides transmission parameters, e.g., PHY rates, transmit power, etc. for different APs and clients in an entire enterprise WLAN. At a high level, our proposal of proxy-based PHY rate selection and re-transmissions may appear similar. However, there is a fundamental difference between the two proposals. Trantor suggests a centralized rate selection in a content-agnostic manner, based solely on potential interferences between different conflicting wireless links. The approach in *Medusa* augments this decision by incorporating knowledge about *value of the packet contents to applications* in deciding the PHY rate of packets, the order in which they should be transmitted, and the number of re-transmission attempts to be made. It also optionally utilizes simple network coding approaches [23] to improve efficiency.

Unicast vs broadcast vs pseudo-broadcast: In a scenario where multiple users are requesting the same content (the same media stream, for example), we advocate the use of 802.11 standard's broadcast mode of operation. The choice is motivated by the observation that a 802.11 broadcast packet can be used to communicate content simultaneously to all receivers. This would substantially reduce the load on the wireless medium. However, MAC-layer broadcast packets do not elicit MAC-layer acknowledgments from receivers, leaving the transmitter unaware of losses on the wireless channel. This is a problem for any broadcast-based wireless system, as the transmitter can no longer decide which packets require re-transmissions. Further, absence of loss information makes it impossible to determine an appropriate

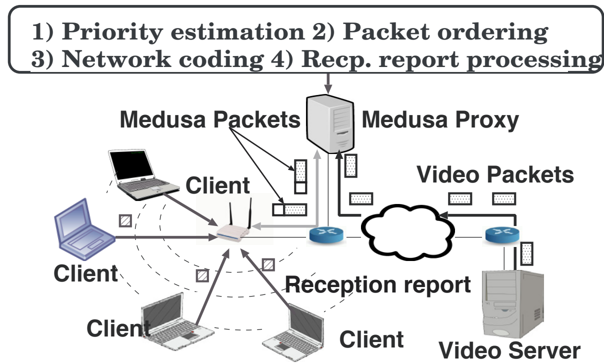


Figure 1: Schematic of the *Medusa* shared media delivery system.

PHY rate adaptation mechanism. Hence, we incorporate higher layer acknowledgments from the clients to the proxy, that help the latter in making these decisions for the APs in a content-dependent manner.

In a single user scenario, we could use 802.11 unicast transmissions. However, even in such settings, it is possible to exploit ER-style network coding opportunities when transmitting wireless packets [23]. Such network coded packets, by definition, have multiple intended recipients. Broadcast-based 802.11 packets are suitable to facilitate this enhancement as well.

Lack of MAC-layer acknowledgments is problematic, however, for one more link layer decision that has to be taken by the APs — channel contention. Lack of acknowledgments indicating packet losses, prevent APs from inferring how the MAC layer backoff counters should be adjusted. Given the lack of MAC-layer acknowledgments for broadcast packets, the existing methods for channel contention are likely to fail. Hence, we use pseudo-broadcast packets in *Medusa* to communicate all broadcast media content, analogous to what was proposed in COPE to deliver network coded multicast data [18]. In pseudo-broadcast, 802.11 unicast mode is used where one of the receivers is picked at random to be the explicitly stated (unicast) recipient, while other intended recipients simply overhear the packet. MAC-layer acknowledgments arrive from the explicit recipient, and backoff parameters can be appropriately adjusted. Note that these MAC-layer acknowledgments are interpreted by the APs solely for the backoff adjustment function, and not used by the proxy to decide PHY rate, transmission order, or eligibility for re-transmission of packets.

***Medusa* system overview:** Figure 1 shows a schematic of different aspects of the *Medusa* system, including an unchanged media server, a media-aware *Medusa* proxy, and APs and clients with minor software-level changes. The *Medusa* proxy intercepts all IP packets corresponding to various video frames and relays

them to the AP for transmission. The proxy instructs the AP to use 802.11 pseudo-broadcast for each wireless packet (irrespective of whether it is part of a multicast or a unicast stream) and also informs the AP what specific PHY rate to transmit the packet at. The *Medusa* proxy makes these decisions, using a combination of four mechanisms: *(i) WiFi reception reports:* Each client provides a periodic reception report to the proxy about various wireless packets that it did or did not receive. While analogous to Reception Reports in RTCP [24], the reception reports in *Medusa* differs from those in RTCP in the detailed MAC layer information that is carried in *Medusa* for rate adaptation and re-transmission purposes. *(ii) Estimating the value of a packet to media applications:* Not all packets are of equal importance to receivers. When the wireless channel capacity is scarce, the value of each packet determines the choice of PHY rate and the number of re-transmission attempts to be made to deliver the packet. We use a simple per packet value assignment function at the *Medusa* proxy to determine the priority level of each packet encapsulating a media stream. *(iii) PHY rate adaptation and re-transmissions for broadcast packets:* We design a PHY rate adaptation and re-transmission strategy for broadcast wireless packets that cannot depend on MAC-layer acknowledgments. Our rate adaptation scheme is *two-paced*. A conservative baseline PHY rate is identified at a slow timescale, and an individual PHY rate for each packet is chosen subsequently using an algorithm called *Inflate-Deflate*. *(iv) Packet order selection and network coded re-transmissions:* We modify the ordering of media packets from traditional FIFO, especially when the channel quality is poor. Under bad channel conditions, it is beneficial to prioritize packet transmission based on packet "value". This would increase the probability of successful reception of important packets. Further, the transmission order is also selected such that there are proxy-initiated re-transmission opportunities for the more important packets. During re-transmissions, we also leverage the gains of ER-style network coding.

Key contributions: Summarizing, the key contributions of this work is two-fold:

(i) We propose an intuitive design of a pseudo-broadcast based WiFi system for media delivery at high video rates. The design incorporates various aspects of rate adaptation, re-transmission, and packet priorities, coupled with higher-layer feedback.

(ii) We integrate all of these ideas together into a functional *Medusa* system and present detailed evaluation of this system. Our results show that *Medusa* provides robust, high-bandwidth (upto 20 Mbps), HD quality media delivery to tens of co-located WiFi users interested in the same content, all sharing the same WiFi AP, across a range of channel conditions and mobility scenarios. Our technique is applicable to unicast media delivery scenarios as well.

2 *Medusa* design overview

We describe the design of *Medusa* by using the example of MPEG4-encoded [4] video delivery over wireless. In MPEG4 the video content is partitioned in an independently decodable sequence of pictures, called Group of Pictures (GOP). Each GOP has frames of three different types: I, P and B. Each frame, in turn are broken down into multiple packets which are then transmitted over wireless channel.

At the receiver, the I frames can be correctly decoded, as long as all constituent packets are received. For decoding P packets, the successful reception of previous I or P frame in sequence is also necessary. Finally, to decode a B frame, the previous as well as the next I or P frame in sequence are needed. Put another way, an I frame does not depend on any other frame, while P frames depend on another frame and B frames depend on two other frames.

As described, *Medusa* involves an unchanged media server, a media-aware proxy, and APs and clients, with small software modification. The only change in the AP is to create a single functionality — for each packet forwarded by the proxy to the AP, the proxy should be able to specify the PHY rate of transmission. The AP simply accepts each such packet (which can be an original packet, a previously transmitted packet, or a network coded packet) and transmits them using the pseudo-broadcast mode using the PHY rate specified by the proxy. The AP continues to perform back-off decisions independently based on MAC-layer acknowledgments received for its pseudo-broadcasts. The client is modified to incorporate WiFi reception reports targeted to the proxy. These reception reports include a higher layer acknowledgment (ACK) bitmap, and is sent infrequently by the clients, roughly once every 100 packets or 100 ms. Since the proxy knows the PHY rates at which different packets were transmitted, it can use these reception reports to infer packet losses observed by different clients at different PHY rates.

Based on this simple setup, the design problem of *Medusa* can be stated as,

For a set of k video packets (including both original packets and packets that need re-transmissions), determine the order of transmission and PHY rates for the packets, and pass this information along with the packets to the APs. Further, determine whether some of these packets should be network coded, and whether some of them should be discarded.

We present a particular solution to the above problem in the rest of this section that exploits knowledge of the value of each packet to applications.

While we describe our scheme for MPEG4 video, our approach generalizes to any other media encoding, where the content is structured in layers, and there are different levels of priority (value) for each layer.

2.1 Determining value of packets

As mentioned above successful decoding of video frames might need reception of other video frames. Hence, all else being equal, the value of each video packet depends on how many other packets (or bytes) depend on this packet for correct decoding of various video frames. Naturally, I-frame packets become more important than P- or B-frame packets. The value of video packets is also influenced by its impending playback deadline and that of other dependent video frames. Packets for video frames that are approaching display deadline are more important. Finally, given that many packets are relevant to more than one client (true for original and re-transmitted and network coded packets), the value of a packet should also grow with the number of intended recipients.

Previous research on video encoding for streaming [9, 12, 19, 26] has proposed LP based techniques for determining the priority of video frames. Such techniques typically utilize a directed acyclic graph (DAG) of video frame dependencies along with a (empirical/theoretical) channel error model to determine relative value for frames.

While such sophisticated designs of packet value can certainly be used, in this paper we consider a relatively easy to compute function to determine packet value to applications that illustrates its usefulness in making rate adaptation, re-transmission, packet ordering, and network coding decisions based on the worth of packet contents.

In our scheme, we assign a weight, X, to each video frame, based on how many bytes the frame can help decode (including itself). This weight is given to all constituent packets of this frame. Our media-aware proxy knows the video encoding process, and can calculate X by buffering and observing packets corresponding to each GOP before making transmission decisions. We next assign a weight, C, proportional to the number of intended recipients of each packet. Finally, we assign a third weight, D, based on the delay until the display deadline of this frame. We normalize all these weights to the same scale, and assign the value of the packet to be the product of these normalized weights, thus,

Value $= \bar{X} \times \bar{C}/\bar{D}$.

2.2 Determining a base PHY rate

Our PHY rate selection process is two paced. Initially, we compute a conservative PHY rate for all the packets. If channel capacity is abundant, all packets will simply be transmitted at this rate to enhance the possibility of successful reception. However, if the channel is error prone, then some of these rates will be updated, as described in Section 2.3. The timescale for adapting base PHY rate depends on the reception report frequency of clients (roughly every 100 ms in our current implementation).

We pick the highest PHY rate such that the expected error probability of the packets at all clients would be below a certain threshold (set to a low value) as the base PHY rate. By retaining PHY rate information for all transmitted packets, the proxy, on receiving ACK bitmaps from client, can infer the necessary error rates. In case statistics for certain PHY rates are missing (possibly because no packets are sent at these rates), standard interpolation techniques can be used to estimate the expected error rates.

An important distinction of our broadcast rate assignment from typical 802.11 unicast rate assignment algorithms([8, 30]) is that, it does not favor a higher rate to merely increase the channel utilization. Instead it tries to ensure high reception probability across multiple broadcast receivers (who might have diverse channel conditions).

Another distinction of *Medusa* rate adaptation from unicast stems from inability of *Medusa* to adapt quickly to changed network conditions, due to delayed ACKs. This can result in *Medusa* persisting at high data rate even when channel quality has deteriorated resulting in high errors. To ensure that such a situation does not occur, we update the error characteristic of the PHY rates with a EWMA function with heavy weight on history(thus preventing it from reacting to transient fades of the channel).

Transmitting packets at base PHY rate ensures that the packets have a good likelihood of successful reception. However, if the base rate of the system is too low (say, due to presence of clients with bad channel quality) which limits successful transmission of all video packets before their respective deadlines. Under such circumstances, we selectively increase the transmit rate of different packets in a certain order as described next.

2.3 Packet order and actual PHY rate

The problem of deciding the video packet schedule while maximizing the delivered quality across a group of users is known to be NP-complete [12]. Hence, we use a heuristic algorithm for packet ordering. We now describe our mechanism to determine the transmission order and PHY rate of packets, using an algorithm, we call *Inflate-Deflate*. The heuristic schedules a batch of packets in each round. For ease of exposition we assume the presence of a virtual timeline and our goal is to place packets on this timeline and determine their PHY rate. Placing the packets at a given timeslot signifies *schedul-*

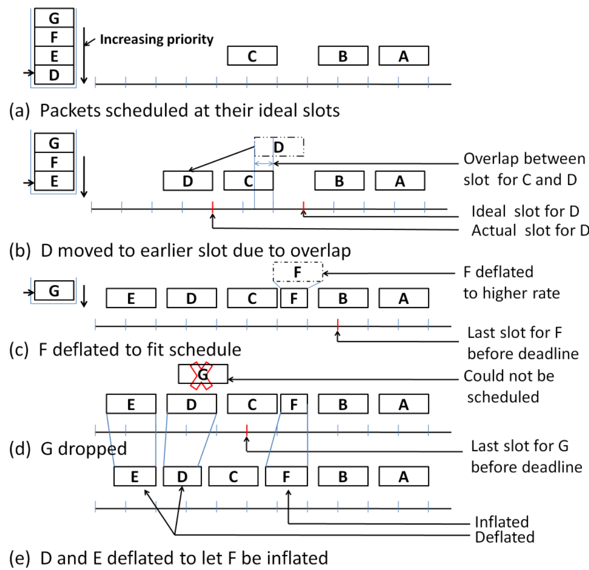


Figure 2: Packet ordering and final rate assignment carried out by *Medusa*.

ing the transmission of the packet at that instant. Packets (including retransmission candidates) are added onto the timeline in the decreasing order of packet value (as calculated in Section 2.1), i.e., higher valued packets are placed first, and lower valued packets later.

We describe our algorithm for ordering packets next and illustrate it with a toy example. We consider A, B, C, D, E, F and G as the seven packets which need to be transmitted (Figure 2). The width of a packet signifies the time required to transmit the packet at its base PHY rate. Initially we try to place each packet at its *ideal* timeslot — a time by which it needs to be transmitted so that it can be re-transmitted multiple times in case of consecutive losses. Also, the latest timeslot at which a packet can be placed is when it just makes its playback deadline for the slowest client, called the *deadline slot* for the packet. The reason for not placing a packet at the earliest available slot, is to ensure that packets which have lower value but have an earlier deadline than the more important packets still get a chance to be scheduled. When the current time is past a packet's deadline slot, and it is not required for decoding any other packet at any receiver, the packet can be discarded. We now walk through the example of how different packets get placed on the scheduling timeline using the following cases.

Case-I (Sufficient time is available to schedule all packets at ideal time-slot): This scenario is depicted in Figure 2(a). Here, packets A, B, and C are scheduled at their ideal timeslot. Also, the packets are assigned their base rates.

Case-II (Ideal slot occupied by a higher valued packet): Under realistic settings, contention from other traffic sources and the necessity to retransmit different packets would mean that scheduling all packets at their ideal slot might not be possible. We depict such a situation in Figure 2(b), in this case scheduling packet D at its ideal slot would lead to an overlap with packet C. We mandate that the packet with the lower value be the one which gets moved around. We find a best fit timeslot in the schedule for the lower valued packet D. Note by considering packets in order of their value implies that only the current packet needs to be moved.

Case-III (No slots left at current PHY rate): In extreme case, we might be unable to find a big enough time slot for a packet, for example in Figure 2(c) we are unable to find a timeslot big enough to fit packet F at its base rate, before its deadline. In such a case, we increase the data-rate of the packet, we call this operation *Deflate* as it results in shortening the packet dimensions on the transmission timeline. For example, we were able to fit F on the timeline after deflating operation. In this operation, we keep trying to find a best-fit timeslot for the packet by increasing its rate. This process continues till we find a slot to fit the packet, or we exhaust all rates without being able to fit the packet on the timeline. This would imply the inability to schedule the packet in the current round. We show this in Figure 2(d). Packet G could not be placed in the timeline even at the highest PHY rate and hence, had to be left out from current iteration.

Case-IV (Compensating for rate optimization): Packets with only a few intended recipients(say, ones with bad channel quality) would have lower value. Thus, such packets would potentially be transmitted at higher rates. This might lead to drastic degradation in received video at clients with bad channel quality. To remedy this, we carry out a round of rate re-assignment before sending out the packets. We call this operation *Inflate* as it decreases the rate of some packets, thus, increasing their size on the transmission timeline. Note that the inflate process does not increase the size of the timeline itself. Inflating is carried out by going through the list of active clients and calculating the expected distortion in video quality, they would suffer if the packets are sent out according to current plan. In case the expected quality of a client falls below a certain minimum threshold, we find out the packet which can increase the expected quality of reception the most and we decrease the rate of transmission of the packet. We then try and compensate by deflating a few other packets which would minimally decrease the quality of video at clients. This process is illustrated in Figure 2(e). In this example packet F is deemed a valuable packet for a client with poor channel quality and hence, its rate is reduced, while the transmission rates of D and E are increased as a compensation.

Note that *Inflate*, might lead to overall reduction in quality of video received over all clients. We keep it in

order to ensure that a minimal quality of video is served to each client.

We would like to note that once this order has been determined, *we do not delay the transmission of packets until the scheduled timeslot*. Instead the packets are sent out at the next transmission opportunity. This ensures that we get even more opportunities to retransmit the packets before its deadline expires. The virtual timeline and time slots are, thus, used only to determine the order of transmissions and the corresponding rates.

An interesting aspect of the PHY rate selection using Inflate-Deflate is that many packets can get transmitted at distinct rates based on the rate assignment algorithm. As a consequence, the proxy can get feedback on a large range of PHY rates from the clients, without having to explicitly raise the base PHY rate. This is another difference between the rate adaptation and error rate estimation technique employed by *Medusa* from unicast rate adaptation techniques such as SampleRate [8].

2.4 Re-transmission planning

We discuss the three inter-related components of re-transmission planning next.

Timeout estimation: A key issue in planning for re-transmissions is to determine the timeouts accurately — under-estimating would lead to redundant packet transmission, while over-estimation would lead to video packets missing their playback deadline. Since each client reception report acknowledges a block of packets, we have to adjust the round trip time and the timeout calculations, to account for additional delays incurred in clients. We adopt a TCP-like Exponentially Weighted Moving Average (EWMA) mechanism for RTT estimation, which takes into account this change. Furthermore, we re-compute the value for all packets that become eligible for re-transmissions, and use this new value to determine the packet transmission ordering and PHY rate.

Network coded re-transmissions: As packet errors at different locations occur independently, multiple clients would potentially (not) receive different packets from a set of consecutive transmissions. This allows us to deploy a simple XOR-based coding [23] of packets to be re-transmitted, to further optimize channel utilization. In our system, we XOR-code a group of packets, only if they satisfy the following rule: *Out of a set of packets to be re-transmitted, if a subset of packets can be found such that each intended recipient of a specific packet has received all other packets in the subset , then the subset can be network coded.* Such coding opportunities occur frequently in the proxy, as MAC-layer re-transmissions are not used in *Medusa* . The algorithm for network coded re-transmissions is shown in Algorithm 2.1.

A key decision in our design is to determine the set of packets to be coded after the packet order has been decided. This is done to keep the packet ordering algorithm simple, as otherwise the algorithm would have to deal with coded packets (with multiple constituent packets of different values), while deciding the sending order and rate. A coded packet is always transmitted with the intended PHY rate of the first packet in the set. This ensures that the probability of error in receiving the first packet at its intended receivers is not hampered, while opportunistically delivering other packets in the coded set to their respective clients. At the client side all received packets (natively or from network coded packets) packets are maintained till their deadline expires. This is done to ensure that packets coded with previously received packets can be recovered. The client sends back acknowledgment for packets which are successfully decoded as part of reception reports.

Algorithm 2.1: NETCODE(P)

INPUT P: set of coding candidates,
arranged in decreasing order of packet values
$Coding_set$: Set of packets to be coded
$P_i.client_set$: Set of clients interested in P_i
OUTPUT S: set of coded packets
for each $P_i \in \mathcal{P}$
  do $Coding_set \leftarrow \phi$
for each $P_j \in \mathcal{P}, j > i$
  do $\begin{cases} \textbf{if } is_coding_worthy(P_j, Coding_set) = \textbf{ true} \\ \textbf{then } Coding_set \leftarrow P_j \end{cases}$
if $Coding_set \neq \phi$
  then $\begin{cases} X \leftarrow make_coded_packet(P_i, Coding_set) \\ X.rate \leftarrow P_i.rate \\ S \leftarrow X \end{cases}$
  else $S \leftarrow P_i$
return (S)
procedure IS_CODING_WORTHY$(P_j, Coding_set)$
  for each $C_i \in Coding_set$
    do $\begin{cases} \textbf{if } P_j.client_set \cap C_i.client_set \neq \phi \\ \textbf{do return } (\textbf{ false }) \end{cases}$
  return (**true**)

Delayed Packet discard: The deadline for packet delivery shifts over the duration of a streaming session. We initially set it to the playback deadline of the frame, which is calculated using the following formula,

$$Deadline = \frac{\text{Frame}_{\text{seqno}}}{\text{Frame Rate}} + Playback\ buffer\ size + \delta,$$

where, the deadline is number of seconds from the transmission time of the first video frame, $\text{Frame}_{\text{seqno}}$ is the sequence number of the frame. Frame Rate is the number of frames that the video player needs to display in a second. Playback buffer size is the amount of time

(in seconds) that the receiver can store the video before it needs to start decoding the frames. And, δ is a small time constant added to account for initial frame delay.

Once the playback deadline of a packet expires, we reset the deadline for delivering its constituent packets to that of the next frame which depends on the successful reception of the packet for decoding. This goes on until the packet is delivered to all clients, or the deadlines of all the frames which depend on the current packet have expired. We drop the packet from our system at that instant.

Similarly, at client, we discard a packet only if its playback deadline has expired and the packet is not useful for decoding any other frame.

3 Putting it all together

We have implemented the *Medusa* proxy and client. The implementation consists of about 3.5K lines of C code. We stream video using the Evalvid tools package [1]. We modified Evalvid to provide information about dependency structure of video frames, frame type of the generated packet and the deadline of the packet. The *Medusa* proxy runs as an application level process. We modified the MadWiFi driver to carry out per-packet rate assignment. Per packet rate assignment is achieved by specifying the target rate in a header of the video packet and then extracting it out of the packet inside the AP's driver.

At the client, the *Medusa* module keeps information regarding number of packets received and the channel quality. The module passes the received video packets to video playback software such as VLC [7] and MPlayer [5], for displaying. It also keeps a copy of received packets, till the expiry of their deadline for decoding other packets.

4 Evaluation

To study the performance of *Medusa* we have experimented with upto 25 users that are associated to a single AP(operating in 802.11g mode) and attempting to receive HD quality video from the *Medusa* proxy. Our setup consists of 30 laptops with Atheros wireless driver running Linux operating system.

Wireless conditions: The experiments were done on a university building floor. We broadly classify our wireless environment into three types: (i) *Low-loss* environment - corresponding to specific client locations where the packet error rates were 5% or less; (ii) *Medium-loss* environment - corresponding to locations where packet error rates were in the range of 5-15%; and (iii) *High-loss* environment - corresponding to locations where packet error rates were in excess of 15%. For the set of experiments reported, an experiment location did not shift from one to another in the course of experiment.

MOS Rating of video quality	PSNR range
Excellent	> 37
Good	31-37
Fair	25-31
Poor	20-25
Bad	< 20

Table 1: Table mapping the MOS based user perception of video quality to the PSNR range

Video setup: We experimented with different video clips, in this paper we present results for the Mobile calender video clip [3] replayed back to back to run for 2 minutes. The video was encoded at rates of 5, 10, 15 and 20 Mbps using FFmpeg [2] tool with H264 codec. We have repeated each experiment for 20 runs. For our experiments, we used a fixed playback buffer of 10 seconds at clients. We intend to evaluate the benefits of adaptively modifying the playback buffer size in future.

Metrics: We compare the performance of different schemes in terms of Peak Signal-to-Noise Ratio (PSNR), jitter, and overall network load imparted.

PSNR: Is a standard metric for measuring the relative quality of video streams [13,20]. The PSNR of a video is well correlated with the perceived quality of video experienced by the user. The relationship between user perception expressed in Mean Opinion Score (MOS)and the PSNR range were detailed in [17, 22] and are summarized in Table 1.

Jitter: We measure the Instantaneous Packet Delay Variation(IPDV) [14] of received packets as a measure of jitter of the delivered video stream. This metric complements the PSNR metric which is oblivious to the delay and jitter of the delivered video, as it assumes the presence of an infinite playback buffer. High jitter value signifies a bad performance.

Network load: We measure the load placed on the network by different schemes in terms of the a) number of packets transmitted in air and also in terms of amount of air-time occupied by the packets sent by different schemes.

Compared schemes: We compare the performance of *Medusa* to the following alternate schemes.

BDCST: This scheme uses WiFi broadcast to transmit packets. However, unlike normal WiFi broadcast, the PHY rate is chosen to maximize the video PSNR performance averaged across all clients. The PHY rate is selected by sending about 30 seconds of traffic at different rates.

UCAST-INDIV: In this scheme we send the video stream to each client using isolated WiFi unicast, in sequence. For example, if there are two clients, we first send the entire video to client 1 and then the same video

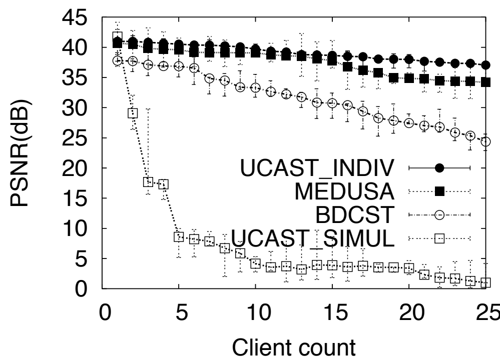


Figure 3: Plot showing the average per client PSNR of different scheme when serving 25 clients with a 20 Mbps video stream, under medium loss conditions. The mean and the variance (errorbars) are shown.

separately to client 2 using WiFi unicast. This scheme gives a quality bound for *Medusa*.

UCAST-SIMUL: In this scheme we send the video traffic to all the clients simultaneously using normal WiFi unicast with SampleRate rate adaptation. This is the traditional method for wireless data delivery.

Note that in all of these alternate schemes described above, there is no proxy and the APs and clients are unmodified, i.e., the APs take PHY rate adaptation and packet re-transmission decisions, while clients do not need to send out reception reports.

To evaluate *Medusa* we first look at the overall system performance in the multicast (Section 4.1) and the unicast (Section 4.2) cases. We then look at contribution of various *Medusa* components to the overall performance in Section 4.3. Specifically, we investigate benefits of rate adaptation in Section 4.3.1 and the performance benefits due to retransmissions in Section 4.3.2.

4.1 Overall performance (multicast)

We begin by evaluating the performance of the *Medusa* system in terms of its scalability for multicast traffic scenarios. We do so by — increasing number of clients and increasing video rates.

Scalability in the number of clients: We compare how different schemes can support HD video delivery to a large number of co-located WiFi clients. Figure 3 shows the performance of a highly loaded system with 25 clients (all receiving the same 20 Mbps video stream) at medium loss locations. We find that *Medusa* performs close to *UCAST-INDIV* (difference of 3-4 dB with 25 clients) with increasing client count, and is significantly superior to all other schemes.

Also, we find that there is a graceful degradation in *Medusa* performance when the number of clients is increased from 1 to 25. But even with 25 clients, the average PSNR value is around 37 while BDCST performance is around 27 (a 10 dB difference). The gradual degradation in performance of *BDCST* is because of the almost similar nature of errors experienced at each client(10-15% packet error).

The performance of *UCAST-SIMUL* suffers as 802.11a/g technology cannot support more than 2 streams with 20 Mbps rate(20 + 20 = 40 Mbps net load).

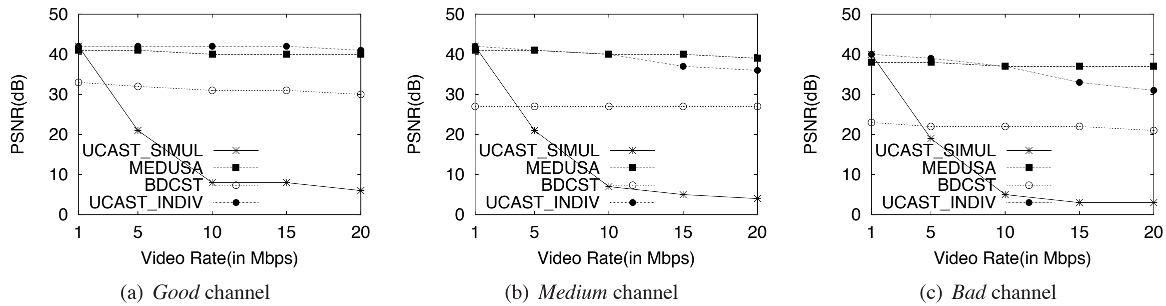
Scalability in video rate: We fix the number of clients to 10 and evaluate how the performance scales with increasing video rate — from 1 Mbps to 20 Mbps. We show the results separately for clients in good, medium and bad channel conditions in Figures 4(a), (b) and (c) respectively.

For good channel condition we observe that *UCAST-SIMUL* quickly degrades in performance with increase in video rates. Even at 5 Mbps (where the aggregate load is expected to be 5×10 Mbps = 50 Mbps), a lot of packet losses and buffer underflows occur. *BDCST* performs better and provides a more gradual performance degradation across the different rates. However, *Medusa* outperforms both and performs identical to *UCAST-INDIV*.

With worsening channel condition as shown in Figure 4(c) the performance of all schemes suffered. An interesting observation is that *the performance of* UCAST-INDIV *became worse than* Medusa as the traffic load (video rate) increased above 15 Mbps. This is due to "head-of-line" blocking in AP wireless NICs in the *UCAST-INDIV* case. Essentially, when various P- or B-packets are encountering losses, the AP spent significant effort in re-transmitting these packets, while more important I-packets waited behind. The lack of knowledge about the value of different packets, prevented the AP from devoting an appropriate amount of re-transmission effort for more important packets. *Medusa* explicitly addresses this problem and hence, led to improved performance.

4.1.1 Jitter variation of *Medusa*

We present the results for Jitter(measured as IPDV) in Figure 5. The experiment involved 10 users. Jitter increases with an increase in the number of clients, for all the schemes. However, the jitter of *Medusa* is significantly lower than both *BDCST* and *UCAST-SIMUL*. The jitter of *UCAST-SIMUL* increases exponentially with the number of clients. This can be attributed to the fact that with increasing number of clients the amount of data necessary to be transmitted becomes more than the network capacity. This results in a cascade of video packet drops in AP buffers and missing of deadlines. The jitter for *BDCST* also grows with the number of clients, as the number of candidates who can loose packets has also increased. Also, we note that the slope of increasing jit-



(a) *Good* channel (b) *Medium* channel (c) *Bad* channel

Figure 4: Average PSNR for 10 clients averaged over 20 runs as a function of the video rate under varying channel conditions.

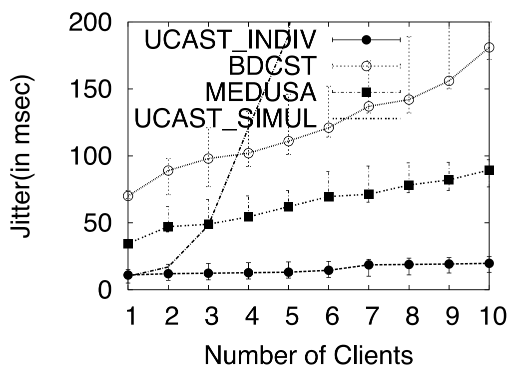


Figure 5: Average jitter experienced by 10 clients under medium channel conditions for 20 runs.

ter for *Medusa* is lower than that of *BDCST*, signifying a more gradual increase.

4.1.2 Induced network load

Apart from providing video quality commensurate with each user's channel quality, a good video multicast system should induce minimal additional network load. We compare the network load imparted by *BDCST*, *UCAST-INDIV* and *Medusa* in Table 2. We calculate the additional load placed in terms of the amount of airtime occupied by the packets (product of data-rate and packet size) which were transmitted using the different schemes. The results are normalized by the amount of airtime taken by *BDCST*. Table 2 shows that *Medusa* has an overhead of 4% for good channel conditions, which goes up to 30% under bad channel conditions. This overhead is to compensate for the 1-5% of errors that occur in good channel conditions. The channel induced losses go upto 15%-25% when the channel conditions are bad in our settings, forcing *Medusa* to inject an extra 30% traffic into the network. Hence, *Medusa* does not place unecessarily high traffic load over the network.

Channel Cond.	*BDCST*	*UCAST-INDIV*	*Medusa*
Good	1	10.12	1.04
Medium	1	10.26	1.1
Bad	1	11.40	1.3

Table 2: Airtime occupied by different schemes normalized to that of *BDCST* for 10 clients watching a 5 Mbps video, averaged over 20 runs, under varying channel conditions.

The above observation would seem to contradict with the fact that *Medusa* uses a conservative rate-adaptation mechanism which should significantly increase its network resource usage. However, we find that conservative rate-adaption while increasing the relative time occupied by individual packets also suffers less packet loss. Thus, keeping the overall network utilization low. We present further results in support of this statement in Section 4.3.

4.1.3 Interaction with other traffic

We investigate the performance of *Medusa* in presence of multiple uncorrelated traffic sources in Section 4.1.3. Since we do not introduce any new end-to-end congestion control mechanism in *Medusa* we do not present in depth results on the interaction of *Medusa* with TCP flows. In our experiments, introducing a *Medusa* flow without congestion control along with multiple TCP flows results in *Medusa* flow forcing the TCP flows to share only the residual bandwidth amongst themselves. We plan to implement a congestion controlled version of *Medusa* as part of our future work. We depict the impact of UDP flows on *Medusa* performance, in Figure 4.1.3.

We vary the number of background UDP flows, each at 4 Mbps, and compare the behavior of *Medusa* operating with a 10 Mbps video for 10 clients. The presence of multiple UDP streams causes a reduction in the quality of video seen at the clients for all schemes. However, *Medusa* outperforms *UCAST-INDIV* as the number of background flows is increased (around 7dB better for

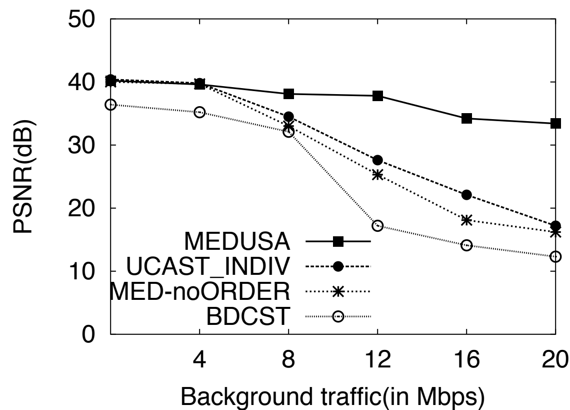


Figure 6: Average PSNR for 10 clients averaged over 20 runs in presence of background UDP flows under medium channel conditions. Video rate is 10 Mbps.

4 background flows). We find that these gains of *Medusa* are mainly due to our intelligent packet (re)-transmission ordering that mitigates the head-of-line blocking problem in *UCAST-INDIV*. We show this explicitly by also introducing a new scheme, called *Medusa*-noORDER, in which the packet re-ordering mechanism of *Medusa* is disabled. The performance of *Medusa*-noORDER is quite similar to that of *UCAST-INDIV*.

4.2 Overall performance (unicast)

To evaluate the performance of *Medusa* in serving multiple unicast video streams, we have increased the number of video flows from one to four in increments of one. We select the client randomly from a pool of 15 clients. Each experiment was run 20 times with a 5 Mbps video stream. There were other uncorrelated background flows (total of 5 Mbps) running during each experiment We plot the results of our observation in Figure 7. In unicast traffic settings, the gains in *Medusa* arrive from content-dependent rate selection, intelligent packet ordering and re-transmissions, as well as network coded re-transmissions. The broadcast advantage is available only for these network-coded re-transmissions, and not for original packets. With unicast video destined to 4 clients, the aggregate load is 20 Mbps, which is quite significant. Under good channel conditions, *Medusa* still delivers an average PSNR of 40 dB, which is very similar to *UCAST-INDIV* and is 9 dB greater than *UCAST-SIMUL*. This gain is even larger (18 dB) under bad channel conditions.

4.3 Micro-benchmarks of *Medusa* components

We now evaluate the effect of individual design choices on overall system performance. We look into performance of rate adaptation under diverse channel conditions in Section 4.3.1. The performance of network coded retransmissions is evaluated in Section 4.3.2 and the overall contribution of different components is summarized in Section 4.3.3.

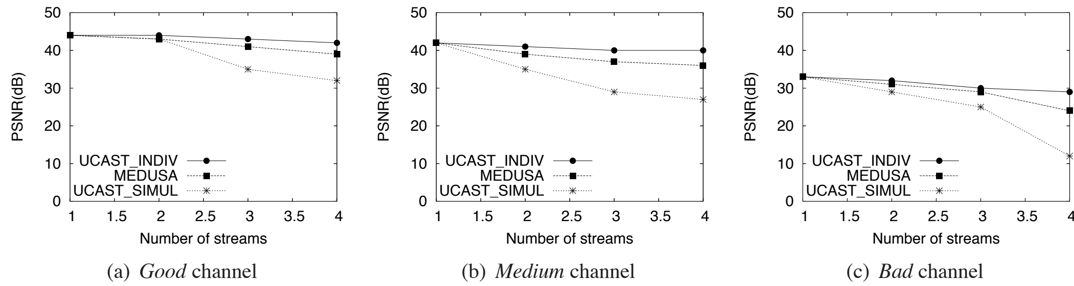
4.3.1 Rate adaptation in *Medusa*

An important aspect of *Medusa* is its ability to adapt the PHY rate based on channel conditions. We investigate the performance of these mechanisms next.

Impact of conservative base PHY rate adaptation: We look at the effects of using a conservative rate adaptation algorithm in *Medusa* on the overall system performance. We conduct experiments with a 5 Mbps video rate to 10 clients in good, medium and bad channel conditions. We ran *Medusa* with a conservative ($err_thresh = 0.02$) and an aggressive ($err_thresh = 0.18$) rate adaptation algorithm. Here, err_thresh signifies the maximum expected error rate which we are willing to tolerate for any PHY rate. We also ran the experiment with *UCAST-INDIV*. All the experiments we repeated for 20 runs. Figure 8(a, b) show the CDF of PHY rates assigned by different schemes under the good and the bad channel conditions. We observe, *Medusa*-conservative assigns lower PHY rates to packets than unicast, while the aggressive algorithm assigns data rates higher than the conservative scheme, but lower than the unicast scheme. To highlight the benefits of conservative rate adaptation, we plot the number of extra bytes transmitted, as a fraction of overall video size, and the PSNR of the resulting video under different channel conditions when using conservative, aggressive and unicast rate adaptation in Figure 8(c). For the *UCAST-INDIV* we plot the number of packets averaged by number of clients present. The following observations can be made from the plot,

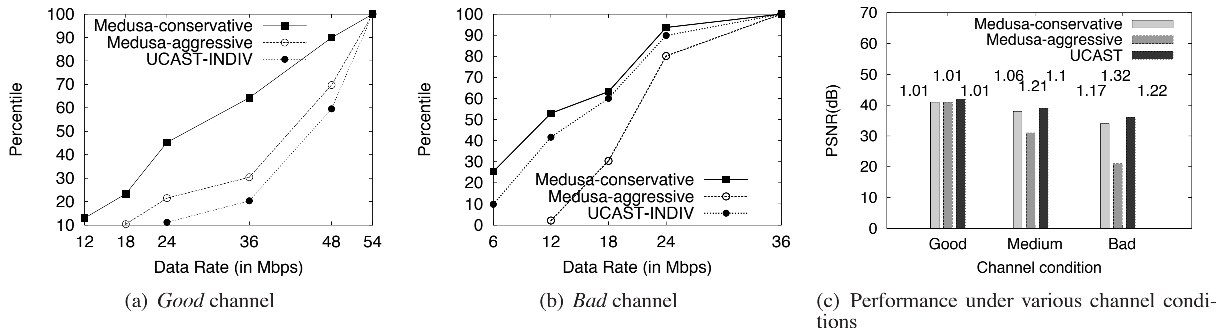
- Under good channel conditions, an aggressive as well as a conservative scheme would lead to similar number of packet losses(1%). Under such circumstances, all three schemes offer similar video quality and send similar amount of traffic over the network. From Figure 8(a), we find that around 80% (74%)of packets were transmitted at 24 Mbps or higher rate in *UCAST-INDIV*(*Medusa*-aggressive), in contrast to only 30% form *Medusa*-conservative. Hence, using an aggressive rate adaptation would had been beneficial in this case, as it would lead to network bandwidth conservation.

- For medium and bad loss environments, *Medusa*-conservative sends around 20% and 10% packets at 24 Mbps or higher. *UCAST-INDIV* sends about 40% and 11% packets at 24 Mbps of higher. In contrast, *Medusa*-aggressive sends about 60% and 20% of its packets at 24 Mbps or higher. This is because of the slowness of the feedback process which



(a) *Good* channel (b) *Medium* channel (c) *Bad* channel

Figure 7: Overall performance of *Medusa* for unicast-only media traffic. Upto 4 clients shown, each requesting a separate media stream with 5 Mbps video rate, under different channel conditions.



(a) *Good* channel (b) *Bad* channel (c) Performance under various channel conditions

Figure 8: CDF of packet rates assigned when transmitting 5 Mbps video to 10 clients in different channel conditions using different rate adaptation mechanisms. The bars in plot (c) shows performance of the schemes in terms of PSNR. The numbers in plot (c), on top of each bar, depict the normalized extra traffic in number of bytes sent by each scheme, relative to *BDCST*.

makes the *Medusa*-aggressive algorithm slow to react to changes in channel conditions. The performance of the *Medusa*-aggressive scheme suffers because of its inability to adapt quickly as shown in Figure 8(c). The number of packets transmitted by the conservative algorithm is around 15% less than *Medusa*-aggressive. This is expected, as the high threshold value ensures that we would make very few errors. The aggressive algorithm also leads to worse video quality(in PSNR) when the network resouces are scarce, precisely because of their inefficient network resource usage. Worsening channel conditions makes the difference in video quality about 6-12 dB(*Medusa*conservative and *UCAST-INDIV* have an advantage over *Medusa*-aggressive).

Thus, except under good channel conditions, keeping a conservative rate leads to better network resource utilization, while the quality is maximized in all conditions by adopting a conservative rate adaptation.

Impact of mobility on rate adaptation: To study the effect of mobility and its impact of adaptation mechanisms, we performed targeted mobility experiments, where we repeatedly moved one user between a high-loss and a low-loss location, while all the other clients stayed stationary at the low-loss location. The mobile client moved from the low loss to the high loss location (across a wall) quickly, stayed there for about 4 seconds, and returned. We show the adaptation performance of *Medusa* in comparison to *UCAST-INDIV* and *BDCST* in Figure 9. The *UCAST-INDIV* scheme running its MAC-layer rate adaptation technique adapts the fastest. *Medusa* with its intent of making rate adaptation decisions (of its PHY base rate) slowly, adapts somewhat slower. It takes *Medusa* about 0.4 seconds to adapt to the change in channel condition for the mobile user. This occurs in both cases —

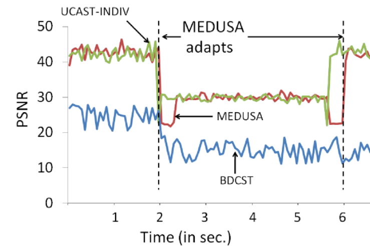


Figure 9: Adaptation of different scheme with targeted mobility, for video at 20 Mbps rate.

when the user moves away from the low loss location, and when it returns to the low loss location. This can be attributed to the higher layer reception reports and slower timescales in which they occur. However, once *Medusa* adapts, it provides the user with the same performance as the *UCAST-INDIV* in this case. The *BDCST* scheme has no adaptation mechanism and does not adapt when the user moves.

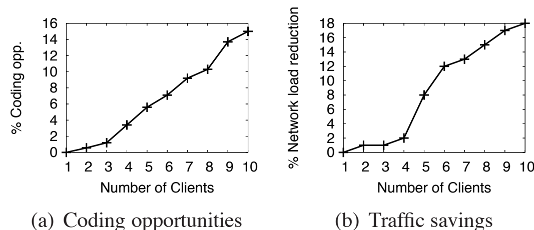
Impact of interference on rate adaptation: We next study the performance of these schemes under targeted interference from an external 802.11 source, that was a hidden terminal to the *Medusa* clients. Figure 10(a) shows the relative performance of *Medusa*, *UCAST-INDIV*, and *BDCST*, when the video rate was 5 Mbps. The interferer used UDP to download a large file starting at time 2 seconds. The performance impact of this interference is similar to that of mobility. *Medusa* performed similar to *UCAST-INDIV* and much superior to *BDCST*. However, it experienced a slight delay in adapting its rate when compared to *UCAST-INDIV.*

As shown in Figure 10(b), at a video rate of 20 Mbps, a similar effect happens with the hidden terminal interference. However, hidden terminal has a significantly greater interference impact and at this high video rate, the PSNR of both *Medusa* and *UCAST-INDIV* drops. Further, at 20 Mbps and with hidden terminal interference, the performance of *UCAST-INDIV* falls slightly below *Medusa*. Examining this performance of *Medusa* more closely, we see that at time 2.4 seconds, the inflate-deflate algorithm kicks in to help improve performance. The table in Figure 10(c) shows the number of I, P, and B frames that *Medusa* had to discard, inflate, deflate, and their channel occupancy time in the three phases (initial no interference, interference starts, and inflate-deflate starts).

4.3.2 Network coded re-transmissions

We evaluate the benefits of using network coded retransmissions with varying number of clients in the system (Figure 11). Panel (a) figure shows the percentage of all packet transmissions in each case that were actually network coded. As can be seen from the plot with increasing number of clients the number of network coding opportunities increases. Also, we would like to note that the computation overhead is never more than 1% of CPU time in any of our experiments. The actual performance gains from network coding can be seen in Figure 11(b) which indicates the reduction in airtime load that occurred due to network coding opportunities.

Finally, we evaluate the benefits of network coded retransmissions under varying channel conditions. We experiment with 5 clients receiving a 5 Mbps video under good, medium, and bad channel conditions respectively. We report the coding opportunities and the airtime reduction due to the network coded scheme in Table 3. The table shows that worsening channel condition leads to higher benefits from network coding. This is expected, as the number of packet losses increases as the channel condition becomes bad, this in turn leads to higher number of retransmissions and thus more coding opportunities.



(a) Coding opportunities (b) Traffic savings

Figure 11: Coding opportunities and percentage traffic reduction as a function of number of clients under medium channel condition with 5 Mbps video averaged over 20 runs.

Chnl. cond.	*Good*	*Medium*	*Bad*
% of coded packets	3.1	7.4	12.6
% Airtime load reduction	5	8	13

Table 3: Coding opportunities and normalized traffic injected as a function of channel condition for five clients with 5 Mbps video averaged over 20 runs.

We note that using network coding also leads in improving PSNR with increasing number of clients or worsening channel error conditions. We do not present the results for sake of brevity.

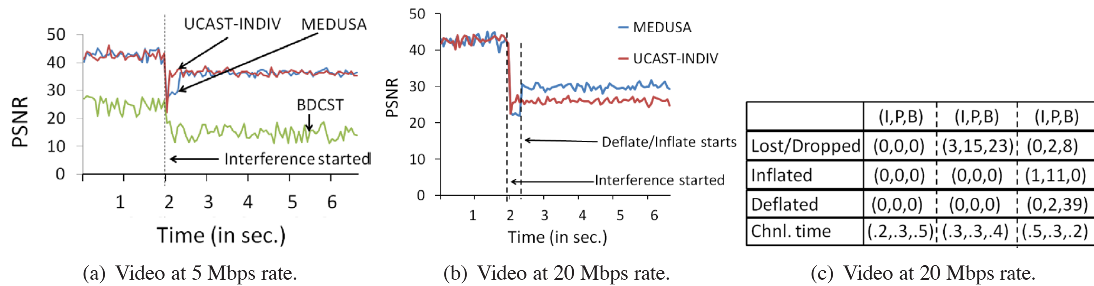
4.3.3 Component contribution

The *Medusa* system employs content aware rate adaptation, selective retransmissions and transmission (re)ordering to provide quality enhancements over broadcast based media delivery. Figure 12 shows the relative contribution of different design components in *Medusa*, over and above standard WiFi broadcast. In the low-loss and the high-loss environments various mechanisms in *Medusa* (re-transmissions, rate adaptations, ordering, rest – from integration of all the components). provides a nearly 9 and 10 dB improvement in PSNR over plain *BDCST*.

5 Related Work

There has been a significant amount of research in the area of video streaming over wireless networks, both in video and systems community (see [29] for a summary). We comment on the most related pieces in this section.

Dynamic transcoding is a standard technique for enhancing the quality of the streaming video. It involved



	(I,P,B)	(I,P,B)	(I,P,B)
Lost/Dropped	(0,0,0)	(3,15,23)	(0,2,8)
Inflated	(0,0,0)	(0,0,0)	(1,11,0)
Deflated	(0,0,0)	(0,0,0)	(0,2,39)
Chnl. time	(.2,.3,.5)	(.3,.3,.4)	(.5,.3,.2)

(a) Video at 5 Mbps rate. (b) Video at 20 Mbps rate. (c) Video at 20 Mbps rate.

Figure 10: Adaptation of different schemes with external hidden terminal interference with video at 5 and 20 Mbps rate. The table shows the number of I, P, and P frames that were discarded, inflated, deflated, and the channel occupancy time for *Medusa* in the three phases.

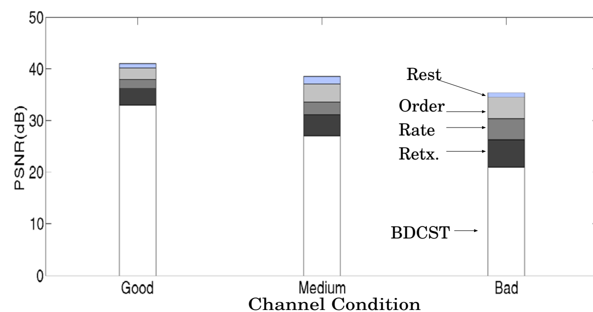


Figure 12: Performance breakdown between rate adaptation and retransmission components of *Medusa* system for 10 clients averaged over 10 runs under varying channel conditions.

estimating the bandwidth available in the medium and then change the video rate itself to ensure the best quality video that the channel can support is delivered to the receivers. Chou et.al. [9, 11, 12] in their seminal work propose a rate-distortion optimization technique to adjust the rate of the transmitted video based on channel quality. However, this body of work depends on the wireless hardware to pick the rate at which the video is to be transmitted. A second set of prior work dealing with identifying the optimal video rate as well as the amount of redundancy to be added to the video stream is represented by the [25, 26]. The authors formulate a complex optimization problem for the same and provide heuristic algorithms which show the performance benefits of the designed algorithms. Such FEC based mechanisms are orthogonal to the set of techniques used in our work. In *Medusa* we mostly leverage understanding from such prior work, and tailor our solutions to the needs of WiFi-based media delivery and specific issues therein.

Authors in [19] use a scalable video codec and optimally determine the amount of FEC required. This is a representative of a large body of literature in the area. This approach is, however, complementary to ours, as we focus on rate adaptation and re-transmission based techniques for WiFi broadcasts in video delivery systems.

In general wide-area network settings, the OxygenTV project [15, 16] has considered performing selective end-to-end re-transmissions of packets based on the video frame type, focusing more on unicast video delivery. They propose the SR-RTP protocol for the such selective retransmissions [16]. In contrast, our work explores various wireless link adaptation mechanisms that leverage packet content information.

In [31], authors present a measurement study different application-layer video streaming mechanisms in multi-hop wireless context. They do not explore interactions between the value of content to applications, and link adaptation mechanisms as we do in this work. In [27], authors present mechanisms to improve the quality of the video while operating in a lossy wireless environment. However they focus on low bit-rate video streams, while our solutions are stylized to deliver HD quality video in WiFi environments.

The authors of [28] present an end-to-end video rate control protocol for mobile media streaming on Internet paths involving wireless links. They implement the control functionalities in the receiver, which is charged with proving feedback to the server. The video server uses this information to change the video codecs used to match the available capacity of the end to end path. The proposed approach is complementary to ours, as we focus on adapting video delivery on the WiFi link, by making link adaptation decisions for WiFi transmitters.

A recent mechanism, SoftCast [17], uses the notion of compressed sensing to create equal priority video packets. This allows users to extract information proportional with their own channel quality. The core of this work focuses on the complementary aspect of compressed sensing. Furthermore, SoftCast also requires changes to the wireless radio hardware (and the PHY layer), while our system makes no changes to the current 802.11 standards.

Finally, DirCast [10] also design and implement a system for WiFi multicast. They advocate the use of pseudo-broadcasts in their system. However, the main difference between our *Medusa* approach and DirCast is that we propose a content-dependent PHY rate selection, re-transmissions, and packet order selection. This is an issue that is not considered by DirCast. DirCast focuses on some complementary problems for the multicast case only (e.g., intelligent client-AP association decisions, FECs, etc.), and is agnostic of value of packets to applications.

We believe that the main contribution of *Medusa* is in combining some understanding of packet contents with various WiFi link layer functions to improve the quality of media delivery. WiFi link layer decisions, until now, have been considered in a mostly content-agnostic manner. *Medusa* suggests an interesting design point for combining application-layer information in making decisions at the link layer.

Various other new techniques can be brought to improve performance of *Medusa* even further. In the future we therefore plan to investigate the use of other complementary but related mechanisms, such as application layer FEC for proactive error recovery, and a congestion control mechanism for co-existence with TCP flows.

6 Conclusions

Media delivery over wireless systems is a growing area of importance. We present the design and implementation of the *Medusa* system which allows efficient delivery of high quality media to one or more WiFi clients. The key contribution of this work is in recognizing that certain link layer functions, e.g., re-transmissions, PHY rate selection, packet transmission order, can be implemented better by having some knowledge about the value of packets to applications. In order to be minimally invasive to existing systems, we implement this function in a proxy. Our results indicate that our collection of techniques can facilitate HD video delivery of 20 Mbps to 25 clients while maintaining a good viewing quality.

7 Acknowledgements

We thank Sanjeev Mehrotra for initial discussions on the state-of-art in media streaming. We acknowledge Sateesh Addepalli for his comments and suggestions on our evaluation plan that helped improve the quality of this paper. We also acknowledge Sriram Subramanian for co-developing a preliminary version of this system from which we learned a number of important lessons. Finally, we thank our shepherd Srinivasan Seshan and other anonymous reviewers whose feedback helped bring the paper to its final form.

Sayandeep Sen and Suman Banerjee were supported in part by the US NSF through awards CNS-0916955, CNS-0855201, CNS-0751127, CNS-0627589, CNS-0627102, and CNS-0747177.

References

[1] Evalvid - a video quality evaluation tool-set. www.tkn.tu-berlin.de/research/evalvid/.
[2] Ffmpeg - digital audio converter. www.ffmpeg.org/.
[3] Mpeg-2 hd test patterns. www.w6rz.net/.
[4] Mpeg-4, moving picture experts group-4 standard. www.chiariglione.org/mpeg/standards/mpeg-4/mpeg-4.htm.
[5] Mplayer - the movie player. www.mplayerhq.hu.
[6] Streambox. www.streambox.com/products/hd/hd_9200_main.html.
[7] Vlc media player. www.videolan.org/vlc/.
[8] Bicket, J. Bit-rate selection in wireless networks. MIT Master's Thesis, 2005.
[9] Chakareski, J., and Chou, P. A. Radio edge: rate-distortion optimized proxy-driven streaming from the network edge. *IEEE/ACM Transactions on Networking* (2006).
[10] Chandra, R., Karanth, S., Moscibroda, T., Navda, V., Padhye, J., Ramjee, R., and Ravindranath, L. Dircast: A practical and efficient wi-fi multicast system. ICNP 2009.
[11] Chou, P., and Miao, Z. Rate-distortion optimized streaming of packetized media. *IEEE Transactions on Multimedia* (2006).
[12] Chou, P., Mohr, A., Wang, A., and Mehrotra, S. Fec and pseudo-arq for receiver-driven layered multicast of audio and video. *Proceeding of Data Compression Conference* (2000).
[13] David, S. A guide to data compression methods, 2002.
[14] Demichelis, C., and Chimento, P. Ip packet delay variation metric for ip performance metrics (ippm). RFC 3393, IETF.
[15] Feamster, N. Adaptive delivery of real-time streaming video. MIT M.Eng. Thesis, 2001.
[16] Feamster, N., and Balakrishnan, H. Packet loss recovery for streaming video. 12th International Packet Video Workshop, 2002.
[17] Jakubczak, S., Rabul, H., and Katabi, D. Softcast: One video to serve all wireless receivers. MIT-CSAIL-TR-2009-005, 2009.
[18] Katti, S., Rahul, H., Hu, W., Katabi, D., Médard, M., and Crowcroft, J. Xors in the air: practical wireless network coding. *IEEE/ACM Transaction Networking*.
[19] Majumdar, A., Sachs, D., Kozintsev, I., Ramchandran, K., and Yeung, M. Multicast and unicast real-time video streaming over wireless lans. *IEEE Transactions on Circuits and Systems for Video Technology,* (2002).
[20] Martinez-Rach, M., and et. al. Quality assessment metrics vs. PSNR under packet loss scenarios in MANET wireless networks.
[21] Murty, R., Wolman, A., Padhye, J., and Welsh, M. An architecture for extensible wireless lans. *HOTNETS-VII* (2008).
[22] Orlov, Z. Network-driven adaptive video streaming in wireless environments. *PIMRC* (2008).
[23] Rozner, E., Iyer, A. P., Mehta, Y., Qiu, L., and Jafry, M. ER: efficient retransmission scheme for wireless lans.
[24] Schulzrinne, H., Casner, S., Frederick, R., and Jacobson, V. RTP: A transport protocol for real-time applications. RFC 3350, IETF.
[25] Seferoglu, H., Altunbasak, Y., Gurbuz, O., and Ercetin, O. Rate distortion optimized joint arq-fec scheme for real-time wireless multimedia.
[26] Seferoglu, H., Gurbuz, O., Ercetin, O., and Altunbasak, Y. Rate-distortion based real-time wireless video streaming. *Image Communications* (2007).
[27] Tan, K., Ribier, R., and Liou, S.-P. Content-sensitive video streaming over low bitrate and lossy wireless network.
[28] Tan, K., Zhang, Q., and Zhu, W. An end-to-end rate control protocol for multimedia streaming in wired-cum-wireless environments. *ISCAS '03.*.
[29] Wang, Y., and Zhu, Q.-F. Error control and concealment for video communication: a review. *Proceedings of the IEEE* (1998).
[30] Wong, S. H. Y., Yang, H., Lu, S., and Bharghavan, V. Robust rate adaptation for 802.11 wireless networks.
[31] Xiaolin, C., Prasant, M., Sung-Ju, L., and Sujata, B. Performance evaluation of video streaming in multihop wireless mesh networks.

Maranello: Practical Partial Packet Recovery for 802.11

Bo Han*, Aaron Schulman*, Francesco Gringoli†, Neil Spring*, Bobby Bhattacharjee*
Lorenzo Nava†, Lusheng Ji‡, Seungjoon Lee‡, Robert Miller‡
* University of Maryland † University of Brescia ‡ AT&T Labs – Research

Abstract

Partial packet recovery protocols attempt to repair corrupted packets instead of retransmitting them in their entirety. Recent approaches have used physical layer confidence estimates or additional error detection codes embedded in each transmission to identify corrupt bits, or have applied forward error correction to repair without such explicit knowledge. In contrast to these approaches, our goal is a practical design that simultaneously: (a) requires no extra bits in correct packets, (b) reduces recovery latency, except in rare instances, (c) remains compatible with existing 802.11 devices by obeying timing and backoff standards, and (d) can be incrementally deployed on widely available access points and wireless cards.

In this paper, we design, implement, and evaluate Maranello, a novel partial packet recovery mechanism for 802.11. In Maranello, the receiver computes checksums over blocks in corrupt packets and bundles these checksums into a negative acknowledgment sent when the sender expects to receive an acknowledgment. The sender then retransmits only those blocks for which the checksum is incorrect, and repeats this partial retransmission until it receives an acknowledgment. Successful transmissions are not burdened by additional bits and the receiver needs not infer which bits were corrupted. We implemented Maranello using OpenFWWF (open source firmware for Broadcom wireless cards) and deployed it in a small testbed. We compare Maranello to alternative recovery protocols using a trace-driven simulation and to 802.11 using a live implementation under various channel conditions. To our knowledge, Maranello is the first partial packet recovery design to be implemented in commonly available firmware.

1 Introduction

Partial packet recovery approaches attempt to repair corrupt packets instead of retransmitting them. Packet recovery relies on the observation that packets with errors may have only a few, localized errors, or at least some salvageable, correct content. Various approaches have been proposed: some rely on physical layer information to identify likely corrupt symbols (related groups of bits) to be retransmitted [12], while others embed block checksums into oversized frames to allow the receiver to recognize partially correct transmissions [11]. Some avoid explicit knowledge and adaptively transmit forward error correction information that is likely to be sufficient to repair bit errors [14]. These approaches have found substantial potential in partial packet recovery, particularly when auto-rate selection mechanisms, which dynamically change the transmission rate to maximize throughput without too many errors, may choose too high a rate, thus creating errored packets to be recovered.

Motivated by the potential of these recent approaches, we set out to construct a partial packet recovery scheme using commonly available 802.11 hardware and evaluate it in live networks. The key challenge in working within 802.11's typical operation is timing, in particular, performing all acknowledgment-related computation within one short inter-frame space (SIFS) interval (10 μs for 802.11b/g or 16 μs for 802.11a). This requirement all but precludes bus transfers to the driver and complex processing on the network devices. To be deployable today, partial packet recovery must exploit features available to programmable *firmware*.

In this paper, we present *Maranello*, a block-based partial packet recovery approach implemented (primarily) in firmware for widely-available Broadcom cards. Maranello takes the following design decisions. We use *block-based recovery*, meaning that we identify incorrect blocks of consecutive bytes for retransmission, as opposed to aggregating by symbol or estimating bit error rate. We transmit independent *repair packets* that contain only the blocks being retransmitted, in contrast to other approaches that may bundle repair information with subsequent transmissions to save on medium acquisition time. Repair packets, by being shorter, are more likely to arrive successfully than full size retransmissions and take less time to transmit, improving performance over 802.11. Using immediate repair packets also limits the amount of buffering (of out of order, incomplete packets) required at the receiver side. We use the *Fletcher-32* checksum [5] to isolate errors to individual blocks; this checksum is sufficient to find all single bit errors, burst errors in a single 16-bit block, and two-bit errors separated by at most 16 bits [25]. Fletcher-32 is also efficient

enough to be computed block-by-block in software during frame reception. Finally, we exploit the deference stations give to acknowledgments of overheard packets: because stations sending acknowledgments have priority over the medium right after a transmission, there is time for a receiver to grab the medium and send prompt feedback about received blocks. Through these decisions, we construct a partial packet recovery scheme that (a) introduces no additional bits in the common case of successful transmissions, (b) decreases recovery time after failed transmissions, (c) is compatible with unmodified 802.11 devices, and (d) can be implemented on typical off-the-shelf hardware and deployed incrementally.

Our goal in constructing a practical partial packet recovery scheme was to permit evaluation both in simulation and on live networks. We apply two strategies. First, we construct a trace-driven simulation to evaluate the performance Maranello would have when run with various combinations of operating system, driver, and chipset, as well as the performance Maranello would have compared to idealized PPR [12] and ZipTx [14]. We study the retransmission behavior of 802.11 implementations so that we might simulate Maranello on each: performance improvement depends on how aggressively the existing firmware retransmits, in particular, whether it performs proper exponential backoff and how it reduces transmission rate. We survey retransmission rate fallback selection schemes and show that Maranello increases throughput regardless of retransmission rate fallback: if the rate chosen is too high, Maranello may increase the delivery probability with a short repair packet [8]; if too low, Maranello decreases the time to transmit relative to retransmission.

Our implementation permits us to evaluate Maranello in terms of delivered throughput and latency in realistic settings. We compare the link throughput of Maranello and that of the original 802.11 in three different environments: an industrial research lab, a home, and a campus office building. We show that Maranello can significantly improve the delivered link throughput. We also verify that, even in the presence of bit corruption, Maranello can maintain or reduce the link latency, in terms of the time to deliver an individual packet and receive an acknowledgment. We also deploy Maranello on programmable access points running OpenWRT to ensure scalability and compatibility by associating both Maranello-enabled and unmodified 802.11 devices. Surprisingly, we find that ACK frames can be modified to report the feedback information of received blocks, without causing errors on coexisting unmodified 802.11 devices.

In the following section, we present an overview of prior wireless error recovery mechanisms including partial packet recovery schemes and those that rely on wireless communication diversity. In Section 3, we present the high-level design of Maranello, show how wireless errors cluster enough to support block-based recovery, and justify the choice of Fletcher-32. In Section 4, we evaluate these design choices in simulation, showing the potential throughput gains by interpreting detailed packet traces. In Section 5, we implement Maranello using the OpenFWWF firmware and a slightly modified driver within the Linux kernel. Section 6 presents performance comparisons collected in our testbeds using this implementation. We offer a discussion in Section 7 and conclude in Section 8.

2 Related Work

In this section, we classify various wireless error recovery protocols. Table 1 summarizes wireless error recovery protocols. We categorize these protocols along two dimensions: the main repair techniques that they employ and the features they provide. The main repair techniques include:

Block checksum (Section 2.1) When transmissions fail, receivers can aid recovery by sending feedback about corrupted blocks based on the per-block checksums transmitted with data packets. Seda [6] and FRJ [11] are protocols in this category.

Forward error correction (Section 2.2) Protocols like ZipTx [14] avoid explicit knowledge about where the error bits are and adaptively transmit error correction bits that are likely to be sufficient to repair corrupted packets.

PHY layer hints (Section 2.3) The PHY layer of GNU Radio systems can provide the confidence of each symbol's correctness. PPR [12] and SOFT [27] benefit from this information to identify corrupt bits without extra error detection codes.

Wireless communication diversity (Section 2.4) Wireless packet losses are path and location dependent. A packet corrupted at its destination may be correctly received by other radios, due to the broadcast nature and diversity of wireless communication. Several protocols exploit this diversity to perform error recovery, such as MRD [20], SPaC [4], and PRO [16].

These error recovery protocols provide the following features:

No extra bits for correct packets Most of the protocols introduce no additional bits for successful transmissions, except Seda and FRJ, which transmit block/segment checksums with all packets, and ZipTx, which sends pilot bits in each transmission.

Technique	Protocol	No extra bits for correct packets	Maintain link latency	Compatible with 802.11	Incremental deployment	Partial Packet Recovery
Checksum	**Maranello**	✓	✓	✓	✓	✓
	Seda [6]			N/A	✓	✓
	FRJ [11]				✓	✓
FEC	ZipTx [14]				✓	✓
PHY layer hints	PPR [12]	✓	✓	N/A		✓
	SOFT [27]	✓	✓	N/A		
Diversity	MRD [20]	✓	✓		✓	
	SPaC [4]	✓	N/A	N/A	✓	
	PRO [16]	✓	✓	✓	✓	

Table 1: Desired behavior and functionality of wireless error recovery protocols

Reduce recovery latency Seda, FRJ, and ZipTx may increase the recovery latency by aggregating feedback for a group of corrupted packets. MRD and SOFT may also increase the recovery latency for the packets that cannot be repaired by frame combining.

Compatible with 802.11 Among the protocols designed for 802.11 wireless networks, MRD, FRJ, and ZipTx disable the retransmission protocol at the MAC layer and thus do not interoperate with native 802.11.

Incremental deployment Most of the protocols are implemented using commercial hardware, either 802.11 cards or MICA motes, and thus can be incrementally deployed on widely available wireless devices. In contrast, PPR and SOFT use physical layer information provided by GNU Radio systems.

Partial packet recovery Protocols like PRO and SOFT always retransmit the entire packet when the original cannot be recovered.

Table 1 shows that none of these protocols achieve all these features simultaneously.

2.1 Block Checksum

Acknowledgment frames can be extended to include feedback to help error recovery protocols. Seda [6] is a recovery mechanism designed for data streaming in wireless sensor networks. In Seda, a sender divides each packet into blocks and encodes each block with a one-byte sequence number and a (one-byte) CRC-8 for error detection. A receiver, after receiving several packets, will test the block-level CRC-8's for packets that fail the CRC-32 (if any) and request retransmission of those blocks.

FRJ [11] uses jumbo frames to increase wireless link capacity. Each jumbo frame comprises 30 segments and each segment has its own CRC checksum. The receivers can check these segment checksums to perform partial retransmissions when the segments are corrupted. FRJ uses both MAC-layer ACKs and its own ACKs. FRJ sends its own ACKs after 100 ms or 64 received frames.

Unlike Seda and FRJ, Maranello introduces no extra bits for correctly received frames and performs retransmission immediately after corrupted frames are detected.

2.2 Forward Error Correction

Forward error correction codes are beneficial to error recovery because they do not require explicit information about error locations. ZipTx [14] uses a two-round forward error correction mechanism to repair corrupted packets. In the first round, the transmitter sends a small number of Reed-Solomon bits for a corrupted packet, based on the feedback provided by the receiver. If the receiver still cannot recover the corrupted packets using these parity bits, the transmitter sends more parity bits in the second round. If both rounds fail, the receiver requests a retransmission of the whole packet. To reduce the number of feedback frames, ZipTx receivers accumulate feedback information to be transmitted after receiving eight packets or after a timeout.

Although ZipTx increases throughput, it may also increase recovery latency. This is because it disables MAC layer retransmission and generates its own ACKs for a group of packets in the driver. As a result, the delay for the recovered packets may be significantly higher than that of the retransmitted native 802.11 packets. Maranello repairs corrupted packets immediately after transmission fails and thus can reduce recovery latency.

2.3 PHY Layer Hints

Error recovery protocols can benefit from physical layer information beyond the best guess at the received symbol, although most commercial 802.11 cards do not expose such extra information. PPR [12] requests retransmissions of only those symbols that are likely corrupted. PPR also provides a compact encoding of the ranges of bits requested for retransmission and replicates the preamble to a "postamble" so that receivers may recover

correct bits at the end of packets that lack a good preamble. PPR was implemented and evaluated on an 802.15.4 (ZigBee) protocol stack.

Driven by per-bit confidence from the PHY Layer, SOFT [27] combines several received versions of a corrupted frame to produce a correct frame. To repair packets sent to an AP, several APs share bit confidence over a wired link. To repair packets sent to a client, the client combines per-bit confidence from a corrupted transmission and one or more retransmissions.

Due to performance limitations of software radio platforms, these protocols are evaluated only at low bit rates. In contrast, Maranello is implemented using readily available commercial 802.11 hardware, and thus it can be immediately realized at speed and deployed. We also show that Maranello provides increased performance even with the encodings used for high bit rates.

2.4 Wireless Communication Diversity

Correcting errors with wireless diversity complements Maranello's packet repair. Diversity approaches attempt to correct packets by observing different copies of the same packet, either as received at different stations or as received in (corrupt) retransmissions. When failure happens, MRD [20] combines many received versions of a given packet at different APs, which may have error bits at different locations, to recreate the original packet. If the original packet cannot be recovered through frame combining, a retransmission protocol, called Request For Acknowledgment (RFA), is proposed to retransmit the whole packet. SPaC [4] exploits the spatial diversity of multihop wireless sensor networks to combine several corrupted receptions of a packet at its destination. These corrupted receptions may be retransmitted by different neighboring nodes to repair the original transmission. PRO [16] is an opportunistic retransmission protocol for 802.11 wireless LANs that allows overhearing relay nodes to retransmit on behalf of the source node after they know that a transmission failed.

Other protocols can benefit from wireless communication diversity, but these are typically evaluated only by theoretical analysis or simulation study. For example, MRQ [24] keeps all the erroneous receptions of a given packet and recovers the original packet by combining these receptions. Like PRO, HARBINGER [28] improves the performance of Hybrid ARQ, by exploiting retransmitted packets from relays that overhear the communication. The approach of Choi et al. [3] uses the error correction bits transmitted in data packets to recover corrupted blocks. It retrieves uncorrected blocks from later retransmissions of the packets and combines them with previous blocks to recover the original packets.

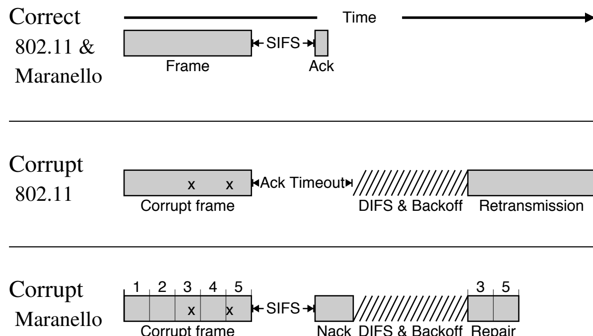


Figure 1: Maranello reacts to packet corruption by sending a NACK when the sender awaits an ACK. The time to repair should decrease relative to retransmission. (Diagram not to scale.)

3 Maranello Design

In this section, we present an overview of Maranello, describe how it achieves the key design goals of a practical partial packet recovery scheme, and justify the choices of block-based recovery and the Fletcher-32 checksum computation. We analyze this design in isolation in the following section (4) before presenting implementation details (Section 5) and evaluating the implementation on real hardware.

3.1 Overview

Figure 1 presents an overview of the Maranello protocol, compared to 802.11. When a Maranello-supporting device receives a frame with errors, it divides the frame into 64-byte blocks (the last block may be smaller) and computes a separate checksum for each block. Then it replies to the transmitter with a NACK that includes these checksums. It saves the corrupted original packet in a buffer, waiting for the sender to transmit correct blocks. This negative acknowledgment is sent when the transmitter expects to receive a positive acknowledgment. A Maranello-supporting transmitter will then match the receiver-supplied checksums to those of the original transmission and send a repair packet with only those blocks of the original transmission that were corrupted. Once the repair packet is received correctly, the receiver sends a normal 802.11 ACK.

Devices that do not support Maranello interoperate easily. Unmodified senders will treat the negative acknowledgment as garbage and retransmit as normal. Unmodified receivers will fail to transmit a Maranello NACK, and cause a Maranello sender to retransmit after timeout.

At very low transmission rate, the NACK for a large packet may be longer than other stations expect to defer to the acknowledgment (i.e., it may extend beyond the

Network Allocation Vector); if it does, we rely on carrier sense to inhibit collisions with the end of the NACK.

The cases when a Maranello-specific packet are lost are straightforward. If a NACK is lost, the transmitter will retransmit the packet as in 802.11. If this retransmission has errors, the receiver will send another NACK. If a repair packet is lost or received with errors, the receiver will transmit nothing. One could alter the protocol to send an abridged NACK to recover correct blocks from errored repair packets, but we expect minimal gain from the added complexity.

3.2 Design Goals

Maranello is a practical partial packet recovery design with four primary goals, described below.

Require no extra bits in correct packets Maranello embraces systems design principles of optimizing the common case, successful transmission, and doing no harm (not increase the size or delay of retransmissions). No additional error checking information, beyond the existing CRC-32, is added to normal packets.

Reduce recovery latency Maranello ensures that recovery latency is smaller than retransmission time by using the time reserved for positive acknowledgments to, in the event a positive acknowledgment is not warranted, send negative acknowledgments.

(In the unlikely event that the entire packet is corrupt, the longer NACK may require more time than an ACK and the retransmission of entire packet may not be avoided, leading to an overall increase in retransmission time.)

Compatibility with existing 802.11 802.11 is widely deployed, cheap, and useful. To extend it requires obedience to key inter-frame spacing and backoff requirements. The receiver must be able to construct and send a NACK before the transmitter decides to retransmit the entire packet, ideally immediately after the SIFS (short inter-frame space) interval when the transmitter expects an ACK. That is, the implementation must support extremely quick computation of block checksums in order to respond to the sender. At the same time, a Maranello sender cannot send repair packets any more quickly than 802.11 sends retransmissions: collisions are a potential cause of transmission error and must be addressed by proper exponential backoff. These two features are necessary for coexistence with 802.11 networks.

Incremental deployability on existing hardware Wireless networks are dynamic: Maranello should not require negotiation or, worse, ubiquitous deployment within a service area. By transmitting Maranello messages such that unmodified 802.11 devices are not confused, Maranello can coexist. In effect, the Maranello NACK is a negotiation; a Maranello station may infer that the receiver does not support Maranello if no NACKs are sent. (Reserved bits in the capability-information field of beacon and association-request frames are also available; it is possible to negotiate protocol features when necessary.) Further, by implementing Maranello in the firmware of existing wireless cards, this partial packet recovery protocol can be deployed today for users just by updating the firmware.

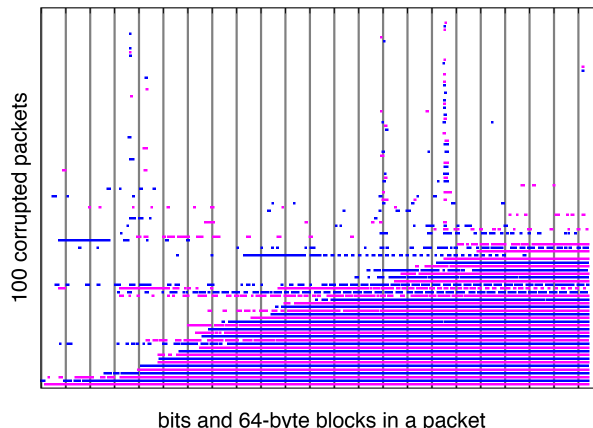


Figure 2: Shaded areas indicate bit errors. Within-packet (horizontal) correlations are likely due to interference or loss of clock synchronization; across-packet (vertical) correlations may be caused by subcarrier fading.

3.3 Block-Based Recovery

Broadly speaking, a partial packet recovery approach can use various means for receivers to solicit retransmission of parts of the packet and various means for transmitters to correct those errors. Maranello sends negative acknowledgments with checksums over blocks; transmitters determine which blocks must be retransmitted and send repair packets in place of retransmissions. (Alternate approaches may report abstract bit error estimates, request retransmission of individual symbols, or piggyback repair on subsequent transmissions, as described in Section 2.)

Block-based recovery, however, relies on a key assumption: that errors are clustered within a packet. In Figures 2 and 3, we present two views of error clustering. Figure 2 shows the positions of bit errors in 100 packets chosen at random from the errored packets in a larger trace of packets. For packets with few bit errors, those errors are constrained within 64-byte blocks. For packets with many bit errors, those errors are similarly often bound within consecutive 64-byte blocks.

Figure 3 plots 17,961 packets by the number of 64-byte blocks that would be needed to repair errors. The x-axis represents the fraction of corrupt packets: each packet occupies the same horizontal space along the axis,

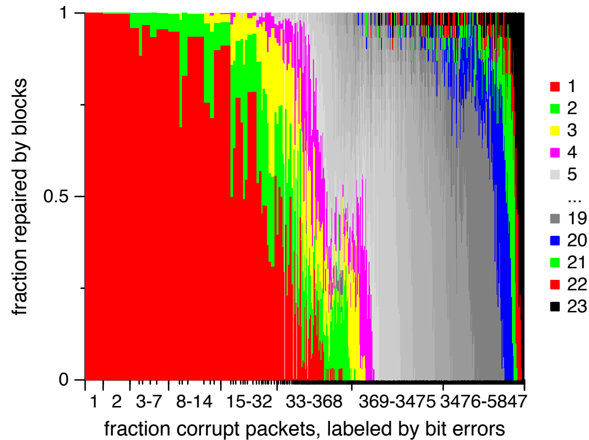


Figure 3: 64-bit blocks required to repair corrupt packets in a trace. Most packets having bit errors have few corrupt blocks; even those with many bit errors typically have a few correct blocks.

sorted in ascending order of the number of bit errors observed in that packet. A stacked bar graph extends above, showing the fraction of those packets required by different numbers of blocks. At the left side of the graph, the dominant color represents the single block's ability to repair all 1-bit errors (of course), 99.7% of two-bit errors, 96% of three-bit errors, etc. This is in contrast to a random bit-error model in which two bit errors in a 1500-byte packet would have only a 4% chance of corrupting only one 64-byte block. At the right end of the graph, relatively few packets require complete retransmission. (This graph may underestimate the number of irreparable transmissions; those that the hardware cannot receive at all would not appear.)

3.4 Fletcher-32

The block checksums a receiver puts into a NACK must be completely computed before the SIFS expires. One approach might be to reprogram the hardware-accelerated CRC-32 engine used by the device to compute whole-packet CRCs. Unfortunately, this engine does not appear to be programmable. Instead, we compute a different checksum, the Fletcher-32 [5] which is more efficiently computed on the wireless card's microprocessor. Historically, the IETF considered Fletcher checksums as an alternative for TCP checksums [30].

To verify the effectiveness of Fletcher-32 to detect bit errors, we perform the following trace-driven simulation. We take the 99,118 corrupted frames from a packet trace, and identify error bit positions in each frame. Then, we apply the error patterns to randomly generated packet contents to construct 9,911,800 errored packets. Finally, we apply CRC-32 and Fletcher-32 to detect corrupted blocks with 64-byte size. All the corrupted blocks can be detected by both CRC-32 and Fletcher-32.

Even with the efficient Fletcher-32 checksum, the microprocessor is still not powerful enough to compute each of the block checksums during the SIFS interval: A single checksum for a 64-byte block can take up to 4 μs. To solve this problem, we exploit an interesting feature of the chipset. The microprocessor, in fact, is idle during the reception of a frame! Instead of allowing it to sleep until the packet is completely received, we modify the firmware to copy partially received packets and begin computation of block checksums during reception of the next block. This approach leaves enough time at the end of a corrupted frame to compute the last checksum (if needed) and to build the NACK.

4 Simulation

Before we describe and evaluate the implementation, we evaluate the design of Maranello in simulation. Maranello's gains depend on the specified, but not always followed, 802.11 backoff and the unspecified retransmission rate fallback behavior implemented in 802.11 drivers and chipsets. We want to see if Maranello improves throughput for cards (we consider both the manufacturer's driver and chipset) that behave unlike Broadcom's, which we implemented Maranello on.

Each card implements a different suite of error control algorithms, including auto-rate selection, retransmission rate fallback, and backoff. 802.11's backoff behavior is defined in the specification, however our observations and those of Bianchi et al. [2] indicate that there are many different interpretations of 802.11 backoff. Although the 802.11 specification dictates backoff behavior, it leaves implementors to decide on auto-rate selection and retransmission rate fallback. 802.11 does not contain definitions for these algorithms because no algorithm will work in all wireless environments. For example an optimistic auto-rate selection may yield higher throughput on some links, but may also result in many errors on others. Our simulated results indicate Maranello can help increase the throughput from optimistic rate selection.

4.1 Maranello Increases Throughput for Popular 802.11 Cards

To characterize a variety of 802.11 backoff and retransmission rate fallback policies, we observed the retransmissions sent by three popular 802.11 cards. We ran the cards on Windows XP to observe the behavior of the most common driver. To analyze many instances of the card retransmitting its maximum number of retransmissions, we prevented the receiver from sending any acknowledgments. For each card, Figure 4 depicts the me-

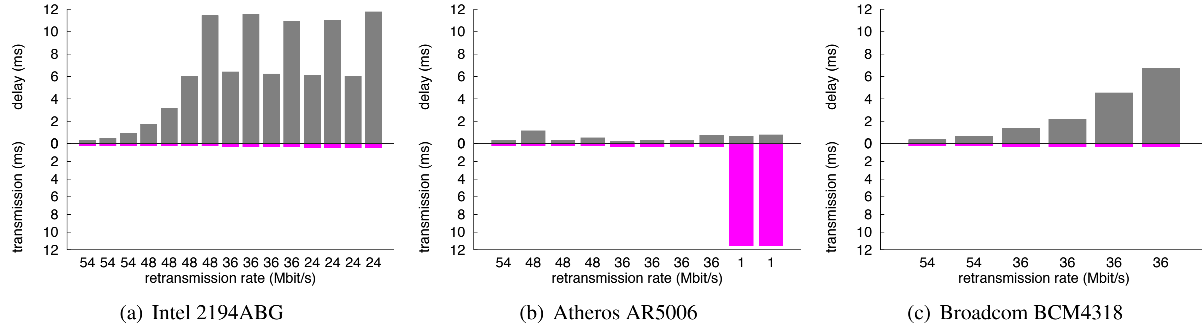


Figure 4: Popular 802.11 cards exhibit different exponential backoff behavior (top) and retransmission rate fallback (x labels show the rate, bottom bar shows transmission duration).

dian inter-retransmission delay and time to transmit for the observed retransmission rate fallback. For backoff, some cards appear to follow 802.11: Intel and Broadcom's median interval between retransmissions doubles for each retransmission. We did not observe Atheros doubling the backoff window after failed retransmissions.

Retransmission rate fallback also varies between cards. Each card appears to attempt a different number of retransmission rates (Intel 4, Atheros 4, Broadcom 2). Atheros does not experience much loss because the card will eventually attempt to retransmit a packet at the lowest possible rate defined in 802.11. Maranello helps Atheros because it will increase the probability that transmission is successful in the first few retransmissions, eliminating or at least reducing the size of retransmissions sent at the lowest bit rate. Intel retransmits at optimistic rates so it may need to retransmit more times than a card that quickly lowers the retransmission rate. For Intel, Maranello will help because it increases the probability of receiving a retransmission correctly, rewarding optimistic retransmission rate selection.

4.2 Trace-Driven Simulation

A trace-driven simulation of Maranello indicates that successfully retransmitting earlier increases throughput for several interpretations of 802.11. The simulator operates on a trace of packets with known payloads. Knowledge of the payload provides several desirable properties: (1) The simulator can determine the number of corrupted blocks in a packet. (2) The simulator can determine if the repair blocks fit inside a contiguous region of correct bits at the beginning of a (potentially corrupted) retransmission packet. (3) Resulting from (2) the simulator can subtract excess retransmissions seen after a successful repair. Table 2 shows the speedup obtained from simulating Maranello for the three popular cards. Intel appears to achieve significant gain because Maranello mitigates the errors caused by retransmitting at an optimistic rate, avoiding long, although standard, backoff times.

card	avg throughput	avg speedup	avg ERR	avg rate
Atheros	8.64	1.05	0.03	15.85
Broadcom	11.92	1.05	0.05	40.98
Intel	8.14	1.17	0.06	33.09

Table 2: In simulation Maranello increases throughput for the Intel chipset by correcting errors caused by optimistic behavior. ERR is the 64-byte block error rate.

4.3 Repair Size

Compared to other partial packet recovery protocols, Maranello does not need to send significantly larger repair packets. We simulated each of the repair protocols (Figure 5) with traces of data packets sent from a Broadcom card. To vary the bit error rate of these traces, we changed the distance between the sender and the receiver. The symbol size (1–216 bits) for symbol based repair (PPR) corresponds to the packet bit rate. We simulated an ideal version of ZipTx that assumes the indexes of corrupted bits are known, so it can pick the smallest number of redundancy bytes for the repair.

To repair corrupted bits, all of the repair protocols must send significantly more repair bits. For traces with a low BER, Maranello requires marginally more bits than the other repair protocols. ZipTx is able to retransmit so few bits because Reed Solomon works well when there are few bit errors. For corrupted packets with a high average BER, PPR's symbol-based repair needs to transmit the least number of bits to repair the packets. However, symbol based repair requires additional hardware to measure the confidence of symbols. Although Maranello needs more bits than symbol based repair, it requires fewer bits than ZipTx. If the packet contains errors clustered in one block, ZipTx's Reed Solomon will waste many repair bits for correct blocks because ZipTx chooses its coding rate based on the BER of the most corrupted block.

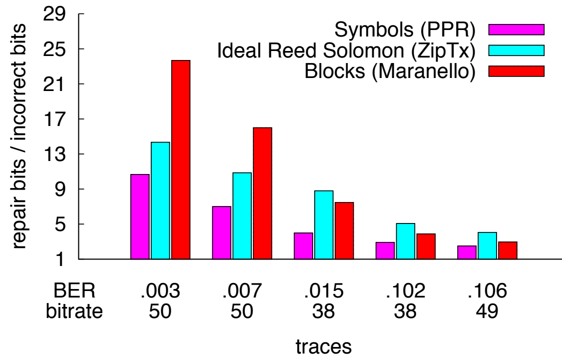


Figure 5: To repair corrupted packets with a high average BER, Maranello uses fewer repair bits than ZipTx. For low-average-BER corrupted packets, Maranello uses more repair bits than the other techniques. The bitrate shown is the average rate chosen by minstrel.

5 Implementation

We implement Maranello using OpenFWWF [7] open firmware and b43 Linux device driver [1] for Broadcom chipsets. In the following, we first discuss why several other potential platforms are not suitable for Maranello. We then present the implementation details of Maranello.

5.1 Why Other Platforms Are Unsuitable

To use the airtime reserved for ACK frames, receivers must construct and send NACK frames within SIFS, which is the defined inter-frame space between data packets and ACK frames [9]. Commercial 802.11 wireless NICs implement this time-critical operation in firmware or hardware.

5.1.1 Driver space of 802.11 wireless NICs

Recently, several wireless research platforms, such as FlexMAC [15] and SoftMAC [21], have been proposed to develop new MAC protocols. They are extensions of the MadWifi driver [18] for Atheros chipsets which runs in Linux kernel space. To determine how fast an implementation in driver space can send back NACK frames for corrupted frames, we perform the following experiment. When the test receiver gets a corrupted packet, it copies the first 100 bytes directly into a NACK frame, and sends it out immediately without performing backoff and using SIFS. From packets traced by a monitor node, we found that the minimum gap between the data packets and NACK frames is higher than 70 μs. This delay is mainly caused by bus transfer delay and interrupt latency and is consistent with the measurement results in Lu et al. [15]. This high latency makes the driver space unsuitable for the implementation of Maranello. Jitter due to DMA transfers makes timing too variable.

5.1.2 GNU Radio

GNU Radio platforms are slow and expensive. However, due to their flexibility, they have attracted increasing attention from the wireless research community and there are 802.11 implementations for them [22, 26]. In GNU Radio, the wireless signal is decoded at the host machine and the delay, depending on the length of the packets, is usually higher than 1000 μs [22]. The decoder could be put into the FPGA (Field-Programmable Gate Array) on the Universal Software Radio Peripheral (USRP), but the FPGA is much slower than the digital signal processor on the wireless NICs. Moreover, another challenge is to generate NACK frames for corrupted packets within SIFS, which is difficult to implement on these platforms.

5.2 Maranello Implementation

We first briefly introduce OpenFWWF and review the architecture of wireless device drivers in Linux kernel. Then we present the implementation of Maranello, focusing on NACK generation and repair packet construction, which are time-critical operations implemented in the firmware. Finally, we describe other operations implemented in the Linux driver.

5.2.1 Background

A microprocessor executes a typically proprietary microcode (firmware), written in assembly language, that handles various operations on wireless cards. OpenFWWF [7] attempts to replace the proprietary firmware with an open source firmware for Broadcom chipsets. It can support almost all the 802.11 primitives in the 2.4GHz frequency band. By changing the standard code path, it is possible to implement from scratch a completely different channel access mechanism, subject to a few basic hardware constraints, such as the PHY layer carrier sensing, the CCK and OFDM modulation schemes.

To better understand how the Maranello implementation works, we briefly review in the following the basic building blocks that equip the Broadcom chipset. The internal microprocessor drives the data exchange between different blocks using two main paths: transmit (TX) and receive (RX). The firmware is built as a main loop that reacts on external conditions such as a new frame's arrival from the air, a channel free indicator, and (programmable) timer expiration. The basic building blocks include:

TX and RX FIFO queues – The microprocessor pulls frames from the TX queue and moves them into the serializer when a transmission opportunity comes. On the opposite path, it moves a received frame from a buffer into the RX queue and raises an IRQ so that the host kernel can retrieve the frame.

Internal shared memory (SHM) – The microprocessor maintains several state variables which can be monitored or even changed by the host kernel.

Template RAM – The microprocessor can compose an arbitrary frame in this memory and transmit the resulting packet as if it came from the TX FIFO.

Internal registers and external conditions (EC) – The microprocessor sets these hardware registers in response to changes in the EC to program the radio interface and set up timers.

The current Linux kernel uses mac80211 [17] for device driver development. mac80211 is an abstraction layer that bridges between the kernel's networking stack and almost all the low-level wireless device drivers. For example, the rate control algorithms are usually implemented in mac80211 and shared by all the drivers. These drivers then act as stage-two bridge since all the 802.11 low level operations, such as retransmissions, acknowledgments, and virtual carrier sense, must be performed by either firmware or hardware, due to hard timing constraints that can not be met by a host-controlled approach.

5.2.2 NACK generation

As we mentioned in Section 3, to compose the NACK, the receiver computes block checksums for corrupted frames in the firmware. Due to hardware limitations, the Maranello block size should be a multiple of 32 bytes. We use 64 bytes as the block size. Longer blocks increase computation efficiency and shorten NACKs, while shorter blocks are parsimonious with repair bytes. In our experience, the 64-byte block represents a good compromise at typical rates, though we discuss possibilities for dynamic adjustment in Section 7.

For some transmission rates, a Maranello NACK uses more airtime than a MAC ACK, which may cause problems in the presence of hidden terminals. The size of an ACK frame is only 14 bytes. A Maranello NACK frame, based on 64-byte blocks, is at most 96 bytes longer than an ACK frame (4-byte checksum for each block, 24 blocks maximum). For a Maranello link, a hidden terminal of the receiver may hear from the transmitter the network allocation vector (NAV) and the earliest time it can start its own transmission is DIFS, 50 μs for 802.11b/g, after the end of NAV (suppose its backoff time is 0). There will be no collision when NACK's bit rate is higher than 12 Mbps. Otherwise, a transmission from the hidden terminal may collide with our NACK frames, which causes the retransmission of the whole packet. Preliminary experiments on a hidden terminal topology, indicate that even in this scenario, enabling Maranello can increase overall throughput.

5.2.3 Repair packet construction

Maranello transmitters must handle both ACK and NACK frames.

- Like 802.11, after a transmitter sends an original data packet or a recovery packet, it will set up an ACK timer.
- If the transmitter gets an ACK frame from the receiver, it will release the resource allocated for the original or recovery packets.
- If the transmitter gets a NACK frame from the receiver, it divides the original packets into blocks, computes the checksums for these blocks, and only retransmits the blocks whose checksums do not match those in the NACK. In practice, the block checksums are precomputed in the driver on the host processor.
- After the transmitter's ACK timer expires, and it does not receive a frame, but it previously attempted to repair the packet, it retransmits the repair packet. Otherwise it retransmits the whole packet

After a transmitter gets a NACK, it compares the received block checksums with the locally computed checksums and decides which block to retransmit inside a repair packet. We always retransmit the first block of a packet, which contains the important headers of various layers. For a repair packet, we reuse the 8-byte LLC header, only for data frames, by (1) changing the first byte to distinguish repair packets from other packets; (2) using the following 3 bytes as a bitmap of retransmitted blocks; and (3) appending an extra checksum (CRC-32 or Fletcher-32) in the last four bytes. The receiver uses this checksum, as an extra measure of safety, to verify that the recovered packet is correct.

Maranello uses the same 802.11 retry limit; each repair packet will increase the retry counter by one. Also before transmitting repair packets, it doubles the contention window.

5.2.4 Driver functionality

We implement non-time-critical operations in the driver, including the pre-computation of block checksums at the transmitter, and the reconstruction of frames at the receiver. We compute the block checksums for data packets in the driver, because the CPU on the host machine is much more powerful than the microprocessor of the wireless card. Checksums are sent to the firmware with each data packet. After the transmitter receives a NACK frame, its firmware can use these checksums directly, without recomputation. Checksums computed at the transmitter are used only to match those in the NACK frames and they are not transmitted. The receiver's driver combines a buffered corrupt packet with a correct recovery packet to reconstruct the original. Recovered packets

that cannot pass the extra Fletcher-32 checksum test are discarded.

6 Evaluation

In this section, we evaluate the throughput and latency performance of Maranello in implementation, isolate the factors that reduce recovery time, and run Maranello alongside unmodified 802.11 senders to ensure cooperative interaction.

We used 802.11b/g channels 1, 6, and 11 in environments with active APs and stations. This experimental approach has the advantage of injecting real-world interference and collisions as sources of packet error, but has the disadvantage of reducing the repeatability of experiments since contention varies. We enable auto rate feedback for all of the experiments and use Linux "minstrel" [19] as the rate control algorithm, which supports multiple rate retries and is the only rate control algorithm enabled in the Linux kernel 2.6.28 and above. (Our driver implementation is in 2.6.29-rc2.)

6.1 Maranello Increases Link Throughput

In the following, we show that Maranello can increase throughput for UDP traffic. We construct testbeds in three different environments: an industry research lab, a home, and a university building. We run Iperf [10] on randomly selected links from these testbeds to generate a CBR UDP stream to saturate the wireless channel. We focus on UDP to isolate link capacity from TCP dynamics.

We compare the throughput of Maranello and unmodified 802.11 in Figure 6. In these plots, the x-axis represents the throughput of 802.11 and the y-axis is the throughput of Maranello. Each point represents a pair of one-minute executions of Iperf, typically separated by less than 15 seconds. This separation is needed because we reload the firmware and driver, set up wireless interfaces, and initialize minstrel's bit rate table. Each point belongs to a group of ten points collected from randomly selected sender and receiver locations. In other words, we collected ten points, moved the receiver or sender station to another location, collected ten points again, and repeated. These figures include 370 (industry research lab), 390 (home), and 1000 (university building) points. The position of a point indicates the apparent throughput gain. For example, if a point is on the line marked "2X", the throughput gain is 2. We divide the points into 5 regions based on their throughput gain and show the percentage of points in each region in these figures. A point on a line is counted in the region above that line.

Figure 6 shows that Maranello can increase the throughput for UDP traffic; often by 30% or more. The university building environment shows higher throughput gain, because of increased contention and poorer channel conditions, than those observed in the other environments. There are more than 10 access points deployed for each of the 802.11b/g channels, 1, 6, and 11, and they are used by many people. For the other environments, each channel usually has fewer than four access points and relatively few users. To estimate the variability in the measurement of throughput over adjacent intervals, we also compare the throughput of 802.11 with itself. We pair the throughput of two consecutive runs with 802.11 into a point. Figure 6(d) shows the results for experiments done in our office building. The uncertainty in the throughput of adjacent measurements of unmodified 802.11 appears comparable to those of measurements between 802.11 and Maranello. Put simply, Maranello does not appear to increase the variability in throughput performance.

6.2 Maranello Reduces Recovery Latency

We define latency in this context to be the interval between when the firmware fetches the pending packet from the head of the TX FIFO to when an ACK is received. This includes the time spent inhibited by carrier sense, waiting for a transmission opportunity, and represents the time that the device is occupied with the transmission of an individual frame. We randomly select a link, then run Iperf for one minute for Maranello and 802.11 separately to get the per-packet latency. We use the firmware to record the measured time directly using the internal board clock: a 64-bit counter incremented every microsecond.

One might consider alternate definitions of latency. One might ignore contention and backoff time required by CSMA/CA; even though the card is occupied in the process of transmitting a packet, no signal is yet being transmitted. Such would be appropriate for measuring peak performance. Alternately, one might consider the time to successful delivery and ignore cases when the ACK is lost; the transmitter, of course, is still occupied.

We plot the CDF of latency for packets that need retransmissions in Figure 7. To make the comparison clear, we omit the latency for packets without retransmission, and we plot the latency of only one configuration of sender and receiver locations (other configurations are qualitatively similar but not composable). Maranello can deliver 90% of the packets that need retransmission within a latency of 4.16 ms. In contrast, 10% of 802.11 recovery latencies are above 17.1 ms. The small modes near 16 and 32 ms for 802.11 represent low-rate retransmissions: The minstrel default retransmission rate fallback attempts retransmissions at the original rate twice, followed by 1 Mbit/s up to four times if need.

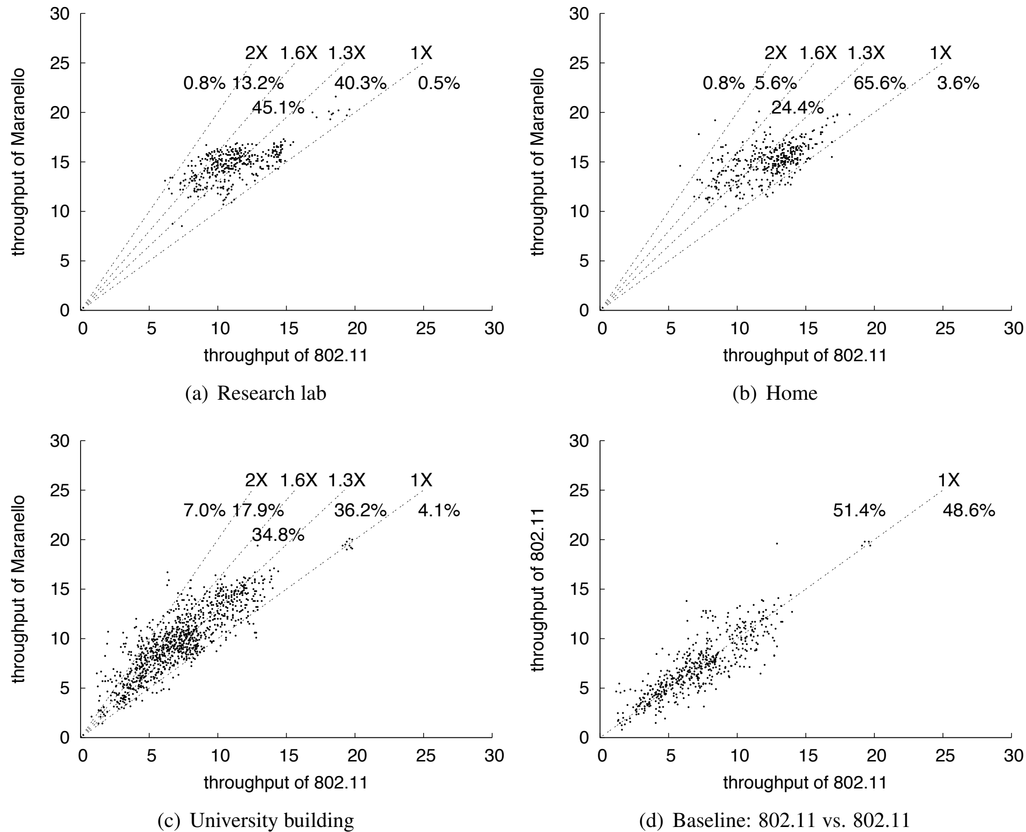


Figure 6: Maranello has a higher throughput than 802.11. Each figure compares 802.11 with Maranello in a different environment, or to show the uncertainty of the comparison, with 802.11 itself. Each point represents the performance of back-to-back one-minute UDP throughput measurements; ten points were collected for each configuration of sender and receiver stations.

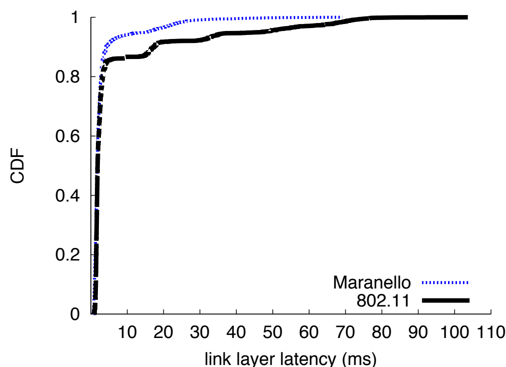


Figure 7: With block-based repair, Maranello recovers packets faster than 802.11's retransmissions.

6.3 The Sources of Throughput Gain and Latency Reduction

To break down the sources of performance improvement, we enhance the transmission status report for each packet with the following information: (1) whether a repair packet was used, (2) if used, at which attempt, and (3) the number of retransmitted blocks in the repair packet. The original report also includes (1) whether the packet is successfully delivered, (2) the number of attempts, (3) the bit rate used for the packet. With this information, we can calculate the delivery probability at each attempt, the transmission airtime and the number of transmitted bytes for each attempt. We run Iperf for one minute for 10 randomly selected links and plot in Figure 8 the probability of successful attempt for two retransmission rate fallback schemes: Linux "minstrel" fallback which always uses 1 Mbps as fallback rate, and 2-step fallback which drops the bit rate selected by minstrel for the initial transmissions by 2 steps (if possible) and uses it as fallback rate. The two-step fallback selection emulates the Broadcom driver for Windows XP (Section 4.1). In this figure, the x-axis is transmission attempt. The retry limit of Broadcom cards is 7, 1 initial transmission, and at most 6 retransmissions. The y-axis is the probability that an attempt can succeed.

Figure 8 shows that the probability of successful re-

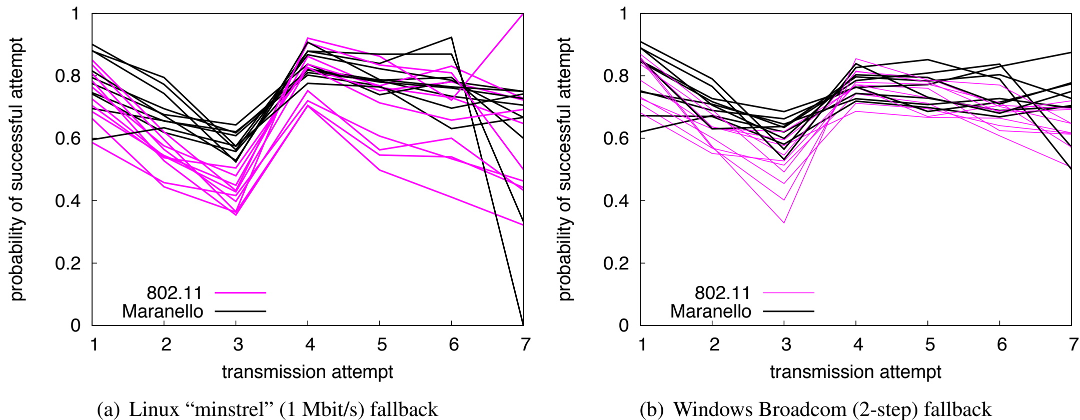


Figure 8: Maranello can successfully retransmit a packet earlier than 802.11. Each line represents a link measured either with 802.11 or Maranello; the probability that Maranello's recovery packets are delivered is typically higher.

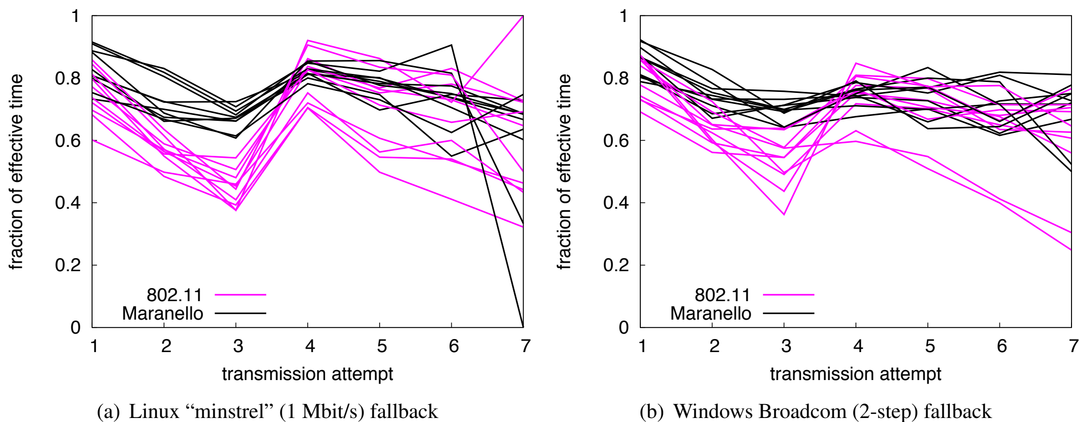


Figure 9: Maranello can use airtime more effectively for packet transmissions. Each line represents a link measured either with 802.11 or Maranello; Maranello spends more time transmitting bits not yet correctly received.

transmission for Maranello is usually higher than that of 802.11. Because the retransmission rate fallback does not budge for the first two retransmissions, the probability of successful retransmission can be thought of as the conditional probability that, given a packet (or two) recently failed to be delivered at the chosen rate, this next transmission at the same rate will be delivered. Not surprisingly, for 802.11, this probability descends more steeply than for Maranello. Maranello, in contrast, can send shorter repair packets, which are less likely to be corrupted [8], even at the original bit rate.

The delivery probability increases at the fourth attempt because the firmware reduces the bit rate for the last four attempts. The successful attempt probabilities for the first three attempts are more important, because most packets can succeed at the first two retransmissions. The estimate of the delivery probability for the seventh attempt (after three previous attempts at 1 Mbit/s) is uncertain due to the dearth of data. For example, the 7th attempt that had 0.0 delivery probability of Maranello, only one packet was transmitted seven times. For the 7th attempt with 1.0 delivery probability of 802.11, there were 5 packets transmitted 7 times and all succeeded at this last attempt.

We also plot the fraction of effective time for each transmission attempt in Figure 9. Effective time represents the time spent transmitting correct blocks; Maranello can use airtime more effectively, because the correct bits in corrupted packets may be combined with recovery packets to reconstruct the original packets and the transmission time of these correct bits is effective.

6.4 Deployment on Access Points

To show that Maranello can increase overall network performance and does not interact poorly with unmodified 802.11 devices, we deploy Maranello on Linksys wireless routers running OpenWRT [23]. We associate two desktop stations, A and B, with the Maranello AP. We

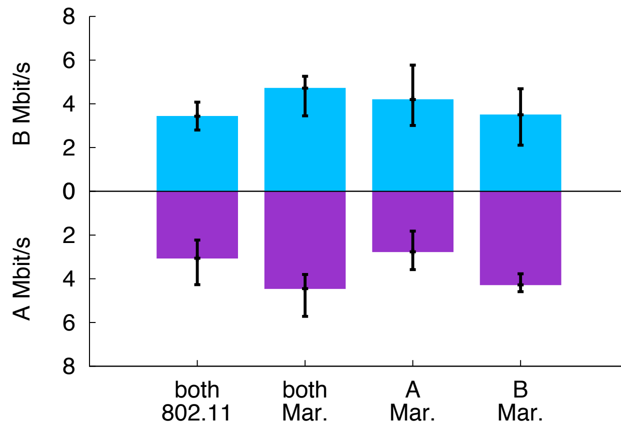


Figure 10: With two clients sending to an AP, on average, Maranello increases their individual and overall throughput. Error bars indicate min and max for five one minute runs.

run four types of experiments: A and B both running Maranello, both running 802.11, A running Maranello and B running 802.11, and vice versa. We connect a third station, C, to the AP using Ethernet, to act as an Iperf server. We do not run the Iperf server on the AP directly due to its limited CPU power. During a single one-minute experiment, A and B send UDP packets to C as fast as they can. Although experimenting with downlink traffic might be more typical of access point use, in that situation, that AP would be the only transmitter and would not show how Maranello transmitters interact with unmodified 802.11 transmitters.

Figure 10 plots the throughput of these two stations using a stacked bar graph. There are two key notes. First, running Maranello does not decrease the performance of the unmodified 802.11 station. That is, Maranello does not "cheat" the existing station of throughput. Second, when both stations run Maranello, the throughput is significantly increased for both stations. An interesting observation is that it appears not to help A or B to individually run Maranello when in contention. (The results in Section 6.1 imply that each station gains individually when running Maranello without a persistently competing station.) We plan to investigate this surprising result that Maranello is more social than selfish when competing with an unmodified station.

7 Discussion

In this section, we discuss how Maranello can be complementary with frame aggregation, which is used in 802.11n, and how the block size affects the performance of Maranello.

7.1 Frame Aggregation and Maranello are Complementary

To increase throughput, 802.11n reduces the 802.11 protocol overheads, such as interframe spacing, PHY layer headers and acknowledgment frames, by aggregating data packets into jumbo frames. Aggregated packets that are received incorrectly are indicated in a block acknowledgment which is sent back to the transmitter. The transmitter can then send a new chunk that contains only the corrupt packets. Even though only part of a packet may have errors in it, 802.11n frame aggregation must retransmit whole packets: correctly received bits are wasted.

Frame combining can improve throughput, but it also significantly increases latency, as senders must wait to aggregate enough frames to fill a jumbo frame. Block acknowledgments provide a complementary aggregation of feedback for 802.11n, where ACKs may be buffered together and sent as a group, similarly increasing per-packet latency. Maranello is complementary with these frame aggregation techniques because by repairing corrupted aggregated packets, Maranello can further increase link throughput.

7.2 Optimal Block Size

The Maranello block size is 64 bytes, primarily because it is the smallest multiple of 32 that can be supported by hardware (Section 5.2.2). A larger block size would increase computation efficiency somewhat and shorten NACKs, which may be useful at low bit rates. When the error rate is low, however, larger blocks may lead to repair packets with unnecessary extra bytes, wasting channel time.

We consider an interesting future direction of research to be dynamically adjusting the block size. The ideal block size may vary based on an estimate of wireless channel conditions and the bit rate chosen by the transmitter, which determines the bit rate of the acknowledgment and thus the transmission time of the NACK. When the NACK is transmitted at a low rate, it may be better for global throughput to keep NACK transmissions short than to be precise about the blocks in error. A similar tradeoff exists in the FEC systems between the coding rate of error correction bits and recovery efficiency. Another approach to determine the optimal block size that we intend to explore is to use theoretical models of wireless communication errors [13, 29].

8 Conclusion

In this paper, we design, implement, and evaluate Maranello, a practical partial packet recovery protocol for 802.11 wireless networks. Maranello has the following features simultaneously: (a) it introduces no extra

bits in correct transmissions, (b) it reduces recovery latency, except in rare cases, (c) it is compatible with the 802.11 protocol, and (d) it can be incrementally deployed on widely available 802.11 devices.

We implemented Maranello using OpenFWWF open source firmware. This implementation, and Maranello's compatibility with 802.11, allowed us to test in three different, live environments over heavily used 802.11b/g channels where contention and interference are realistic. We found significant throughput gains when running Maranello over 802.11 in consecutive intervals. We also installed Maranello on access points running OpenWRT to demonstrate that Maranello does not compete unfairly with unmodified 802.11 devices and that the processing requirements of Maranello do not preclude performance improvement. Moreover, we evaluate Maranello's performance compared to recently-proposed recovery protocols using a trace-driven simulation.

9 Acknowledgment

We thank the anonymous reviewers, Daniel Halperin, Aravind Srinivasan and Dave Levin, and our shepherd Srini Seshan, for insightful comments and discussion. Vinko Erceg answered our questions about Broadcom WiFi chipsets. This work was supported in part by NSF ITR Award CNS-0426683, NSF Award CNS-0626636 and NSF Award CNS-0643443. Part of this work was done when Bo Han was a summer intern at AT&T Labs – Research.

References

[1] b43 linux kernel driver for broadcom chipsets. http://linuxwireless.org/en/users/Drivers/b43/.

[2] G. Bianchi, *et al.* Experimental assessment of the backoff behavior of commercial IEEE 802.11b network cards. In *INFOCOM*, 2007.

[3] S. Choi, Y. Choi, and I. Lee. IEEE 802.11 MAC-level FEC scheme with retransmission combining. *IEEE Transactions on Wireless Communications*, 5(1):203–211, 2006.

[4] H. Dubois-Ferriere, D. Estrin, and M. Vetterli. Packet combining in sensor networks. In *SenSys*, 2005.

[5] J. G. Fletcher. An arithmetic checksum for serial transmissions. *IEEE Transactions on Communications*, 30(1):247–252, 1982.

[6] R. K. Ganti, P. Jayachandran, H. Luo, and T. F. Abdelzaher. Datalink streaming in wireless sensor networks. In *SenSys*, 2006.

[7] F. Gringoli and L. Nava. Open firmware for WiFi networks. http://www.ing.unibs.it/openfwwf/.

[8] B. Han and S. Lee. Efficient packet error rate estimation in wireless networks. In *TridentCom*, 2007.

[9] IEEE std 802.11, 2007.

[10] Iperf. http://sourceforge.net/projects/iperf/.

[11] A. P. Iyer, *et al.* Fast resilient jumbo frames in wireless LANs. In *IWQoS*, 2009.

[12] K. Jamieson and H. Balakrishnan. PPR: Partial packet recovery for wireless networks. In *SIGCOMM*, 2007.

[13] A. Kopke, A. Willig, and H. Karl. Chaotic maps as parsimonious bit error models of wireless channels. In *INFOCOM*, 2003.

[14] K. C.-J. Lin, N. Kushman, and D. Katabi. ZipTx: Harnessing partial packets in 802.11 networks. In *MOBICOM*, 2008.

[15] M.-H. Lu, P. Steenkiste, and T. Chen. Using commodity hardware platform to develop and evaluate CSMA protocols. In *WinTech*, 2008.

[16] M.-H. Lu, P. Steenkiste, and T. Chen. Design, implementation and evaluation of an efficient opportunistic retransmission protocol. In *MOBICOM*, 2009.

[17] Linux kernel mac80211 framework for wireless device drivers. http://linuxwireless.org/en/developers/Documentation/mac80211/.

[18] Madwifi linux kernel driver for WLAN devices with Atheros chipsets. http://madwifi-project.org/.

[19] Minstrel rate control algorithm. http://linuxwireless.org/en/developers/Documentation/mac80211/RateControl/minstrel/.

[20] A. Miu, H. Balakrishnan, and C. E. Koksal. Improving loss resilience with multi-radio diversity in wireless networks. In *MOBICOM*, 2005.

[21] M. Neufeld, *et al.* SoftMAC – flexible wireless research platform. In *HotNets-IV*, 2005.

[22] G. Nychis, *et al.* Enabling MAC protocol implementations on software-defined radios. In *NSDI*, 2009.

[23] OpenWrt linux distribution for embedded devices. http://openwrt.org/.

[24] P. S. Sindhu. Retransmission error control with memory. *IEEE Transactions on Communications*, 25(5):473–479, 1977.

[25] J. Stone, M. Greenwald, C. Partridge, and J. Hughes. Performance of checksums and CRC's over real data. *IEEE/ACM Transactions on Networking*, 6(5):529–543, 1998.

[26] K. Tan, *et al.* Sora: High performance software radio using general purpose multi-core processors. In *NSDI*, 2009.

[27] G. Woo, P. Kheradpour, D. Shen, and D. Katabi. Beyond the bits: Cooperative packet recovery using physical layer information. In *MOBICOM*, 2007.

[28] B. Zhao and M. C. Valenti. Practical relay networks: A generalization of Hybrid-ARQ. *JSAC*, 23(1):7–18, 2005.

[29] M. Zorzi, R. R. Rao, and L. B. Milstein. Error statistics in data transmission over fading channels. *IEEE Transactions on Communications*, 46(11):1468–1477, 1998.

[30] J. Zweig and C. Partridge. TCP alternate checksum options. IETF RFC-1145, 1990.

Reverse traceroute

Ethan Katz-Bassett[*] *Harsha V. Madhyastha*[†] *Vijay Kumar Adhikari*[‡] *Colin Scott*[*]
Justine Sherry[*] *Peter van Wesep*[*] *Thomas Anderson*[*] *Arvind Krishnamurthy*[*]

Abstract

Traceroute is the most widely used Internet diagnostic tool today. Network operators use it to help identify routing failures, poor performance, and router misconfigurations. Researchers use it to map the Internet, predict performance, geolocate routers, and classify the performance of ISPs. However, traceroute has a fundamental limitation that affects all these applications: it does not provide reverse path information. Although various public traceroute servers across the Internet provide some visibility, no general method exists for determining a reverse path from an arbitrary destination.

In this paper, we address this longstanding limitation by building a reverse traceroute system. Our system provides the same information as traceroute, but for the reverse path, and it works in the same case as traceroute, when the user may lack control of the destination. We use a variety of measurement techniques to incrementally piece together the path from the destination back to the source. We deploy our system on PlanetLab and compare reverse traceroute paths with traceroutes issued from the destinations. In the median case our tool finds 87% of the hops seen in a directly measured traceroute along the same path, versus only 38% if one simply assumes the path is symmetric, a common fallback given the lack of available tools. We then illustrate how we can use our reverse traceroute system to study previously unmeasurable aspects of the Internet: we present a case study of how a content provider could use our tool to troubleshoot poor path performance, we uncover more than a thousand peer-to-peer AS links invisible to current topology mapping efforts, and we measure the latency of individual backbone links with average error under a millisecond.

1 Introduction

Traceroute is a simple and widely used Internet diagnostic tool. It measures the sequence of routers from the source to the destination, supplemented by round-trip delays to each hop. Operators use it to investigate routing failures and performance problems [39]. Researchers use it as the basis for Internet maps [1, 22, 34], path prediction [22], geolocation [42, 14], ISP performance analysis [25], and anomaly detection [46, 19, 44, 43].

However, traceroute has a fundamental limitation – it provides no reverse path information, despite the fact that policy routing and traffic engineering mean that paths are generally asymmetric [15]. As Richard Steenbergen, CTO for nLayer Communications, put it at a recent tutorial for network operators on troubleshooting, "the number one go-to tool is traceroute," but "asymmetric paths [are] the number one plague of traceroute" because "the reverse path itself is completely invisible" [39].

This invisibility hinders operators. For instance, although Google has data centers distributed around the world, 20% of client prefixes experience unreasonably high latency, even with a nearby server. In working with a Google group trying to improve this performance, we found that we would have been able to more precisely troubleshoot problems if we could measure the path from clients back to Google [21].

Similarly, the lack of reverse path information restricts researchers. Traceroute's inability to measure reverse paths forces unrealistic assumptions of symmetry on systems with goals ranging from path prediction [22], geolocation [42, 14], ISP performance analysis [25], and prefix hijack detection [46]. Recent work shows that measured topologies miss many of the Internet's peer-to-peer links [29, 16] because mapping projects [1, 22, 34] lack the ability to measure paths from arbitrary destinations.

Faced with this shortcoming with the traceroute tool, operators and researchers turn to various limited workarounds. Surprisingly, network operators often resort to posting problems on operator mailing lists asking others to issue traceroutes to help diagnosis [30, 41]. Public web-accessible traceroute servers hosted at various locations around the world provide some help, but their numbers are limited. Without a server in every network, one cannot know whether any of those available have a path similar to the one of interest. Further, they are not intended for the heavy load incurred by regular monitoring. A few modern systems attempt to deploy traceroute clients on end-user systems around the world [34, 9], but none of them are close to allowing an arbitrary user to trigger an on-demand traceroute towards the user from anywhere in the world.

Our goal is to address this basic restriction of traceroute by building a tool to provide the same basic information as traceroute – IP-address-level hops along the path, plus round-trip delay to each – but along the reverse path from the destination back to the source. We have implemented our reverse traceroute system and make

[*]Dept. of Computer Science, Univ. of Washington, Seattle.
[†]Dept. of Computer Science, Univ. of California, San Diego.
[‡]Dept. of Computer Science, Univ. of Minnesota.

it available at `http://revtr.cs.washington.edu`. While traceroute runs as a stand-alone program issuing probes on its own behalf, ours is a distributed system comprised of a few tens to hundreds of vantage points, owing to the difficulty in measuring reverse paths. As with traceroute, our reverse traceroute tool does not require control of the destination, and hence can be used with arbitrary targets. All our tool requires of the target destination is an ability to respond to probes, the same requirement as standard traceroute. It does not require new functionality from routers or other network components.

Our system builds a reverse path incrementally, using a variety of methods to measure reverse hops, and stitching them together into a path. We combine the view of multiple vantage points to gather information unavailable from any single one. We start by measuring the paths from the vantage points to the source. This limited atlas of a few hundred or thousand routes to the source serves to bootstrap the rest of our measurements, allowing us to measure the path from an arbitrary destination by building back the path from the destination until it intersects the atlas. We use three main measurement techniques to build backwards. First, we rely on the fact that Internet routing is generally destination-based, allowing us to piece together the path one hop at a time. Second, we employ the IP timestamp and record route options to identify hops along the reverse path. Third, we use limited source spoofing – spoofing from one vantage point as another – to use the vantage point in the best position to make the measurement. This controlled spoofing allows us to overcome many of the limitations inherent in using IP options [36, 35, 13], while remaining safe, as the spoofed source address is one of our hosts. Just as many projects use traceroute, others have used record route and spoofing for other purposes. Researchers used record route to identify aliases and generate accurate topologies [35], and our earlier work used spoofing to characterize reachability problems [19]. In this work, we are the first to show that the combination of these techniques can be used to measure arbitrary reverse paths.

Experienced users realize that, while traceroute is useful, it has numerous limitations and caveats, and can be potentially misleading [39]. Similarly, our tool has limitations and caveats. Section 5.1 includes a thorough discussion of how the output of our tool might differ from a direct traceroute from the destination to the source, as well as how both might differ from the actual path traversed by traffic. Just as traceroute provides little visibility when routers do not send TTL-expired messages, our technique relies on routers honoring IP options. When our measurement techniques fail to discern a hop along the path, we fall back on assuming the hop is traversed symmetrically; our evaluation results show that, in the median (mean) case for paths between PlanetLab sites, we measure 95% (87%) of hops without assuming symmetry. The need to assume symmetry in cases of an unresponsive hop points to a limitation of our tool compared to traceroute; whereas traceroute can often measure past an unresponsive hop or towards an unreachable destination, our tool must sometimes guess that it is measuring the proper path.

We rely on routers to be "friendly" to our techniques, yet some of our techniques have the potential for abuse and can be tricky for novices to use without causing disturbances. As we ultimately want our tool widely used operationally, we have attempted to pursue our approach in a way that encourages continued router support. Our system performs measurements in a way that emphasizes network friendliness, controlling probe rate across all measurements. We presented the work early on at NANOG [28] and RIPE [32] conferences, and so far the response from operators has been positive towards supporting our methods (including source spoofing). We believe the goal of wide use is best served by a single, coordinated system that services requests from all users.

We evaluate the effectiveness of our system as deployed today, though it should improve as we add vantage points. We find that, in the median (mean) case for paths between PlanetLab sites, our technique reveals 87% (83%) of the routers and 100% (94%) of the points-of-presence (PoPs), compared to a traceroute issued from the destination. Paths between public traceroute servers and PlanetLab show similar results. Because our technique requires software at the source, our evaluation is limited to paths back to PlanetLab nodes we control. We believe our reverse traceroute system can be useful in a range of contexts, and we provide three illustrative examples. We present a case study of how a content provider could use our tool to troubleshoot poor reverse path performance. We also uncover thousands of links at core Internet exchange points that are invisible to current topology mapping efforts. We use our reverse traceroute tool to measure link latencies in the Sprint backbone network with less than a millisecond of error, on average.

2 Background

In this section, we provide the reader some background on Internet routing and traceroute.

Internet routing: First, a router generally determines the route on which to forward traffic based only on the destination. With a few caveats such as load-balancing and tunneling, the route from a given point towards a particular destination is consistent for all traffic, regardless of its source. While certain tunnels may violate this assumption, best practices encourage tunnels that appear as atomic links. Second, asymmetry between forward and reverse paths stems from multiple causes. An AS is free to choose its next hop among the alternatives, whether or

not that leads to a symmetric route. Two adjacent ASes may use different peering points in the two directions due to policies such as early-exit/hot-potato routing. Even within an individual AS, traffic engineering objectives may lead to different paths.

Standard traceroute tool: Traceroute measures the sequence of routers from a source to a destination, without requiring control of the destination. When traceroute was originally developed, most paths were symmetric, but that assumption no longer holds. Traceroute works by sending a series of packets to the destination, each time incrementing the time-to-live (TTL) from an initial value of one, in order to get ICMP TTL exceeded responses from each router on the path in turn. Each response will have an IP address of an interface of the corresponding router. Additionally, traceroute measures the time from the sending of each packet to the receipt of the response, yielding a round-trip latency to each intermediate router. Because the destination resets the TTL in its reply, traceroute only works in the forward direction.

The path returned by traceroute is a feasible, but possibly inaccurate route. First, each hop comes from a response to a different probe packet, and the different probes may take different paths for reasons including contemporaneous routing changes or load balancing. The Paris traceroute customizes probe packets to provide consistent results across flow-based load balancers, as well as to systematically explore the load-balancing options [2]. For all our traceroutes, we use the Paris option that measures a single consistent path. Second, some routers on the path may not respond. For example, some routers may be configured to rate-limit responses or to not respond at all. Third, probe traffic may be treated differently than data traffic.

Despite these caveats, traceroute has proved to be extremely useful. Essential to traceroute's utility is its universality, in that it does not require anything of the destination other than an ability to respond to probe packets.

3 Reverse Traceroute

We seek to build a reverse path tool equivalent to traceroute. Like traceroute, ours should work universally without requiring control of a destination, and it should use only features available in the Internet as it exists today. The reverse traceroute tool should return IP addresses of routers along the reverse path from a destination back to the source, as well as the round-trip delay from those routers to the source.

At a high level, the source requests a path from our system, which coordinates probes from the source and from a set of distributed vantage points to discover the path. First, distributed vantage points issue traceroutes to the source, yielding an atlas of paths towards it (Fig. 1(a)). This atlas provides a rich, but limited in scope, view of how parts of the Internet route towards the source. We use this limited view to bootstrap measurement of the desired path. Because Internet routing is generally destination-based, we assume that the path to the source from any hop in the atlas is fixed (over short time periods) regardless of how any particular packet reaches that hop; once the path from the destination to the source reaches a hop in the atlas, we use the atlas to derive the remainder of the path. Second, using techniques we explain in Sections 3.1 and 3.2, we measure the path back from the destination incrementally until it intersects this atlas (Fig. 1(b)). Finally, as shown in an example in Section 3.3, we merge the two components of the path, the destination-specific part measured from the destination until it intersects the atlas, and the atlas-derived path from this intersection back to the source, to yield a complete path (Fig. 1(c)).

3.1 Identify Reverse Hops with IP Options

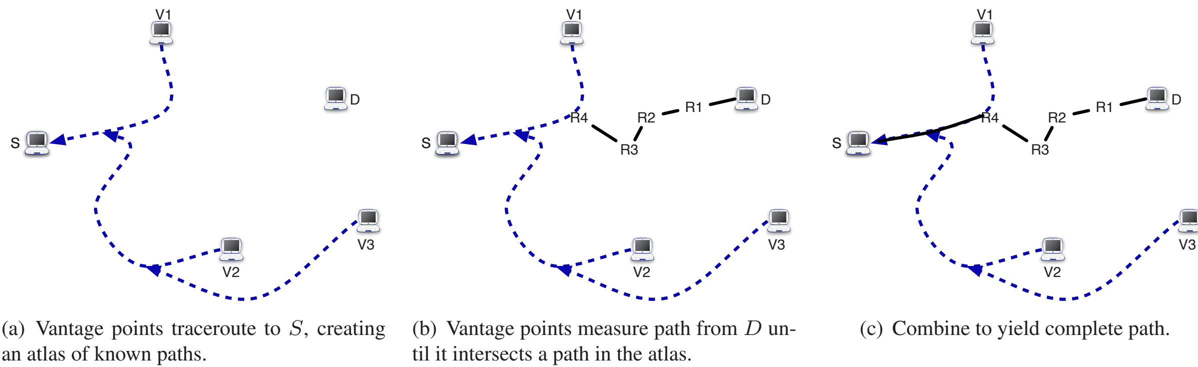
We use two basic measurement primitives, the Record Route and Timestamp IP options. While TTL values are reset by the destination, restricting traceroute to measuring only on the forward path, IP options are generally reflected in the reply from a destination, so routers along both the forward and reverse path process them. We briefly explain how the options work:

IP Record-route option (*RR*): With this option set, a probe records the router interfaces it encounters. The IP standard limits the number of recorded interfaces to 9; once those fill, no more interfaces are recorded.

IP timestamp option (*TS*): IP allows probes to query a set of specific routers for timestamps. Each probe can specify up to four IP addresses, in order; if the probe traverses the router matching the next IP address that has yet to be stamped, the router records a timestamp. The addresses are ordered, so a router will not timestamp if its IP address is in the list but is not the next one.

We use these options to gather reverse hops as follows:

- *RR-Ping*$(S \rightarrow D)$: As shown in Figure 2(a)), the source S issues an ICMP Echo Request (henceforth *ping*) probe to D with the RR option enabled. If RR slots remain when the destination sends its response, then routers on the reverse path will record some of that route. This allows a limited measurement of the reverse path, as long as the destination is fewer than 9 hops from the source.
- *TS-Query-Ping*$(S \rightarrow D|D, R)$: As shown in Figure 2(b)), the source S issues an ICMP ping probe to D with the timestamp query enabled for the ordered pair of IP addresses D and R. R will record its timestamp only if it is encountered by the probe after D has stamped the packet. In other words, if S receives a timestamp for R, then it knows R appears

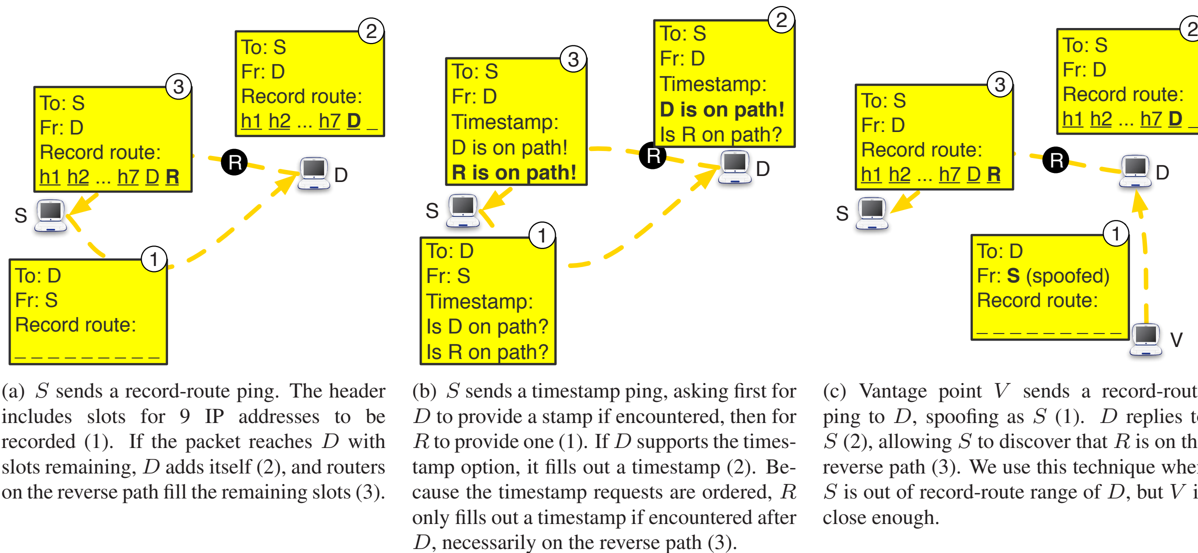


(a) Vantage points traceroute to S, creating an atlas of known paths.

(b) Vantage points measure path from D until it intersects a path in the atlas.

(c) Combine to yield complete path.

Figure 1: High-level overview of the reverse traceroute technique. We explain how to measure from D back to the atlas in § 3.1- 3.2.



(a) S sends a record-route ping. The header includes slots for 9 IP addresses to be recorded (1). If the packet reaches D with slots remaining, D adds itself (2), and routers on the reverse path fill the remaining slots (3).

(b) S sends a timestamp ping, asking first for D to provide a stamp if encountered, then for R to provide one (1). If D supports the timestamp option, it fills out a timestamp (2). Because the timestamp requests are ordered, R only fills out a timestamp if encountered after D, necessarily on the reverse path (3).

(c) Vantage point V sends a record-route ping to D, spoofing as S (1). D replies to S (2), allowing S to discover that R is on the reverse path (3). We use this technique when S is out of record-route range of D, but V is close enough.

Figure 2: Three measurement techniques that allow us to establish that R is on the reverse path from D back to S. In §4, we give two techniques, akin to (c), that use spoofing to overcome limitations in timestamp support.

on the reverse path. For our purposes, the value of the timestamp is meaningless; we just care whether or not a particular router processes the packet. Thus, if we guess a router on the return path, the TS option can confirm our hypothesis.

We use existing network topology information – specifically IP-level connectivity of routers from Internet mapping efforts [22] – to determine candidate sets of routers for the reverse path. Routers adjacent to D in the topology are potential next hops; we use timestamp query probes to check whether any of these potential next hops is on the path from D to S.

Note that there are some caveats to using the probes outlined above. One is that from each vantage point, only a fraction of routers will be reachable within record route's limit of 9 hops. Another is that some ISPs filter and drop probes with IP options set. Further, some routers do not process the IP options in the prescribed manner. Fortunately, we can overcome these limitations in the common case by carefully orchestrating the measurements from a diverse set of vantage points.

3.2 Spoof to Best Use Record Route

A source-spoofed probe (henceforth referred to as a spoofed probe) is one in which the prober sets the source address in the packet to one other than its own. We use a limited form of spoofing, where we replace the source address in a probe with the "true" source of the reverse traceroute. This form of spoofing is an extremely powerful measurement tool. When V probes D spoofing as S, D's response will go to S; we refer to V as the spoofer and S as the receiver. This method allows the probe to traverse the path from D to S without having to traverse the path from S to D and without having a vantage point in D's prefix. We could hypothetically achieve a similar probe trajectory using loose source routing (from V

to S, but source routed through D) [31]. However, a source-routed packet can be identified and filtered anywhere along the path, and such packets are widely filtered [3], too often to be useful in our application. On the other hand, if a spoofed packet is not ingress filtered near the spoofer, it thereafter appears as a normal packet; we can use a source capable of spoofing to probe along any path. Based on our measurements to all routable prefixes, many routers that filter packets with the source route option do not filter packets with the timestamp or record route options. This difference is likely because source routing can be used to violate routing policy, whereas the timestamp and record route options cannot.

This arrangement allows us to use the most advantageous vantage point with respect to the particular measurement we want to perform. Our earlier system Hubble used limited spoofing to check one-way reachability [19]; we use it here to overcome limitations of IP options. Without spoofing, RR's 9 IP address limit restricts it to being useful only when S is near the target. However, as shown in Figure 2(c),if some vantage point V is close enough to reach the target within 8 RR slots, then we can probe from V spoofing as S to receive IP addresses on the path back to S. Similarly, spoofing can bypass problematic ASes and machines, such as those that filter timestamp-enabled packets or those that do not correctly implement the option.

Although spoofing is often associated with malicious intent, we use it in a very controlled, safe fashion. A node requests a reverse path measurement, then receives responses to probes sent by vantage points spoofing as it. No harm can come from causing one of our own nodes to receive measurement packets. This form of spoofing shares a purpose with the address rewriting done by middleboxes such as NATs, controlling the flow of traffic to a cooperative machine, rather than with malicious spoofing which seeks concealment. Since some ISPs filter spoofed packets, we test from each host and only send further spoofed probes where allowed. We have been issuing spoofed probes for over two years without complaint.

Roughly 20% of PlanetLab sites allow spoofing; this ability is not limited to PlanetLab: the Spoofer project tested 12,000 clients and found that 31% could send spoof packets [5]. Even if filtering increases, we believe, based on positive feedback from operators, that the value of our service will encourage an allowance (supported by router ACLs) for a small number of measurement nodes to issue spoofed probes using a restricted set of ports. An even simpler approach is to have routers rate limit these spoofed options packets (just as with UDP probes) and filter spoofed probes sent to broadcast addresses, thereby reducing the security concerns without diminishing their utility for network measurements.

3.3 Incrementally Build Paths

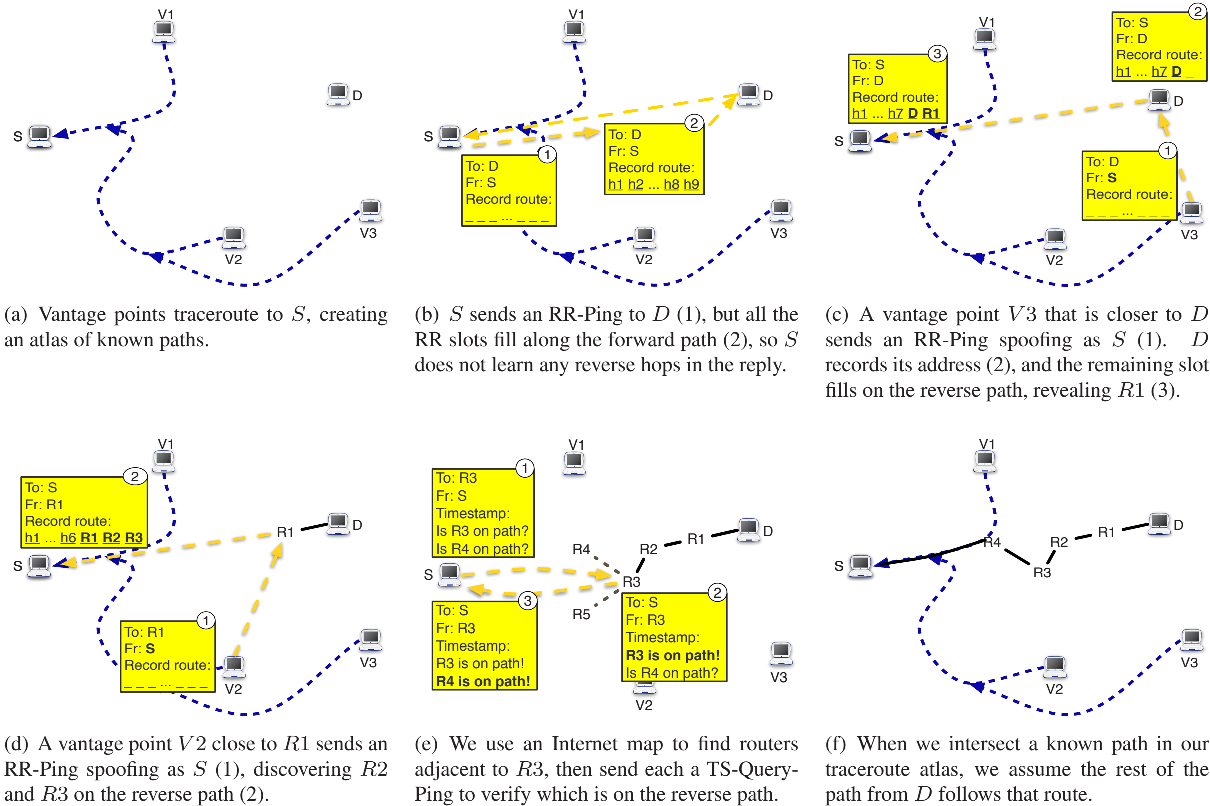
IP option-enabled probes, coupled with spoofing as S from another vantage point, give us the ability to measure a reverse hop from D on the path back to S. We can use the same techniques to stitch together a path incrementally – once we know the path from D goes through R, we need only determine the route at R towards S when attempting to discover the next hop. Because Internet routing is generally based on the destination, each intermediate router R we find on the path can become the new destination for a reverse traceroute back to the source. Further, if R is on a path from some vantage point V to S, then we can infer the rest of D's return path from R onward as being the same as V's. This assumption holds even in cases of packet-, flow-, and destination-based load balancing, so long as R balances traffic independently of other routers and of the source.

Figure 3 illustrates how we can compose the above set of techniques to determine the reverse path from D to S, when we have control over S and a set of other vantage points (V_1, V_2, V_3). We assume that we have a partial map of router-level connectivity, e.g., from a traditional offline mapping effort.

We begin by having the vantage points issue traceroute probes to S (Figures 1(a) and 3(a)). These serve as a baseline set of observed routes towards S that can be used to complete a partially inferred reverse path. We then issue *RR-Ping*$(S \rightarrow D)$ to determine if the source S is within 8 RR hops of the destination, i.e., whether a ping probe from S can reach D without filling up its entire quota of 9 RR hops (Figure 3(b))[1]. If the source is within 8 RR hops of D, this probe would determine at least the first hop on the reverse path, with further hops recovered in an iterative manner.

If the source is not within 8 hops, we determine whether some vantage point is within 8 RR hops of D (Section 4.5 describes how we do this). Let V_3 be one such vantage point. We then issue a spoofed RR ping probe from V_3 to D with the source address set to S (Figure 3(c)). This probe traverses the path $V_3 \rightarrow D \rightarrow S$ and records IP addresses encountered. The probe reveals R_1 to be on the reverse path from D to S. We then iterate over this process, with the newly found reverse hop as the target of our probes. For instance, we next determine a vantage point that is within 8 RR hops of R_1, which could be a different vantage point V_2. We use this new vantage point to issue a spoofed RR ping probe to determine the next hop on the reverse path (Figure 3(d)).

[1] This is not quite as simple as sending a TTL=8 limited probe, because of issues with record route implementations [35]. Some routers on the forward path might not record their addresses, thereby freeing up more slots for the reverse path, while some other routers might record multiple addresses or might record their address but not decrement or respond to TTL-limited probes.



(a) Vantage points traceroute to S, creating an atlas of known paths.

(b) S sends an RR-Ping to D (1), but all the RR slots fill along the forward path (2), so S does not learn any reverse hops in the reply.

(c) A vantage point $V3$ that is closer to D sends an RR-Ping spoofing as S (1). D records its address (2), and the remaining slot fills on the reverse path, revealing $R1$ (3).

(d) A vantage point $V2$ close to $R1$ sends an RR-Ping spoofing as S (1), discovering $R2$ and $R3$ on the reverse path (2).

(e) We use an Internet map to find routers adjacent to $R3$, then send each a TS-Query-Ping to verify which is on the reverse path.

(f) When we intersect a known path in our traceroute atlas, we assume the rest of the path from D follows that route.

Figure 3: Illustration of the incremental construction of a reverse path using diverse information sources.

In some cases, a single RR ping probe may determine multiple hops, as in the illustration with R_2 and R_3.

Now, consider the case where neither S nor any of the vantage points is within 8 hops of R_3. In that case, we consider the potential next hops to be routers adjacent to R_3 in the known topology. We issue timestamp probes to verify whether the next hop candidates R_4 and R_5 respond to timestamp queries *TS-Query-Ping*$(S \rightarrow D|D, R_4)$ and *TS-Query-Ping*$(S \rightarrow D|D, R_5)$ (as shown in Figure 3(e)). When R_4 responds, we know that it is adjacent to R_3 in the network topology and is on the reverse path from R_3, and so we assume it is the next hop on the reverse path. We continue to perform incremental reverse hop discovery until we intersect with a known path from a vantage point to S [2], at which point we consider that to be the rest of the reverse path (Figures 1(c) and 3(f)). Once the procedure has determined the hops in the reverse path, we issue pings from the source to each hop in order to determine the round-trip latencies.

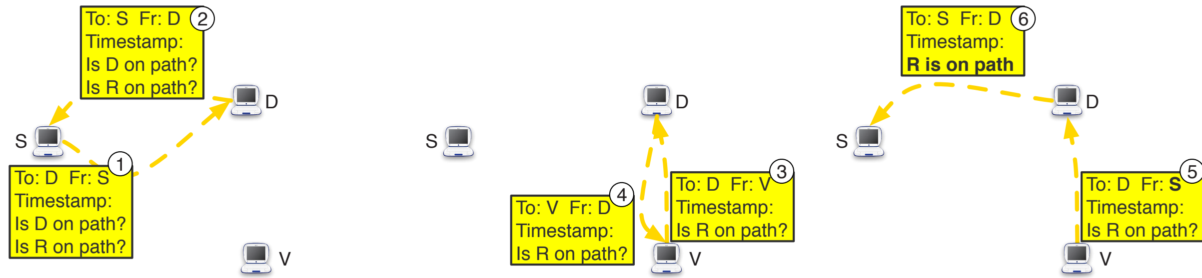
Sometimes, we may be unable to measure a reverse hop using any of our techniques, but we still want to provide the user with useful information. When reverse traceroute is unable to calculate the next hop in a path, the source issues a standard traceroute to the last known hop on the path. We then assume the last link is traversed symmetrically, and we try to calculate the rest of the reverse path from there. In Section 5.1 and Section 5.2, we present results that show that we usually do not have to assume many symmetric hops and that, even with this approximation, we still achieve highly accurate paths.

[2]Measurement techniques may discover different addresses on a router [36], so we determine intersections using alias data from topology mapping projects [20, 22, 35] and a state-of-the-art technique [4].

4 System Implementation

Section 3 describes how our techniques in theory would allow us to measure a reverse path. In this section, we discuss how we had to vary from that ideal description in response to realities of available vantage points and of router implementations. In addition, the following goals drive our system design:

- *Accuracy:* It should be robust to variations in how options-enabled packets are handled by routers.
- *Coverage:* The system should work for arbitrary destinations irrespective of ISP-specific configurations.
- *Scalability:* It needs to be selective with the use of vantage points and introduce as little measurement traffic as possible.



(a) S sends a timestamp ping to D (1) and receives a reply, but D has not filled out a timestamp (2).

(b) We find a V s.t., when V pings D asking for R's timestamp (3), it does not receive a stamp (4). This response indicates that R is not on V's path to D.

(c) V spoofs as S, pinging D (5), and S receives a timestamp for R (6). Because we established that R is not on V's path to D, R must be on the reverse path from D to S.

Figure 4: Example of how we discover a reverse hop with timestamp even though D does not stamp, as long as it replies.

4.1 Architecture

Our system consists of vantage points (VPs), which issue measurements, a controller, which coordinates the VPs to measure reverse paths, and sources, which request paths to them. We use a local machine at UW as a controller. When a VP starts up, it registers with the controller, which can then send it probe requests. A source runs our software to issue standard traceroutes, *RR-Pings*, and *TS-Query-Pings*, and to receive responses to probes from VPs spoofing as the source. However, it need not spoof packets itself. Currently, our source software only runs on Linux and (like the ICMP traceroute option `traceroute -I`) requires root permission. The controller receives requests from sources and combines the measurements from VPs to report reverse path information. While measuring a reverse traceroute, the controller queues up all incoming requests. When the ongoing measurement completes, the controller serves all requests in the queue as a batch, in synchronized rounds of probes. This design lets us carefully control the rate at which we probe any particular router, as well as to reuse measurements when a particular source requests multiple destinations.

We use topology maps from iPlane [22][3] to identify adjacent routers to test with *TS-Query-Pings* (Fig. 3(e)). To increase the set of possible next-hop routers, we consider the topology to be the union of maps from the previous 2 weeks. Since we verify the reverse hops using option-enabled pings, stale topology information makes our system less efficient but does not introduce error.

4.2 Current Deployment

Our current deployment uses one host at each of the more than 200 active PlanetLab sites as VPs to build an atlas of traceroutes to a source (Fig. 3(a)). Over the course of our study, 60+ PlanetLab sites allowed spoofed probes at least some of the time. We employ one host at each of these sites as spoofing nodes. Routers upstream from the other PlanetLab sites filter spoofed probes, so we did not spoof from them. We also use one host at each of 14 Measurement Lab sites [26], most of which allow spoofing. Various organizations provide public web-accessible traceroute servers, and we employ 1200 of them [3]. These nodes issue only traceroutes and cannot set IP options, spoof, or receive spoofed probes. We use them to expand the sets of known paths to our sources.

We have currently tested our client software only on PlanetLab (Linux) machines. We make it available as a demo; our website `http://revtr.cs.washington.edu` allows users to enter an IP address and measure the reverse path back from it to a PlanetLab node. Packaging the code for widespread deployment is future work. Because we have dozens of operators asking to use the system, we are being patient to avoid a launch that does not perform up to expectations.

4.3 Correcting for Variations in IP Options Support

We next explain how we compensate for variations in support for the timestamp option. When checking if R is on the reverse path from D, we normally ask for both D's and R's timestamp, to force R to only stamp on the reverse path. However, we found that, of the addresses in a day's iPlane atlas that respond to ping, 16.6% of the addresses respond to timestamp-enabled pings, but do not stamp, so we cannot use that technique to know that R stamped on the reverse path. Figure 4 illustrates how we use spoofing to address this behavior. Essentially, we find a VP V which we can establish does not have R on its path to D, then V pings D spoofing as S, asking for R's timestamp (but not D's). If S receives a stamp for R, it proves R is on the reverse path from D. This technique will not work if all vantage points have R on their paths. We examined iPlane traceroutes to destinations in 140,000 prefixes and found at least two adjacent hops for 55% of destinations.

[3]iPlane issues forward traceroutes from PlanetLab sites and traceroute servers to around 140K prefixes.

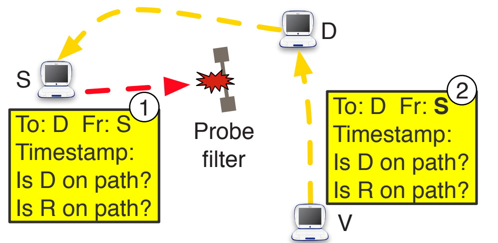


Figure 5: If a timestamp probe from S encounters a filter (1), we can often bypass it by spoofing as S from a different vantage point (2), as long as the filter is just on the forward path.

4.4 Avoiding Probe Filters to Improve Coverage

We next discuss techniques to improve the coverage of our measurements. Some networks may filter ICMP packets, and others filter packets with options enabled. In the course of measuring a reverse path, if a source attempts a TS or RR measurement and does not receive a response, we retry the measurement with a VP spoofing as the source. As seen in Figure 5, if filtering occurs only along the source's forward path, and the VP's forward path does not have a filter, the original source should receive the response.

We demonstrate the effectiveness of this approach on a small sample of 1000 IP addresses selected at random out of those in the iPlane topology known to respond to timestamp probes. The 1000 destinations include addresses in 662 ASes. We chose 10 spoofing PlanetLab vantage points we found to receive (non-spoofed) timestamp responses from the highest number of destinations, plus one host at each of the 209 working non-spoofing PlanetLab site. First, each non-spoofing node sent a series of timestamp pings to each destination; redundant probes account for loss due to something other than permanent filters. Of the 209 hosts, 103 received responses from at least 700 destinations; we dropped them from the experiment, as they do not experience significant filtering. Then, each spoofing vantage point sent 106 timestamp pings to each destination, spoofing as each of the remaining PlanetLab hosts in turn. Of these, 63 failed to receive any responses to either spoofed or non-spoofed probes; they are completely stuck behind filters or were not working. For the remaining 43 hosts, Figure 6 shows how many destinations each host receives responses from, both without and with spoofing. Our results show that some sites benefit significantly. In reverse traceroute's timestamp measurements, whenever the source does not receive a response, we retry with 5 spoofers. Since some vantage points have filter-free paths to most destinations, we use the 5 best overall, rather than choosing per destination. For the nodes that experience widespread filtering, spoofing enables a significant portion to still use timestamps as part of reverse traceroute. As we show in Section 5.2, our timestamp techniques help the overall coverage of the tool.

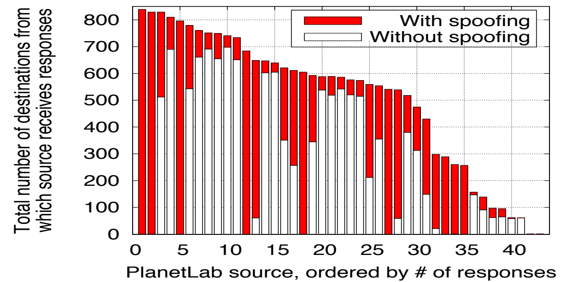


Figure 6: For 43 PlanetLab nodes, the number of destinations (out of 1000) from which the node receives timestamp responses. The graph shows the total number of unique destinations when sending the ping directly and then when also using 10 spoofers. The nodes are ordered by the total number of responding destinations. Other PlanetLab sites were tested but are not included in the graph: 103 did not experience significant filtering and 63 did not receive responses even with spoofing.

4.5 Selective Use of Vantage Points for Scalability

With spoofed RR, only nearby spoofers can find reverse hops, since each packet includes only 9 slots. Because many routers rate limit after only a few probes, we cannot send from many vantage points at once, in the hopes that one will prove close enough – the router might drop the probes from the VPs within range. Our goal is to determine which VPs are likely to be near a router before we probe it. Because Internet routing is generally based on the destination's prefix, a VP close to one address in a prefix is likely close to other addresses in the prefix.

Each day, we harvest the set of router IP addresses seen in the Internet atlas gathered by iPlane on the previous day and supplement the set with a recent list of pingable addresses [17]. Each day, every VP issues a record route ping to every address in the set. For each address, we determine the set of VPs that were near enough to discover reverse hops. We use this information in two ways during a reverse traceroute. First, if we encounter one of the probed addresses, we know the nearest VP to use. Second, if we encounter a new address, the offline probes provide a hint: the group of VPs within range of some address in the same prefix. Selecting the minimal number of vantage points to use from this group is an instance of the well known set cover optimization problem. We use the standard greedy algorithm to decide which VPs to use for a prefix, ordered by the number of additional addresses they cover within the prefix.

For a representative day, Figure 7 shows the coverage we achieve at given numbers of VPs per prefix. Our system determines the covering VPs for all prefixes, but the graph only includes prefixes for which we probed at least 15 addresses, as it is trivial to cover small prefixes. We see that, for most prefixes, we only need a small number of VPs. For example, in the median case, a single VP suffices for over 95% of addresses in the prefix, and we rarely need more than 4 VPs to cover the entire prefix.

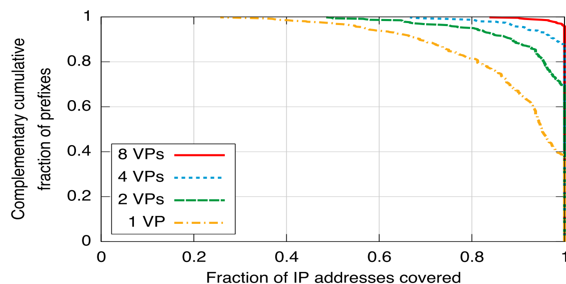


Figure 7: For prefixes in which iPlane observed $\geq$ 15 addresses, the fraction of the addresses for which we can find reverse hops using RR probes from a given number of vantage points per prefix. Note that we only include addresses within range of at least one vantage point. Prefixes with few addresses are trivial to cover using a small number of vantage points, so the graph excludes them to clearly show that we still only need a small number for most prefixes.

5 Evaluation

To test how well our reverse traceroute system can determine reverse paths, we consider evaluation settings that allow us to compare a reverse traceroute from *D* to *S* to a direct traceroute from *D* to *S*. A complete evaluation of the accuracy of our technique would require ground truth information about the path back from the destination. Obviously, we lack ground truth for the Internet, but we use two datasets, one PlanetLab-based and one using public traceroute servers, in which we can compare to a traceroute from *D*. For the reverse traceroute, we assume we do not control *D* and must measure the path using the techniques described in this paper. For the direct traceroute, we do control *D* and can simply issue a standard traceroute from *D* to *S*.

In the PlanetLab set, we employ as sources a host at each of 11 PlanetLab sites chosen at random from the spoofing nodes. As destinations, we use one host at each of the 200 non-spoofing PlanetLab sites that were working. Although such a set is not representative of the entire Internet, the destinations includes hosts in 35 countries. The measured reverse paths traversed 13 of the 14 transit-free commercial ISPs. Previous work observed route load balancing in many such networks [2], providing a good test for our techniques.

In the traceroute server set, we employ as sources a host at 10 of the same PlanetLab sites (one had gone down in the meantime). The 1200 traceroute servers we utilize belong to 186 different networks (many of which offer multiple traceroute servers with different locations). For each source, we choose a traceroute server at random from each of the 186 networks. We then issue a traceroute from the server to the PlanetLab source. Because in many cases we do not know the IP address of the traceroute server, we use the first hop along its path as the destination in our reverse traceroute measurements. When measuring a reverse traceroute from this destination back to the source, we exclude from our system all traceroute servers in the same network, to avoid providing our system with such similar paths as to make its task trivial.

5.1 Accuracy

How similar are the hops on a reverse traceroute to a direct traceroute from the destination back to the source? For the PlanetLab dataset, the *RevTR* line in Figure 8 depicts the fraction of hops seen on the direct traceroute that are also seen by reverse traceroute. Figure 9 shows the same for the traceroute server dataset. Note that, outside of this experimental setting, we would not normally have access to the direct traceroute from the destination. Using alias data from topology mapping projects [20, 22, 35] and aliases we discover using a state-of-the-art technique [4], we consider a traceroute hop and a reverse traceroute hop to be the same if they are aliases for the same router. We need to use alias information because the techniques may find different IP addresses on the same router [36]. For example, traceroute generally finds the ingress interface, whereas record route often returns the egress or loopback address. However, alias resolution is an active and challenging research area, and faulty aliases in the data we employ could lead us to falsely label two hops as equivalent. Conversely, missing aliases could cause us to label as different two interfaces on the same router. Because the alias sets we use are based on measurements from PlanetLab or similar vantage points, we likely have more complete alias data for our PlanetLab dataset than for our traceroute server dataset.

Using the available alias data, we find that the paths measured by our technique are quite similar to those seen by traceroute. In the median (mean) case, we measure 87% (83%) of the hops in the traceroute for the PlanetLab dataset. For the traceroute server dataset, we measure 75% (74%) of the hops in the direct traceroute, but 28% (29%) of the the hops discovered by reverse traceroute do not appear in the corresponding traceroute.

The figures also compare reverse traceroute to other potential ways of estimating the reverse path. All techniques used the same alias resolution. Researchers often (sometimes implicitly) assume symmetry, and operators likewise rely on forward path measurements when they need reverse ones. The *Guess Fwd* lines depict how many of the hops seen in a traceroute from *R* to *S* are also seen in a traceroute from *S* to *R*. In the median (mean) case, the forward path shares 38% (39%) of the reverse path's hops for the PlanetLab dataset and 40% (43%) for the traceroute server dataset. Another approach would be to measure traceroutes from a set of vantage points to the source. Using iPlane's PlanetLab

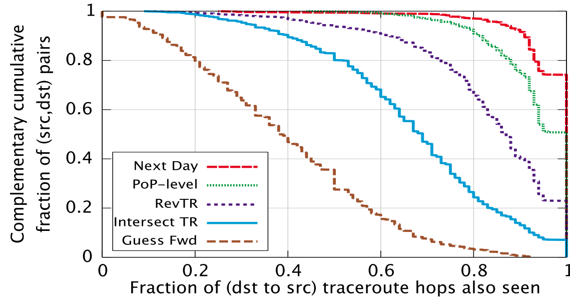


Figure 8: For reverse traceroute and techniques for approximating the reverse path, the fraction of hops on a direct traceroute from the destination to the source that the technique also discovers. Uses our PlanetLab dataset (reverse paths from 200 PlanetLab destinations back to 11 PlanetLab sources). [Key labels are in the same top-to-bottom order as the lines.]

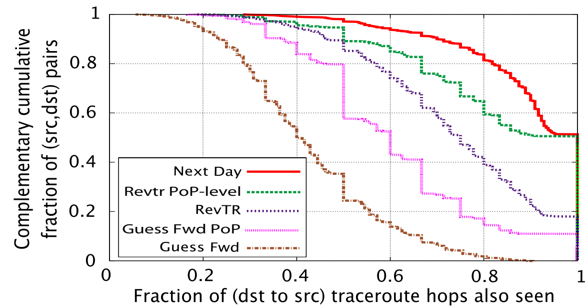


Figure 9: For reverse traceroute and techniques for approximating the reverse path, the fraction of hops on a direct traceroute from the destination to the source that the technique also discovers. Our traceroute server dataset includes reverse paths from servers in 186 networks back to 10 PlanetLab sources. [Key labels are in the same top-to-bottom order as the lines.]

and traceroute server measurements, the *Intersect TR* line in Figure 8 shows how well this approach works, by assuming the reverse path is the same as the forward path until it intersects one of the traceroutes [4]. No system today performs this type of path intersection on-demand for users. In the median (mean) case, this traceroute intersection shares 69% (67%) of the actual traceroute's hops. This result suggests that simply having a few hundred or thousand traceroute vantage points is not enough to reliably infer reverse paths; our system uses our novel measurement techniques to build off these traceroutes and achieve much better results.

What are the causes of differences between a reverse traceroute and a directly measured traceroute? Although it is common to think of the path given by traceroute as the true path, in reality it is also subject to measurement error. In this section, we discuss reasons traceroute and reverse traceroute may differ from each other and/or from the true path taken.

Assumptions of symmetry: When reverse traceroute is unable to identify the next reverse hop, we resort to assuming that hop is symmetric. These assumptions may lead to inaccuracies. For the PlanetLab dataset, if we consider only cases when we measure a complete path without assuming symmetry, in the median (mean) case reverse traceroute matches 93% (90%) of the traceroute alias-level hops. Similarly, for the traceroute server dataset, in the median (mean) case reverse traceroute finds 83% (81%) of the traceroute hops. We discuss how often we have to assume symmetry in Section 5.2.

Incomplete alias information: Many of the differences between the paths found by reverse traceroute and traceroute are due to missing alias information. Most alias-pair identification relies on sending probes to the two IP addresses and comparing the IP-IDs of the responses. For the PlanetLab dataset, of all the *missing* addresses seen in a traceroute that are not aliases of any hop in the corresponding reverse traceroute, 88% do not allow for such alias resolution [4]. Similarly, of all *extra* addresses seen in some reverse traceroute that are not aliases of any hop in the corresponding reverse traceroute, 82% do not allow for alias resolution. For the traceroute server dataset, 75% of the missing addresses and 74% of the extra ones do not allow it. Even for addresses that do respond to alias techniques, our alias sets are likely incomplete.

In these cases, it is possible or even likely that the two measurement techniques observe IP addresses that appear different but are in fact aliases of the same router. To partially examine how this lack of alias information limits our comparison, we use iPlane's Point-of-Presence (PoP) clustering, which maps IP addresses to PoPs defined by (AS,city) pairs [22]. For many applications such as diagnosis of inflated latencies [21], PoP level granularity suffices. iPlane has PoP mappings for 71% of the missing addresses in the PlanetLab dataset and 77% of the extra ones. For the traceroute server dataset, for which we have less alias information, it has mappings for 79% of the missing addresses and 86% of the extra ones. Figures 8 and 9 include *PoP-Level* lines showing the fraction of traceroute hops seen by reverse traceroute, if we consider PoP rather than router-alias-level comparison. In the median case, the reverse traceroute includes all the traceroute PoPs in both graphs (mean=94%, 84%). If reverse traceroute were measuring a different path than traceroute, then one would expect PoP-level comparisons to differ about as much as alias-level ones. The implication of the measured PoP-level similarity is that, when traceroute and reverse traceroute differ, they usually differ only in which router or interface in a PoP the path traverses. As a point of comparison, Figure 9 includes a PoP-level version of the *Guess Fwd* line; in the median (mean) case, it includes only 60% (61%) of the PoPs; the paths are quite asymmetric even at the PoP granularity.

[4]We omit the line from Figure 9 to avoid clutter.

Load-balancing and contemporaneous path changes: Another measurement artifact is that traceroute and reverse traceroute may uncover different, but equally valid, paths, either due to following different load-balanced options or due to route changes during measurement. To partly capture these effects, the *Next Day* lines in Figure 8 and 9 compare how many of the traceroutes' hops are also on traceroutes issued the following day. For the PlanetLab dataset, 26% of the paths exhibit some router-level variation from day to day. For the traceroute dataset, 49% of paths changed at the router level and (not shown in the graph) 15% changed at the PoP-level. In a loose sense, these results suggest an upper bound – even the same measurement issued at a different time may yield a different path.

Hidden or anonymous routers: Previous work comparing traceroute to record route paths found that 16% of IP addresses appear with RR but do not appear in traceroutes [36, 35]. *Hidden* routers, such as those inside some MPLS tunnels, do not decrement TTL. *Anonymous* routers decrement TTL but do not send ICMP replies, appearing as '*' in traceroute.

Exceptional handling of options packets: Packets with IP options are generally diverted to a router's route processor and may be processed differently than on the line card. For example, previous work suggests that packets with options are load-balanced per-packet, rather than per-flow [35].

An additional source of discrepancies between the two techniques is that traceroute and reverse traceroute make different assumptions about routing. Our techniques assume destination-based routing – if the path from *D* to *S* passes through *R*, from that point on it is the same as *R*'s path to *S*. An options packet reports only links it actually traversed. With traceroute, on the other hand, a different packet uncovers each hop, and it assumes that if *R1* is at hop *k* and *R2* is at hop *k+1*, then there is a link *R1–R2*. However, it does not make the same assumption about destination routing, as each probe uses *(S,D)* as the source and destination. These differing assumptions lead to two more causes of discrepancies between a traceroute and a reverse traceroute:

Traceroute inferring false links: Although we use the Paris traceroute technique for accurate traversal of flow-based load balancers, it can still infer false links in the case of packet-based balancing [2]. These spurious links appear as discrepancies between traceroute and reverse traceroute, but in reality show a limitation of traceroute.

Exceptions to destination-based routing: With many tunnels, an option-enabled probe will see the entire tunnel as a single hop. With certain tunnels, however, our assumption of destination-based routing may not hold. When probed directly, an intermediate router inside the tunnel may use a path to the destination other than the one that continues through the tunnel. To partly capture the degree of this effect, we perform a study that eliminates it. From each of the 200 PlanetLab nodes used as destinations in this section, we issue both a traceroute and an RR ping to each of the 11 used as sources, so the RR ping will have the same source and destination as the traceroute (unlike with reverse traceroute's RR probes to intermediate routers). Since the RR slots may fill up before the probe reaches the destination, we only check if the traceroute matches the portion of the path that appears in the RR. After alias resolution, the median fraction of RR hops seen in the corresponding traceroute is 0.67, with the other factors described in this section accounting for the differences. This fraction is 0.2 lower than that for reverse traceroute, showing the difficulty in matching RR hops to traceroute hops.

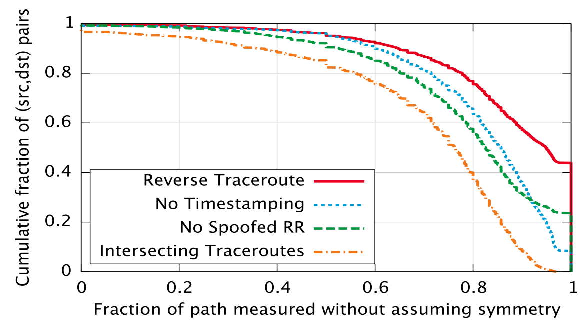


Figure 10: For the PlanetLab dataset, the fraction of reverse path hops measured, rather than assumed symmetric. The graph includes results with subsets of the reverse traceroute techniques.

5.2 Coverage

In Section 5.1, we noted that our paths are more accurate when our techniques succeed in measuring the entire path without having to fall back to assuming a link is symmetric. As seen in Figure 8, if forced to assume the entire path is symmetric, in the median case we would discover only 39% of the hops on a traceroute. In this section, we investigate how often our techniques are able to infer reverse hops, keeping us from reverting to assumptions of symmetry. Using the PlanetLab dataset, Figure 10 presents the results for our complete technique, as well as for various combinations of the components of our technique. The metric captured in the graph is the fraction of hops in the reverse traceroute that were measured, rather than assumed symmetric.

Reverse traceroute finds most hops without assuming symmetry. In the median path in the PlanetLab dataset, we measure 95% of hops (mean=87%), and in 80% of cases we are able to measure at least 78% of the path without assumptions of symmetric. By contrast, the traceroute intersection estimation technique from Figure 8 assumes in the median that the last 25% of the

path is symmetric (mean=32%). Although not shown in the graph, the results are similar for the traceroute server dataset – in the median case, reverse traceroute measures 95% of hops (mean=92%) without assuming symmetry.

The graph also depicts the performance of our technique if we do not use spoofed record route pings or do not use timestamping. In both cases, the performance drops off somewhat without both probing methods.

5.3 Overhead

We assess the overhead of our technique using the traceroute server dataset from Section 5.1, comparing the time and number of probes required by our system to those required by traceroute. The median (mean) time for one of the 10 PlanetLab sites to issue a traceroute to one of the 186 traceroute servers was 5 seconds (9.4 seconds). Using our current system, as available on `http://revtr.cs.washington.edu` and described in Section 4, the median (mean) time to measure a reverse traceroute was 41 seconds (116.0 seconds), including the time to send an initial forward traceroute (to determine if the destination is reachable and to present a round-trip path at the end). We have not yet pursued improving this aspect of the system. In the future, we will investigate lowering this delay by setting more aggressive timeouts for flaky PlanetLab vantage points, by cutting down on the communication overhead, and by attempting to adapt our probing rate to the rate limit of the particular target.

For each reverse traceroute measurement, our system sends the initial forward traceroute and a number of options-enabled ping packets, some of which may be spoofed. In cases when it is unable to determine the next reverse hop, it sends a forward traceroute and assumes the last hop is symmetric. In addition, we require traceroutes to build an atlas of paths to the source, and we use ongoing background mapping to identify adjacencies and to determine which vantage points are within RR-range of which prefixes.

If we ignore the probing overhead of the traceroute atlas and the mapping, in the median case, the only traceroute required is the initial one (mean=1.2 traceroutes). In the median (mean) case, a reverse traceroute requires 2 (2.6) record route packets, plus an additional 9 (21.2) spoofed RR packets. The median (mean) number of non-spoofed timestamp packets is 0 (5.1), and the median (mean) number of spoofed timestamp packets is also 0 (6.5). The median (mean) total number of options packets sent is 13 (35.4). As a point of comparison, traceroute uses around 45 probe packets on average, 3 for each of around 15 hops. At the end of a reverse traceroute, we also send 3 pings to each hop to measure latency. So, ignoring the creation of the various atlases, reverse traceroute generally requires roughly 2-3x more packets than traceroute.

IP address	AS name	Location	RTT
132.170.3.1	UCF	Orlando, FL	0ms
198.32.155.89	FloridaNet	Orlando, FL	0ms
198.32.132.64	FloridaNet	Jacksonville, FL	3ms
198.32.132.19	Cox Comm.	Atlanta, GA	9ms
68.1.0.221	Cox Comm.	Ashburn, VA	116ms
216.52.127.8	Internap	Washington, DC	35ms
66.79.151.129	Internap	Washington, DC	26ms
66.79.146.202	Internap	Washington, DC	24ms
66.79.146.241	Internap	Miami, FL	53ms
66.79.146.129	Internap	Seattle, WA	149ms

Table 1: Traceroute giving forward path from University of Central Florida to *66.79.146.129*.

The atlases represent the majority of our probe overhead. However, in many circumstances these atlases can be reused and/or optimized for performance. For example, if the source requests reverse paths for multiple destinations within a short period of time [45], we can reuse the atlas. As an optimization, we may need to only issue those traceroutes that are likely to intersect [22], and we can use known techniques to reduce the number of probes to generate the atlas [11]. We borrow the adjacency information needed for our timestamp probes from an existing mapping service [22]. To determine which spoofing vantage points are likely within record route range of a destination, we regularly issue probes from every spoofer to a set of addresses in each prefix. In the future, we plan to investigate if we can reduce this overhead by probing only a single address within each prefix.

6 Applications of Reverse Traceroute

We believe many opportunities exist for improving systems and studies using reverse traceroute. We next discuss three such examples of how reverse traceroute can be used in practice. We intend these sections to illustrate a few ways in which one can apply our tool; they are not complete studies of the problems.

6.1 Case study of debugging path inflation

Large content providers attempt to optimize client performance by replicating their content across a geographically distributed set of servers. A client is then redirected to the server to which it has minimum latency. Though this improves the performance perceived by clients, it can still leave room for improvement. Internet routes are often inflated [37], which can lead to round-trip times from a client to its nearest server being much higher than what they should be given the server's proximity. Using Google as an example, 20% of client prefixes experience more than 50ms latency over the minimum latency to the prefix's geographical region. Google wants a way to identify which AS is the cause of inflation, but it is hindered by the lack of information about reverse paths back to their servers from clients [21].

IP address	AS name	Location	RTT
66.79.146.129	Internap	Seattle, WA	148ms
66.79.146.225	Internap	Seattle, WA	141ms
137.164.130.66	TransitRail	Los Angeles, CA	118ms
137.164.129.15	TransitRail	Los Angeles, CA	118ms
137.164.129.34	TransitRail	Palo Alto, CA	109ms
137.164.129.2	TransitRail	Seattle, WA	92ms
137.164.129.11	TransitRail	Chicago, IL	41ms
137.164.131.165	TransitRail	Ashburn, VA	23ms
132.170.3.1	UCF	Orlando, FL	0ms
132.170.3.33	UCF	Orlando, FL	0ms

Table 2: Reverse traceroute giving reverse path from *66.79.146.129* back to University of Central Florida. The circuitous reverse path explains the huge RTT jump between the last two hops on the forward path. The third hop, 137.164.130.66 (internap-peer.lsanca01.transitrail.net), is a peering point between Internap and TransitRail in L.A.

As an illustration, we used reverse traceroute to diagnose an example of path inflation. We measured the RTT on the path from the PlanetLab node at the University of Central Florida to the IP address *66.79.146.129*, which is in Seattle, to be 149ms. Table 1 shows the forward path returned by traceroute, annotated with the locations of intermediate hops inferred from their DNS names. The path has some circuitousness going from Orlando to Washington via Ashburn and then returning to Miami. But, that does not explain the steep rise in RTT from 53ms to 149ms on the last segment of the path, because a hop from Miami to Seattle is expected to only add 70ms to the RTT[5].

To investigate the presence of reverse path inflation back from the destination, we determined the reverse path using reverse traceroute. Table 2 illustrates the reverse path, which is noticeably circuitous. Starting from Seattle, the path goes through Los Angeles and Palo Alto, and then returns to Seattle before reaching the destination via Chicago and Ashburn. We verified with a traceroute from a PlanetLab machine at the University of Washington that TransitRail and Internap connect in Seattle, suggesting that the inflation is due to a routing misconfiguration. Private communication with an operator at one of the networks confirmed that the detour through Los Angeles was unintentional. Without the insight into the reverse path provided by reverse traceroute, such investigations would not be possible by the organizations most affected by inflated routes.

6.2 Topology discovery

Studies of Internet topology rely on the set of available vantage points and data collection points. With a limited number available, routing policies bias what researchers measure. As an example, with traceroute alone, topology discovery is limited to measuring forward paths from a few hundred vantage points to each other and to other destinations. Reverse traceroute allows us to expose many peer-to-peer links invisible to traceroute.

[5]Interestingly, the latency to Ashburn seems to also be inflated on the reverse path.

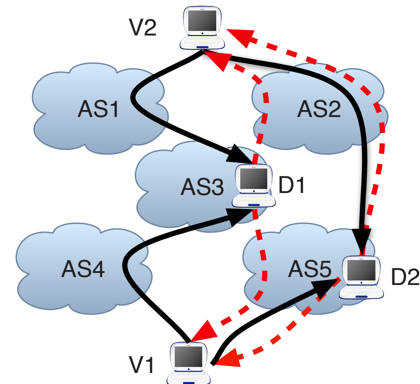


Figure 11: Example of our techniques aiding in topology discovery. With traceroutes alone, V1 and V2 can measure only the forward (solid) paths. If V2 is within 8 hops of D1, a record route ping allows it to measure the link AS3-AS2, and a record route ping spoofed as V1 allows it to measure AS3-AS5.

Figure 11 illustrates one way in which our techniques can uncover links. Assume that *AS3* has a peer-to-peer business relationship with the other ASes. Because an AS does not want to provide free transit, most routes will traverse at most one peer-to-peer link. In this example, traffic will traverse one of *AS3*'s peer links only if it is sourced or destined from/to *AS3*. *V1*'s path to *AS3* goes through *AS4*, and *V2*'s path *AS3* goes through *AS1*. Topology-gathering systems that rely on traceroute alone [22, 1, 34] will observe the links *AS1-AS3*, *AS4-AS3*, and *AS2-AS5*. But, they will never traverse *AS3-AS5*, or *AS3-AS2*, no matter what destinations they probe (even ones not depicted). *V2* can never traverse *AS1-AS3-AS5* in a forward path (assuming standard export policies), because that would traverse two peer-to-peer links. However, if *V2* is within 8 hops of *D1*, then it can issue a record-route ping that will reveal *AS3-AS2*, and a spoofed record route (spoofed as *V1*) to reveal *AS3-AS5* [6].

Furthermore, even services like RouteViews [27] and RIS [33], with BGP feeds from many ASes, likely miss these links. Typical export policies mean that only routers in an AS or its customers see the AS's peer-to-peer links. Since RouteViews has vantage points in only a small percentage of the ASes lower in the AS hierarchy, it does not see most peer links [29, 16].

To demonstrate how reverse traceroute can aid in topology mapping, we apply it to a recent study on mapping Internet exchange points (IXPs) [3]. That study used existing measurements, novel techniques, and thousands of traceroute servers to provide IXP peering matrices that were as complete as possible. As part of the

[6]Note that, because we only query for hops already known to be adjacent, our timestamp pings are not useful for topology discovery.

study, the researchers published the list of ASes they found to be peering at IXPs, the IXPs at which they peered, and the IP addresses they used in those peerings.

We measured the reverse paths back from those IP addresses to all PlanetLab sites. We discovered 9096 IXP peerings (triples of the two ASes and the IXP at which they peer) that are not in the published dataset, adding an additional 16% to the 58,534 peerings in their study. As one example, we increased the number of peerings found at the large London LINX exchange by 19%. If we consider just the ASes observed peering and not which IXP they were seen at, we found an additional 5057 AS links not in the 51,832 known IXP AS links, an increase of 10%. Of these AS links, 1910 do not appear in either traceroute [22] or BGP [40] topologies – besides not being known as IXP links, we are discovering links not seen in some of the most complete topologies available. Further, of the links in both our data and UCLA's BGP topology, UCLA classifies 1596 as Customer-to-Provider links, whereas the fact that we observed them at IXPs strongly suggests they are Peer-to-Peer links. Although the recent IXP study was by far the most exhaustive yet, reverse traceroute provides a way to observe even more of the topology.

6.3 Measuring one-way link latency

In addition to measuring a path, traceroute measures a round-trip latency for each hop. Techniques for geolocation [42, 18], latency estimation [22], and ISP comparisons [25], among others, depend on link latency measurements obtained by subtracting the RTT to either endpoint, then halving the difference (possibly with a filter for obviously wrong values). This technique should yield fairly accurate values if routes traverse the link symmetrically. However, previous work found that 88-98% of paths are asymmetric [15] resulting in substantial errors in link latency estimates [39]. More generally, the inability to isolate individual links is a problem when using network tomography to infer missing data – tomography works best only when the links are traversed symmetrically or when one knows both the forward and reverse paths traversed by the packets [10, 6].

A few alternatives exist for estimating link latencies but none are satisfactory. Rocketfuel infers link weights used in routing decisions [23], which may or may not reflect latencies. The geographic locations of routers provide an estimate of link latency, but such information may be missing, wrong, or outdated, and latency does not always correspond closely to geographic distance [18].

In this section, we revisit the problem of estimating link latencies since we now have a tool that provides reverse path information to complement traceroute's forward path information. Given path asymmetry, the reverse paths from intermediate routers likely differ from

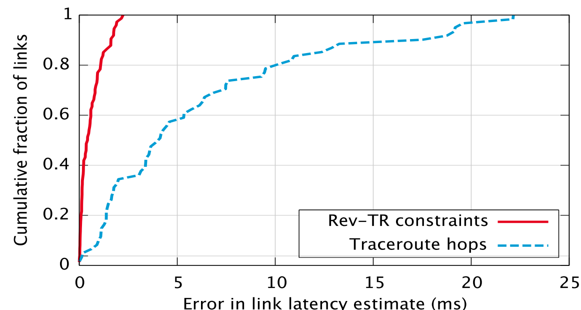


Figure 12: Error in estimating latencies for Sprint inter-PoP links. For each technique, we only include links for which it provided an estimate: 61 of 89 links using traceroute, and 74 of 89 using reverse traceroute. Ground truth reported only to 0.5ms granularity.

the end-to-end traceroutes in both directions. Without reverse path information from the intermediate hops back to the hosts, we cannot know which links a round-trip latency includes. Measurements to endpoints and intermediate hops yield a large set of paths, which we simplify using IP address clustering [22]. We then generate a set of linear constraints: for any intermediate hop R observed from a source S, the sum of the link latencies on the path from S to R plus the sum of the link latencies on the path back from R must equal the round-trip latency measured between S and R. We then solve this set of constraints using least-squares minimization, and we also identify the bound and free variables in the solution. Bound variables are those sufficiently constrained for us to solve for the link latencies, and free variables are those that remain under constrained.

We evaluate our approach on the Sprint backbone network by comparing against inter-PoP latencies Sprint measures and publishes [38]. We consider only the directly connected PoPs and halve the published round-trip times to yield link latencies we use as ground truth, for 89 links between 42 PoPs. We observe 61 of the 89 links along forward traceroutes and 79 with reverse traceroute. We use these measurements to formulate constraints on the inter-PoP links, based on round-trip latencies measured from PlanetLab nodes to the PoPs using ping. This set of constraints allows us to solve for the latencies of 74 links, leaving 5 free and 10 unobserved.

As a comparison point, we use a traditional method for estimating link latency from traceroutes [22]. For each forward traceroute that traverses a particular Sprint link, we sample the link latency as half the difference between the round-trip delay to either end, then estimate the link latency to be the median of these samples across all traceroutes. Figure 12 shows the error in the latency estimates of the two techniques, compared to the published ground truth. Our approach infers link latencies with errors from 0ms to 2.2ms for the links, with a me-

dian of 0.4ms and a mean of 0.6ms. Because Sprint reports round-trip delays with millisecond granularity, the values we use for ground truth have 0.5ms granularity, so our median "error" is within the granularity of the data. The estimation errors using the traditional traceroute method range from 0ms to 22.2ms, with a median of 4.1ms and a mean of 6.2ms – $10x$ our worst-case, median, and mean errors. Based on this initial study of a single large network for which we have ground-truth, using reverse traceroute to generate and solve constraints yields values very close to the actual latencies, whereas the traditional approach does not.

7 Related work

Measurement techniques: Previous work concluded that too many paths dropped packets with IP options for options to form the basis of a system [13]. The Passenger and DisCarte projects, however, showed that the record route option, when set on traceroute packets, reduces false links, uncovers more routers, and provides more complete alias information [36, 35]. Hubble demonstrated the use of spoofed packets to probe a path in one direction without having to probe the other [19], but it does not determine the routers along the reverse path. Addressing this limitation in Hubble was part of the original motivation for this work.

The contributions of these various projects is in how they employ existing IP techniques – options and spoofing – towards useful ends. Our work employs the same IP techniques in new ways. We demonstrate how spoofing with options can expose reverse paths. Whereas Passenger and DisCarte used RR to improve forward path information, we use RR in non-TTL-limited packets to measure reverse paths. As far as we are aware, our work is the first to productively employ the timestamp option.

Techniques for inferring reverse path information: Various earlier techniques proposed methods for inferring limited reverse path information. Before such packets were routinely filtered, one study employed loose source-routing [31] to measure paths from numerous remote sites. Other interesting work used return TTL values to estimate reverse routing maps towards sources; however, the resulting maps contained less than half the actual links, as well as containing multiple paths from many locations [7]. PlanetSeer [43] and Hubble [19] included techniques for isolating failures to either the forward or reverse path; neither system, however, can give information about where on a reverse path the failure occurs. Netdiff inferred path asymmetry in cases where hop counts differ greatly in the two directions [25]; however, as our example in Section 6.1 shows, very asymmetric paths can have the same hop count. Tulip used ICMP timestamps (not the IP timestamp option we use) and other techniques to identify reordering and loss along either the forward or reverse path [24].

Systems that would benefit from reverse path information: Many systems seem well-designed to make use of reverse path information, but, lacking it, make various substitutions or compromises. We mention some recent ones here. Geolocation systems use delay and path information to constrain the position of targets [14, 18, 42], but, lacking reverse path data, are under constrained. iPlane shows that knowledge of a few traceroutes from a prefix greatly improves path predictions [22], but lacks vantage points in most. iSpy attempted to detect prefix hijacks using forward-path traceroutes, yet the signature it looked for is based on the likely pattern of reverse paths [46]. Similarly, intriguing recent work on inferring topology through passive observation of traffic bases its technique on an implicit assumption that the hop counts of forward and reverse paths are likely to be the same [12]. Similarly, systems for network monitoring often assume path symmetry [8, 25]. All these efforts can potentially benefit from the work described here.

8 Conclusion

Although widely-used and popular, traceroute is fundamentally limited in that it cannot measure reverse paths. This limitation leaves network operators and researchers unable to answer important questions about Internet topology and performance. To solve this problem, we developed a reverse traceroute system to measure reverse paths from arbitrary destinations back to the user. The system uses a variety of methods to incrementally build a path back from the destination hop-by-hop, until it reaches a known baseline path. We believe that our system makes a strong argument for both the IP timestamp option and source spoofing as important measurement tools, and we hope that PlanetLab and ISPs will consider them valuable components of future measurement testbeds.

Our reverse traceroute system is both effective – in the median case finding all of the PoPs seen by a direct traceroute along the same path – and useful. The tool allows operators to conduct investigations impossible with existing tools, such as tracking down path inflation along a reverse route. Many operators seem to view reverse traceroute as a useful tool – based on the results presented in this paper, we received requests to help us test the tool and offers of spoofing vantage points, including hosts at all the PoPs of an international backbone network. The system's probing methods have also proved useful for topology mapping. In illustrative examples, we demonstrated how our system can discover more than a thousand peer-to-peer links invisible to both BGP route collectors and to traceroute-based mapping

efforts, as well as how it can be used to accurately measure the latency of backbone links. We believe the accuracy and coverage of the tool will only improve as we add additional vantage points. A demo of our tool is available at http://revtr.cs.washington.edu.

Acknowledgments

We gratefully acknowledge John Byers, our shepherd, and the anonymous NSDI reviewers for their valuable feedback on earlier versions of this paper. Adam Bender, Rob Sherwood, Ricardo Oliveira, Brice Augustin, and Balachander Krishnamurthy provided access to and help with their tools and data. This work was funded partially by Google, Cisco, and NSF grants CNS-0905568 and MRI-0619836. We are thankful for their support.

References

[1] Archipelago measurement infrastructure. http://www.caida.org/projects/ark/.

[2] B. Augustin, X. Cuvellier, B. Orgogozo, F. Viger, T. Friedman, M. Latapy, C. Magnien, and R. Teixeira. Avoiding traceroute anomalies with Paris traceroute. In *IMC*, 2006.

[3] B. Augustin, B. Krishnamurthy, and W. Willinger. IXPs: Mapped? In *IMC*, 2009.

[4] A. Bender, R. Sherwood, and N. Spring. Fixing Ally's growing pains with velocity modeling. In *IMC*, 2008.

[5] R. Beverly, A. Berger, Y. Hyun, , and k claffy. Understanding the efficacy of deployed Internet source address validation filtering. In *IMC*, 2009.

[6] T. Bu, N. Duffield, F. Presti, and D. Towsley. Network tomography on general topologies. In *SIGMETRICS*, 2002.

[7] H. Burch. *Measuring an IP network in situ*. PhD thesis, CMU, 2005.

[8] Y. Chen, D. Bindel, H. Song, and R. H. Katz. An algebraic approach to practical and scalable overlay network monitoring. In *SIGCOMM*, 2004.

[9] D. Choffnes and F. E. Bustamante. Taming the torrent: A practical approach to reducing cross-ISP traffic in peer-to-peer systems. In *SIGCOMM*, 2008.

[10] M. Coates, A. Hero, R. Nowak, and B. Yu. Internet tomography. *IEEE Signal Processing Magazine*, 2002.

[11] B. Donnet, P. Raoult, T. Friedman, and M. Crovella. Efficient algorithms for large-scale topology discovery. In *SIGMETRICS*, 2005.

[12] B. Eriksson, P. Barford, and R. Nowak. Network discovery from passive measurements. In *SIGCOMM*, 2008.

[13] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica. IP options are not an option. Technical report, EECS Department, University of California, Berkeley, 2005.

[14] B. Gueye, A. Ziviani, M. Crovella, and S. Fdida. Constraint-based geolocation of Internet hosts. *IEEE/ACM TON*, 2006.

[15] Y. He, M. Faloutsos, S. Krishnamurthy, and B. Huffaker. On routing asymmetry in the Internet. In *Autonomic Networks Symposium in Globecom*, 2005.

[16] Y. He, G. Siganos, M. Faloutsos, and S. Krishnamurthy. A systematic framework for unearthing the missing links. In *NSDI*, 2007.

[17] Internet address hitlist dataset, PREDICT ID USC LANDER internet_address_hitlist_it28wbeta20090914. http://www.isi.edu/ant/lander.

[18] E. Katz-Bassett, J. P. John, A. Krishnamurthy, D. Wetherall, T. Anderson, and Y. Chawathe. Towards IP geolocation using delay and topology measurements. In *IMC*, 2006.

[19] E. Katz-Bassett, H. V. Madhyastha, J. P. John, A. Krishnamurthy, D. Wetherall, and T. Anderson. Studying black holes in the Internet with Hubble. In *NSDI*, 2008.

[20] K. Keys, Y. Hyun, and M. Luckie. Internet-scale alias resolution with MIDAR. Under preparation, 2010.

[21] R. Krishnan, H. V. Madhyastha, S. Srinivasan, S. Jain, A. Krishnamurthy, T. Anderson, and J. Gao. Moving beyond end-to-end path information to optimize CDN performance. In *IMC*, 2009.

[22] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane: An information plane for distributed services. In *OSDI*, 2006.

[23] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. Inferring link weights using end-to-end measurements. In *IMW*, 2002.

[24] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. User-level Internet path diagnosis. In *SOSP*, 2003.

[25] R. Mahajan, M. Zhang, L. Poole, and V. Pai. Uncovering performance differences among backbone ISPs with Netdiff. In *NSDI*, 2008.

[26] Measurement Lab website. http://www.measurementlab.net/.

[27] D. Meyer. RouteViews. http://www.routeviews.org.

[28] NANOG 45, 2009.

[29] R. Oliveira, D. Pei, W. Willinger, B. Zhang, and L. Zhang. In search of the elusive ground truth: The Internet's AS-level connectivity structure. In *SIGMETRICS*, 2008.

[30] Outages mailing list. http://isotf.org/mailman/listinfo/outages.

[31] J.-J. Pansiot and D. Grad. On routes and multicast trees in the Internet. *SIGCOMM CCR*, 1998.

[32] RIPE 58, 2009.

[33] RIPE RIS. http://www.ripe.net/ris/.

[34] Y. Shavitt and E. Shir. DIMES: Let the Internet measure itself. *SIGCOMM CCR*, 2005.

[35] R. Sherwood, A. Bender, and N. Spring. Discarte: A disjunctive internet cartographer. In *SIGCOMM*, 2008.

[36] R. Sherwood and N. Spring. Touring the Internet in a TCP sidecar. In *IMC*, 2006.

[37] N. Spring, R. Mahajan, and T. Anderson. Quantifying the causes of path inflation. In *SIGCOMM*, 2003.

[38] Sprint IP network performance. https://www.sprint.net/performance/.

[39] R. A. Steenbergen. A practical guide to (correctly) troubleshooting with traceroute. In *NANOG 45*, 2009. http://www.nanog.org/meetings/nanog45/presentations/Sunday/RAS_traceroute_N45.pdf.

[40] UCLA Internet topology collection. http://irl.cs.ucla.edu/topology/.

[41] http://www.merit.edu/mail.archives/nanog/. North American Network Operators Group mailing list.

[42] B. Wong, I. Stoyanov, and E. G. Sirer. Octant: A comprehensive framework for the geolocalization of Internet hosts. In *NSDI*, 2007.

[43] M. Zhang, C. Zhang, V. Pai, L. Peterson, and R. Wang. PlanetSeer: Internet path failure monitoring and characterization in wide-area services. In *OSDI*, 2004.

[44] Y. Zhang, Z. M. Mao, and M. Zhang. Effective diagnosis of routing disruptions from end systems. In *NSDI*, 2008.

[45] Y. Zhang, V. Paxson, and S. Shenker. The stationarity of Internet path properties: Routing, loss, and throughput. *ACIRI Technical Report*, 2000.

[46] Z. Zhang, Y. Zhang, Y. C. Hu, Z. M. Mao, and R. Bush. iSpy: detecting IP prefix hijacking on my own. In *SIGCOMM*, 2008.

Seamless BGP Migration With Router Grafting

Eric Keller
Princeton University

Jennifer Rexford
Princeton University

Jacobus van der Merwe
AT&T Labs - Research

Abstract

Network operators are under tremendous pressure to make their networks highly reliable to avoid service disruptions. Yet, operators often need to change the network to upgrade faulty equipment, deploy new services, and install new routers. Unfortunately, changes cause disruptions, forcing a trade-off between the benefit of the change and the disruption it will cause. In this paper we present *router grafting*, where parts of a router are seamlessly removed from one router and merged into another. We focus on grafting a BGP session and the underlying link–from one router to another, or between blades in a cluster-based router. Router grafting allows an operator to rehome a customer with no disruption, compared to downtimes today measured in minutes. In addition, grafting a BGP session can help in balancing load between routers or blades, planned maintenance, and even traffic management. We show that grafting a BGP session is practical even with today's monolithic router software. Our prototype implementation uses and extends Click, the Linux kernel, and Quagga, and introduces a daemon that automates the migration process.

1 Introduction

In nature, grafting is where a part of one living organism (e.g., tissue from a plant) is removed and fused into another organism. In this paper, we apply this concept to routers to enable new network-management capabilities which allow network changes to be made with minimal disruption. We call this *router grafting*. With router grafting, we view routers in terms of their parts and enable splitting these parts from one router and merging them into another. This capability makes the view of the network a more fluid one where the topology can readily change, allowing operators to adapt their networks without disruption in the service offered to users. We envision router grafting to eventually be applicable to arbitrary subsets of router resources and/or protocols. However, in this paper we take the first step towards this vision by focusing how to "graft" a BGP session and the underlying link from one router to another.

1.1 A Case for Router Grafting

The ability to adapt the network is an essential component of network management. Unfortunately, today's routers and routing protocols make change difficult. Changes to the network cause disruption, forcing operators to weigh the benefit of making a change against the potential impact performing the change will have. For example, today, the basic task of rehoming a BGP session requires shutting down the session, reconfiguring the new router, restarting the session, and exchanging a large amount of routing information typically leading to downtimes of several minutes. Further complicating matters is the fact that service-level agreements with customers often prohibit events that result in downtime without receiving prior approval and scheduling a maintenance window. This hand-cuffs the operator. In this section we provide several motivating examples of why seamless migration is needed and why it would be desirable to do at the level of individual sessions.

Load balancing across blades in a cluster router: Today's high-end routers have modular designs consisting of many cards—processor blades for running routing processes and interface cards for terminating links—spread over multiple chassis. In essence, the router itself is a large distributed system. Load balancing is an important function in distributed systems, and routers are no exception—today's routers often run near their limits of processing capacity [1]. Unfortunately, routers are not built with load balancing in mind. A BGP session is associated with a routing process on a particular blade upon establishment, making it difficult to shift load to another blade. A common approach used with Web servers is to drain load by directing new requests to other servers

and waiting for existing requests to complete. Unfortunately, this technique is not applicable to routers, since routing sessions run indefinitely and unlike web services have persistent state. However, with the ability to migrate individual sessions, achieving better utilization of the router's processing capabilities is possible.

Rehoming a customer: An ISP homes a customer to a router based on geographic proximity and the availability of a router slot that can accommodate the customer's request [2]. However, this is done only at the time when a customer initiates service, based on the state of the network at that time. Rehoming might be necessary if the customer upgrades to a new service (such as multicast, IPv6, or advanced QoS or monitoring features) available only on a subset of routers. Rehoming is also necessary when an ISP upgrades or replaces a router and needs to move sessions from the old router to the new one. Customer rehoming involves moving the edge link—which can be done quickly because of recent innovations in layer-two access networks—as well as the BGP session.

Planned maintenance: Maintenance is a fact of life for network operators, yet, even though maintenance is planned in advance, little can be done to keep the router running. Consider a simple task of replacing a power supply. The best common practice is for operators to reconfigure the routing protocols to direct traffic away from that router and, once the traffic stops flowing, to take the router offline. Unfortunately, this approach only works for core routers within an ISP where alternate paths are available. At the edge of the network, an attractive alternative would be to graft all of the BGP sessions with neighboring networks to other routers to avoid disruptions in service. Migrating at the level of individual sessions is preferable to migrating all of the sessions and the routing processes as a group, since fine-grain migration allows multiple different routers to absorb only a small amount of extra load during the maintenance interval.

Traffic engineering: Traffic engineering is the act of reconfiguring the network to optimize the flow of traffic, to minimize congestion. Today, traffic engineering involves adjusting the routing-protocol parameters to coax the routers into computing new paths that better match the offered traffic, at the expense of transient disruptions during routing convergence. Router grafting enables a new approach to traffic engineering, where certain customers are rehomed to an edge router that better matches the traffic patterns. For example, if most of a customer's traffic leaves the ISP's network at a particular location, that customer could be rehomed closer to that egress point. In other words, we no longer need to consider the traffic matrix as fixed when performing traffic engineering—instead, we can change the traffic matrix to better match the backbone topology and routing by having traffic enter the network at a new location.

1.2 Challenges and Contributions

The benefits of router grafting are numerous. However, the design of today's routers and routing protocols make realizing router grafting challenging. Grafting a BGP session involves (i) migrating the underlying TCP connection, (ii) exchanging routing state, (iii) moving the routing-protocol configuration from one router to another, and (iv) migrating the underlying link. Ideally, all these actions need to be performed in a manner that is completely transparent (i.e., without involving the routers and operators in neighboring networks) and does not disrupt forwarding and routing (i.e., data packets are not dropped and routing adjacencies remain up).

Unfortunately, we cannot simply apply existing techniques for application-level session migration. Moving a BGP session to a different router changes the network topology and hence, the routing decisions at other routers. In particular, the remote end-point of the session must be informed of any routing changes—that is, any differences between the "best routes" chosen by the new and old homing points. Similarly, other routers in the ISP network need to change how they route toward destinations reachable through that remote end-point—they need to learn that these destinations are now reachable through the new homing location.

In addition, we cannot simply apply recently-proposed techniques for virtual-router migration [3], for two main reasons. First, the two physical routers may not be compatible—they may run different routing software (e.g., Cisco, Juniper, Quagga, or XORP). Second, we want to migrate and merge only a single BGP session, not the entire routing process, as many scenarios benefit from finer granularity. Instead, we view virtual-router migration as a complementary management primitive.

Fortunately, extending existing router software to support grafting requires only modest changes. The essential state that must be migrated is often well separated in the code. This makes it possible to export the state from one router and import it to another without much complexity. In this paper, we present an architecture for realizing router grafting and make the following contributions:

- Introduce the concept of router grafting, and realize an instance of it through BGP session migration. We demonstrate that BGP session migration can be performed in today's monolithic routing software, without much modification or refactoring of the code. Our fully-automated prototype router-grafting system is built by using and extending Click, Linux, and Quagga.

- Achieve transparency, where the remote BGP session end-point is not modified and is unaware migration is happening. We achieve this by bootstrap-

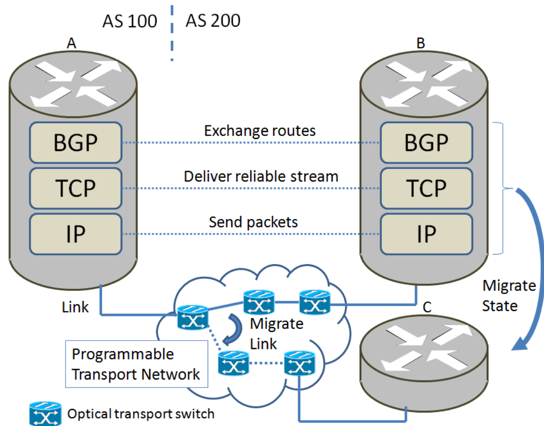


Figure 1: Migration protocol layers.

ping a routing session at the new homing location, with the old router emulating the remote end-point. The new homing point then takes over the role of the old router, sending the necessary routing updates to notify the remote end-point of routing changes.

- Introduce optimizations to nearly eliminate the impact of migration on other routers not directly involved in the migration. We achieve this by capitalizing on the fact that the routers already have much of the routing information they need, and that we know the identity of the old and new homing points.

- Describe an architecture where unplanned routing changes (such as link failures) during the grafting process do not affect correctness, and where packets are delivered successfully even during the migration. At worst, packets temporarily traverse a different path than the control plane advertises—a common situation during routing convergence.

The remainder of the paper is organized as follows. Section 2 discusses how the operation of BGP makes router grafting challenging. In Section 3 we present the router grafting architecture, focusing only on the control plane. Section 4 explains how we ensure correct routing and forwarding, even in the face of unplanned routing changes. In Section 5 we present our prototype, followed by a discussion of optimizations that reduce the overhead of grafting a BGP session in Section 6. We present an evaluation of our prototype and proposed optimizations in Section 7, followed by related work in Section 8 and the conclusion in Section 9.

2 BGP Routing Within a Single AS

Grafting a BGP session is difficult because BGP routing relies on many *layers* in the protocol stack and many *components* within an AS. In this section, we present a brief overview of BGP routing from the perspective of a single autonomous system (AS) to identify the challenges our grafting solution must address.

2.1 Protocol Layers: IP, TCP, & BGP

As illustrated in Figure 1, two neighboring routers exchange BGP update messages over a BGP session that runs on top of a TCP connection that, in turn, directs packets over the underlying IP link(s) between them. As such, grafting a BGP session will require moving the IP link, TCP connection, and BGP session from one location to another.

IP link: An AS connects to neighboring ASes through IP links. While a link could be a direct cable between two routers, these IP-layer links typically correspond to multiple hops in an underlying layer-two network. For example, routers at an exchange point often connect via a shared switch, and an ISP typically connects to its customers over an access network. These layer-two networks are increasingly programmable, allowing dynamic set-up and tear-down of layer-three links [4, 5, 6, 7]. This is illustrated in Figure 1 where the link between routers A and B is through a programmable transport network which can be changed to connect routers A and C. These innovations enable seamless migration of an IP link from one location to another within the scope of the layer-two network, such as rehoming a customer's access link to terminate on a different router in the ISP's network[1].

TCP connection: The neighboring routers exchange BGP messages over an underlying TCP connection. Unlike a conventional TCP connection between a Web client and a Web server, the connection must stay "up" for long periods of time, as the two routers are continuously exchanging messages. Further, each router sends keep-alive messages to enable the other router to detect lapses in connectivity. Upon missing three keep-alive messages, a router declares the other router as dead and discards all BGP routes learned from that neighbor. As such, grafting a BGP session requires timely migration of the underlying TCP connection.

BGP session: Two adjacent routers form a BGP session by first establishing a TCP session, then sending messages negotiating the properties of the BGP session, then exchanging the "best route" for each destination prefix. This process is controlled by a state machine that specifies what messages to exchange and how to handle them. Once the BGP session is established, the two

[1]Depending on the technology used to realize the layer-two network, the scope might be geographically contained, e.g., in the case of a packet access network, or might be significantly more spread out, e.g., in the case of a national footprint programmable optical transport network.

routers send incremental update messages—announcing new routes and withdrawing routes that are no longer available. A router stores the BGP routes learned from its neighbor in an *Adj-RIB-in* table, and the routes announced to the neighbor in an *Adj-RIB-out* table. Each BGP session has configuration state that controls how a router filters and modifies BGP routes that it imports from (or exports to) the remote neighbor. As such, grafting a BGP session requires transferring a large amount of RIB (Routing Information Base) state, as well as moving the associated configuration state.

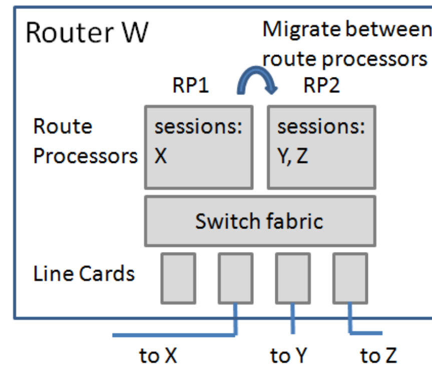


Figure 2: Migrating the session with X between route processor blades (from RP1 to RP2).

2.2 Components: Blades, Routers, & ASes

A BGP session is associated with a routing process that runs on a processor blade within one of the routers in a larger AS. As such, grafting a BGP session involves extracting the necessary state from the routing process, transferring that state to another location, and changing the routing decisions at other routers as needed.

Processor blade: The simplest router has a processor for running the routing process, multiple interfaces for terminating links, and a switching fabric for directing packets from one interface to another. The BGP routing process maintains sessions with multiple neighbors and runs a decision process over the *Adj-RIB-in* tables to select a single "best" route for each destination prefix. The routing process stores the best route in a *Loc-RIB* table, and applies export policies to construct the *Adj-RIB-out* tables and send the corresponding update messages to each neighbor.

IP router: Today's high-end routers are large distributed systems, consisting of hundreds of interfaces and multiple processor blades spread over one or more chassis. These routers run multiple BGP processes—one on each processor blade—each responsible for a portion of the BGP sessions as shown in Figure 2. For a cluster-based router to scale, each BGP process runs its own decision process and exchanges its "best" route with the other BGP processes in the router, using a modified version of internal BGP (iBGP) [8]. This allows the distributed router to behave the same way as a simple router that runs a single BGP process. Any BGP process can handle any BGP session, since all processors can reach the interface cards through the switching fabric. As such, grafting a BGP session from one blade to another in the same router (e.g., the session with X from RP1 to RP2 in Figure 2) does not require migrating the underlying layer-three link.

Autonomous System (AS): An AS consists of multiple, geographically-distributed routers. Each router forms BGP sessions with neighboring routers in other ASes, and uses iBGP to disseminate its "best" route to other routers within the AS. The routers in the same AS also run an Interior Gateway Protocol (IGP), such as OSPF or IS-IS to compute paths to reach each other. Each router in the AS runs its own BGP process(es) and selects its own best route for each prefix. The routers may come to different decisions about the best route, not only because they learn different candidate routes but also because the decision depends on the IGP distances to other routers (in a practice known as hot-potato routing). This can be seen in Figure 3 where routers B and C have different paths to the destination d. As such, grafting a BGP session from one router to another (e.g., the session with A from router B to C in Figure 3) may change the BGP routing decisions.

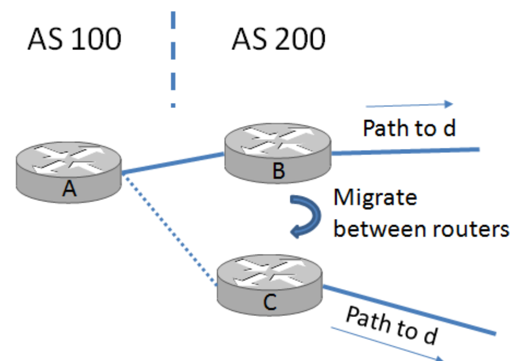


Figure 3: Migrating session with A between routers (from B to C).

3 Router Grafting Architecture

Seamless grafting of a BGP session relies on a careful progression through a number of coordinated steps. These steps are summarized in Figure 4, which shows a *migrate-from* router that hands off one of its BGP sessions to a *migrate-to* router in the same AS. These routers do not need to run the same software or be from the same vendor—they need only have the added support

for router grafting. When the grafting process starts, the migrate-from router is responsible for handling a BGP session with the remote end-point router A (not shown). This BGP session with router A is to be migrated. The migrate-from router begins exporting the routing information and the migrate-to router is initialized with its own session-level data structures and a copy of the policy configuration, without actually establishing the session (Figure 4(a)). Then, the TCP connection is migrated, followed by the underlying link (Figure 4(b)). Finally, the migrate-to router imports the routing state and updates the other routers (Figure 4(c)), resulting in the migrate-to router handling the BGP session with the remote end-point ((Figure 4(d)). This section focuses exclusively on control-plane operations, deferring discussion of the data plane until Section 4.

3.1 Copying BGP Session Configuration

Each BGP session end-point has a variety of configuration state needed to establish the session with the remote end-point (with a given IP address and AS number) and apply policies for filtering and modifying route announcements. The network operators, or an automated management system, configure the session end-point by applying configuration commands at the router's command-line interface or uploading a new configuration file. The router stores the configuration information in various internal data structures.

Rather than exporting these internal data structures, we capitalize on the fact that the current configuration is captured in a well-defined format in the configuration file. Our design simply "dumps" the configuration file for the migrate-from router, extracts the commands relevant to the BGP session end-point, and applies these commands to the migrate-to router, after appropriate translation to account for vendor-dependent differences in the command syntax. This allows the migrate-to router to create its own internal data structures for the configuration information.

However, the migrate-to router is not yet ready to assume responsibility for the BGP session. To finish initializing the migrate-to router, we extend the BGP state machine to include an 'inactive' state, where the router can create data structures and import state for the session without attempting to communicate with the remote end-point. The migrate-to router transitions from the 'inactive' state to 'established' state when instructed by the grafting process.

3.2 Exporting & Resetting Run-Time State

A router maintains a variety of state for BGP session end-points. To meet our goals, BGP grafting need only consider the Routing Information Bases (RIBs)—the other state may be simply reinitialized at the migrate-to router[2].

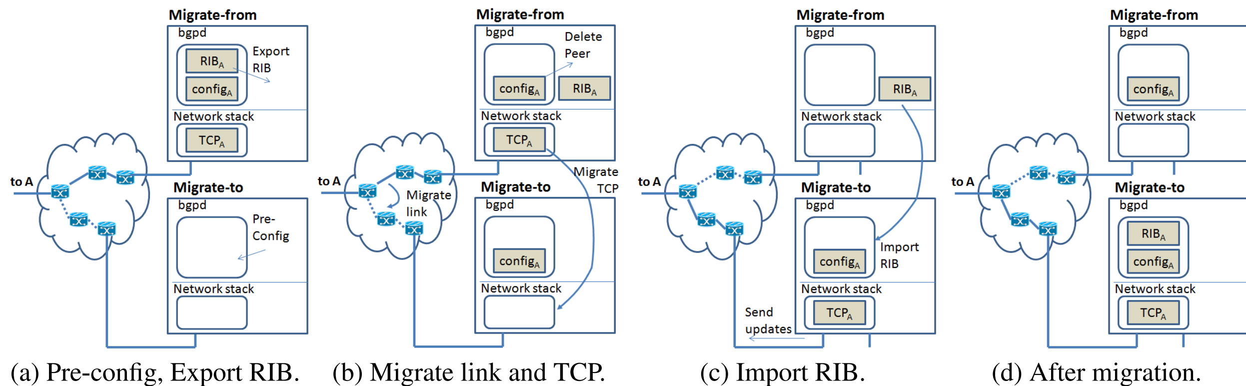
Routing Information Bases (RIBs): The most important state associated with the BGP session-end-point is stored in the routing information bases—the *Adj-RIB-in* and *Adj-RIB-out*. In our architecture, we dump the RIBs at the migrate-from router to prepare for importing the information at the migrate-to router. While the RIBs are represented differently on different router platforms, the information they store is standardized as part of the BGP protocol. In most router implementations, the RIB data structure is factored apart from the rest of the routing software, and many routers support commands for "dumping" the current RIBs. Even though the RIB dump formats vary by vendor, de facto standards like the popular MRT format [9] do exist.

State in the BGP state machine: A BGP session end-point stores information about the BGP state machine. We can forgo migrating this state – the BGP session is either 'established' or not. If the session is in one of the not-established states, we can simply close the session at the migrate-from router and start the migrate-to router in the idle state. This does not trigger any transient disruption—since the session is not "up" anyway. If the session at the migrate-from router is 'established,' we can start the new session at the migrate-to router in the 'inactive' state.

BGP timers: BGP implementations also include a variety of timers, many of which are vendor-dependent. For example, some routers use an MRAI (Minimum Route Advertisement Interval) timer to pace the transmission of BGP update messages. This is purely a local operation at one end-point of the session, not requiring any agreement with the remote end-point. Another common timer is the keep-alive interval that drives the periodic sending of heartbeat messages, and a hold timer for detecting missing keep-alive messages from the remote end-point. Fortunately, missing a single keep-alive message, or sending the message slightly early or late, would not erroneously detect a session failure because routers typically wait for *three* missed keep-alive messages before tearing down the session. As such, we do not migrate BGP timer values and instead simply initialize whatever timers are used at the migrate-to router.

BGP statistics: BGP implementations maintain numerous statistics about each session and even individual routes. These statistics, while broadly useful for network monitoring, are not essential to the correct operation of the router. They only have meaning at the local session

[2]Router grafting does not preclude the remaining state from being included, simply we chose not to in order to keep code modifications at a minimum while still meeting our goals of (i) routing protocol adjacencies staying up and (ii) all routing protocol messages being received.



(a) Pre-config, Export RIB. (b) Migrate link and TCP. (c) Import RIB. (d) After migration.

Figure 4: Router grafting mechanisms – migrating a session with Router A (not shown) from router Migrate-from to router Migrate-to. The boxes marked bgpd and network stack are the software programs. The boxes marked RIB_A, $config_A$, and TCP_A are the routing, configuration, and TCP state respectively.

end-point. In addition, these statistics are vendor dependent and not well modularized in the router software implementations. As such, we do not migrate these statistics and instead allow the migrate-to router to initialize its own statistics as if it were establishing a new session.

3.3 Migrating TCP Connection & IP Link

As part of BGP session grafting, the TCP connection must move from the migrate-from router to the migrate-to router. Because we do not assume any support from the remote end-point, the migrate-to router must use the same IP addresses and sequence and acknowledgment numbers that the migrate-from router was using. In BGP, IP addresses are used to uniquely identify the BGP session end-points and not the router as a whole. Further, we assume the link between the remote end-point and the migrate-from (or migrate-to) router is a single hop IP network where the IP address is not used for reachability, but only for identification. As such, the session end-point can easily retain its address (and sequence and acknowledgment numbers) when it moves. That is, the single IP address identifying the migrating session can be disassociated from the migrate-from router and associated with the migrate-to router. Our architecture simply migrates the local state associated with the TCP connection from one router to another.

As with any TCP migration technique, the network must endure a brief period of time when neither router is responsible for the TCP connection. TCP has its own retransmission mechanism that ensures that the remote end-point retransmits any unacknowledged data. As long as the transient outage is short, the TCP connection (and, hence, the BGP session) remains up. TCP implementations tolerate a period of at least 100 seconds [10] without receiving an acknowledgment—significantly longer than the migration times we anticipate. The amount of TCP state is relatively small, and the two routers are close to one another, leading to extremely fast TCP migration times.

The underlying link should be migrated (e.g., by changing the path in the underlying programmable transport network) close to the same time as the TCP connection state, to minimize the transient disruption in connectivity. Still, the network may need to tolerate a brief period of inconsistency where (say) the TCP connection state has moved to the migrate-to router while the traffic still flows via the migrate-from router. During this period, we need to prevent the migrate-from router from erroneously responding to TCP packets with a TCP RST packet that resets the connection. This is easily prevented by configuring the migrate-from router's interface to drop TCP packets sent to the BGP port (i.e., 179). The migrate-from route *can* successfully deliver regular *data* traffic received during the transmission, as discussed later in Section 4.

3.4 Importing BGP Routing State

Once link and connection migration are complete, the migrate-to router can move its end-point of the BGP session from the 'inactive' state to the 'established' state. At this time, the migrate-to router can begin "importing" the RIBs received from the migrate-from router. However, the import process is not as simple as merely loading the RIB entries into its own internal data structures. The migrate-from and migrate-to routers could easily have a different view of the "best" route for each destination prefix, as illustrated in Figure 5. In this scenario, before the migration, A reaches E's prefixes over the direct link between them, and B reaches E's prefixes via A; after the migration, A should reach E's prefixes via B, and B should reach E's prefixes over the direct link. Similarly, suppose routers C and D connect to a common

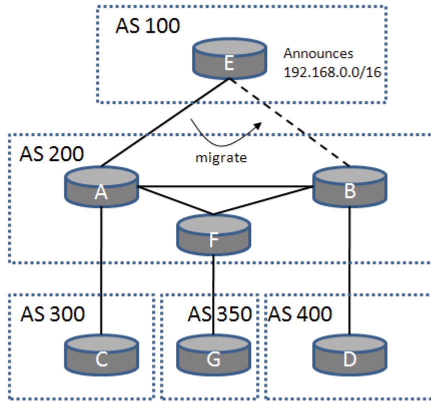


Figure 5: A topology where AS 200 has migrate-from router A, migrate-to router B, internal router F, and external routers C, D, and G, and remote end-point E.

prefix. Before the migration, E follows the AS path "100 200 300" (through C) to reach that prefix; after the migration E follows the AS path "100 200 400" (through D). Reaching these conclusions requires routers A and B to rerun the BGP decision process based on the new routes, and disseminate any routing changes to neighboring routers.

To make the process transparent to the remote end-point, we essentially emulate starting up a new session at router B, with router A temporarily playing the role of the remote end-point to announce the routes learned from E. This requires router A to replay the Adj-RIB-in state associated with E to router B. Router B stores these routes and reruns its BGP decision process, as necessary, to compute the new best routes to prefixes E is announcing. This will cause update messages to be sent to other routers within the AS and, sometimes, to external routers (like C and D). If the attributes of the route (e.g., the AS-PATH) do not change, as is the case in Figure 5, other ASes like AS 300 and AS 400 do not receive *any* BGP update message (since, from their point of view, the route has not changed), thus minimizing the overhead that router grafting imposes on the global BGP routing system.

Next, we update E with the best routes selected by B. Here, we take advantage of the fact that E has already learned routes from the migrate-from router A. The change in topology might change some of those routes, and we need to account for that. To do so, the migrate-to router runs the BGP decision process to compare its currently-selected best route to the route learned from the migrate-from router. If the best route changes, B sends an update message to its neighbors, including router E. This is in fact exactly the same operation the router would perform upon receiving a route update from any of its neighbors. We expect that routers A and B would typically have the same best route for most prefixes, especially if A and B are relatively close to each other in the IGP topology. As such, most of the time router B would not change its best route and hence would not need to send an update message to router E.

4 Correct Routing and Forwarding

Router grafting cannot be allowed to compromise the correct functioning of the network. In this section, we discuss how grafting preserves correct routing state (in the control plane) and correct packet forwarding (in the data plane), even when unexpected routing changes occur in the middle of the grafting process.

4.1 Control Plane: BGP Routing State

Routing changes can, and do, happen at any time. BGP routers easily receive millions of update messages a day, and these could arrive at any time during the grafting process – while the migrate-from router dumps its routing state, while the TCP connection and underlying link are migrated, or while the migrate-to router imports the routing state and updates its routing decisions. Our grafting solution can correctly handle BGP messages sent at any of these times.

While the migrate-from router dumps the BGP routing state: The goal is to have the in-memory Routing Information Base (RIB) be consistent with the RIB that was dumped as part of migration. Here, we take advantage of the fact that the dumping process and the BGP protocol work on a per-prefix basis. Consider a Adj-RIB-in with three routes (p1, p2, p3) corresponding to three prefixes, of which (p1 and p2) have been dumped already. When an update p3' (for the same prefix as p3) is received, the in-memory RIB can be updated since it corresponds to a prefix that has not been dumped, – to prevent dumping a prefix while it is being updated, the single entry in the RIB needs to be locked. If we receive an update p1' (for the same prefix as p1), processing it and updating the in-memory RIB without updating the dumped image will cause the two to be inconsistent – delaying processing the update is an option, but that would delay convergence as well. To solve this, we capitalize on BGP being an incremental protocol where any new update message implicitly withdraws the old one. Since we treat the dumped RIB as a sequence of update messages, we can process the update immediately and append p1' to the end of the dumped RIB to keep it consistent.

While the TCP connection and link are migrating: BGP update messages may be sent while the TCP connection and the underlying link are migrating. If a message is sent by the remote end-point, the message is not

delivered and is correctly retransmitted after the link and TCP connection come up at the migrate-to router. If an update message is sent by another router to the migrate-from router over a different BGP session, there is not a problem because the migrate-from router is no longer responsible for the recently-rehomed BGP session. Therefore, the migrate-from router can safely continue to receive, select, and send routes. If an update message is sent by another router to the migrate-to router over a different BGP session, the migrate-to router can install the route in its Adj-RIB-in for that session and, if needed, update its selection of the best route – similar to when a route is received before the migration process.

While the migrate-to router imports the routing state: The final case to consider is when the migrate-to router receives a BGP update message while importing the routing state for the rehomed session. Whether from the remote end-point or another router, if the route is for a prefix that was already imported, there is no problem since the migration of that prefix is complete. If it is for a prefix that has not already been imported, only messages from the remote end-point router need special care. (BGP is an asynchronous protocol that does not depend on the relative order of processing for messages learned from different neighbors.) A message from the remote end-point must be processed after the imported route but we would like to process it immediately. Since the update implicitly withdraws the previous announcement (which is in the dump image), we mark the RIB entry to indicate that it is more recent than the dump image. This way, we can skip importing any entries in the dump image which have a more recent RIB update.

4.2 Data Plane: Packet Forwarding

Thus far, this paper has focused on the operation of the BGP control plane. However, the control plane's only real purpose is to select paths for forwarding data packets. Fortunately, grafting has relatively little data-plane impact. When moving a BGP session between blades in the same router, the underlying link does not move and the "best" routes do not change. As such, the forwarding table does not change, and data packets travel as they did before grafting took place – the data traffic continues to flow uninterrupted.

The situation is more challenging when grafting a BGP session from one router to another, where these two routers do not have the same BGP routing information and do not necessarily make the same decisions. Because the TCP connection and link are migrated *before* the migrate-to router imports the routing state, the remote end-point briefly forwards packets through the migrate-*to* router based on BGP routes learned from the migrate-*from* router. Since BGP route dissemination within the AS (typically implemented using iBGP) ensures that each router learns at least one route for each destination prefix, the two routers will learn routes for the same set of destinations. Therefore, the undesirable situation where the remote end-point forwards packets that the migrate-to router cannot handle will not occur.

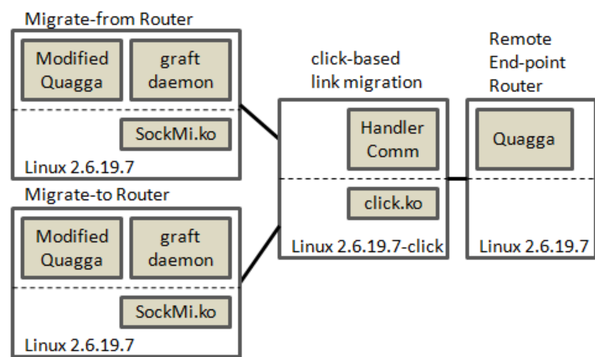


Figure 6: The router grafting prototype system.

Although data packets are forwarded correctly, the end-to-end forwarding path may temporarily differ from the control-plane messages. For example, in Figure 5, data packets sent by E will start traversing the path through AS 400, while E's control plane still thinks the AS path goes through AS 300. These kinds of temporary inconsistencies are a normal occurrence during the BGP route-convergence process, and do not disrupt the flow of traffic. Once the migrate-to router finishes importing the routes, the remote end-point will learn the new best route and control- and data-plane paths will agree again.

Correct handling of data traffic must also consider the packets routed *toward* the remote end-point. During the grafting process, routers throughout the AS forward these packets to the migrate-from router until they learn about the routing change (i.e., the new egress point for reaching these destinations). Since the migrate-from router knows where the link, TCP connection, and BGP session have moved, it can direct packets in flight there through temporary tunnels established between the migrate-from router and the migrate-to router.

5 BGP Grafting Prototype

We have developed an initial prototype to demonstrate router grafting. Figure 6 depicts the main components of the prototype. These include (i) a modified Quagga [11] routing software, (ii) the graft daemon for controlling the entire process, (iii) the SockMi [12] kernel module for TCP migration, and (iv) a Click [13] based data plane for implementing link migration.

The controlling entity in the prototype is the graft daemon. This is the entity that initiates the BGP session

grafting, interacting with each of the other components to perform the necessary steps. We assume each graft daemon can be reached by an IP address. With this, the graft daemon on the migrate-from router will initiate a TCP connection with the daemon on the migrate-to router. Once established, the migration process follows the six general steps discussed in the following subsections.

5.1 Configuring the Migrate-To Router

In our architecture, configuration state is gleaned from a dump of the migrate-from router's configuration file, rather than its internal data structures. The graft daemon first extracts BGP session configuration from the configuration file of the migrate-from router, including the rules for filtering and modifying route announcements. Then the extracted configuration commands are applied to the migrate-to router. Our current implementation includes a simplistic parser for Quagga's commands for configuring BGP sessions[3]. In order to configure the migrate-to router before migrating the TCP connection, we added an 'inactive' state to the BGP state machine. We also added a configuration command to the Quagga command-line interface:

```
neighbor w.x.y.z inactive
```

that triggers the router to create all internal data structures for the session, without attempting to open or accept a socket with the remote end-point.

5.2 Exporting Migrate-From BGP State

Once the migrate-to router is configured, the grafting process can proceed to the second step, which is initiating the export of the routing state on the migrate-from router. The grafting daemon on the migrate-from router initiates the export process by calling a command in Quagga that we added:

```
neighbor w.x.y.z migrate out
```

When this command is executed, our modified Quagga software traverses the internal data structures, dumping the necessary routing state (Adj-RIB-in and the selected routes in the loc-RIB) to a file.

[3] As we add support for XORP, we will develop a more complete parser as the configuration will require translating between configuration languages—generally a hard problem, though easier in our case because we focus on a relatively narrow aspect of the configuration.

5.3 Exporting Migrate-From TCP State

Once the routing state is dumped, the modified Quagga calls the `export_socket` function as part of the SockMi API to migrate the TCP state. This function makes an `ioctl` call to the kernel module, passing the socket's file descriptor. The SockMi kernel module is a Linux kernel module for kernels 2.4 through 2.6—we tested with kernel version 2.6.19.7. The `ioctl` call causes the kernel module to interact with Linux's internal data structures. It removes the TCP connection from the kernel, writing the socket state to a character device. Note that part of this state is related to the protocol itself (e.g., the current sequence number) as well as the buffers (e.g., the receive queue and the transmit queue of packets sent, but not acknowledged). When this state is written, the kernel module sends a signal to the graft daemon on the migrate-from router, which can read from the character device and send to the daemon on the migrate-to router.

5.4 Importing the TCP State

The next step is to initiate the import of the TCP state at the migrate-to router. Upon receiving the state from the migrate-from router, the graft daemon on the migrate-to router first notifies Quagga that it is about to import state for a given 'inactive' session. This is done through a command we added:

```
neighbor w.x.y.z migrate in
```

Upon executing the command, our modified Quagga invokes the `import_socket` function in the SockMi API. This function blocks until a TCP connection is imported. During this time, the graft daemon makes an `ioctl` to the SockMi kernel module. The graft daemon then passes the TCP session state to a character device which is read by the kernel module. The SockMi kernel module accesses the Linux data structures to add a socket with that TCP connection state, which unblocks the `import_socket` function.

5.5 Migrating the Layer-Three Link

At this point, the graft daemon of the migrate-to router triggers the migration of the underlying link. This includes removing the migrating session's IP address from the migrate-from router, adding the IP address to the migrate-to router, and migrating the layer-two link. As we did not have access to equipment to use a programmable transport network, we instead built our own simple layer-two network that connects both the migrate-from and migrate-to router to the remote end-point with a Click [13] configuration that emulates a 'programmable

transport'. This Click configuration performs a simple switching primitive that connects the remote end-point to either the migrate-from or the migrate-to router. In one setting, packets from the migrate-from router are sent to the remote end-point router, packets from the migrate-to router are dropped, and packets from the remote end-point router are sent to the migrate-from router. With the alternative setting, the reverse occurs, forming a link between the migrate-to router and the remote-end point router. This switch value is settable via a handler, making it accessible to the graft daemon running on the migrate-from router.

5.6 Importing Routing State

As the final step, when the importing of the TCP connection is complete and the `import_socket` function is unblocked, the modified Quagga reads the routing state, which was stored in a file when the local graft daemon read it in from the graft daemon running on the migrate-from router. Much as the "normal" operation of the router, which receives a BGP message from a socket and then calls a function to handle the update, the importing process will read the Adj-RIB-in from a file and call the same function to process the routing update. For comparing the RIB from the migrate-from router to the migrate-to router, the importing process reads the route from the file, looks up the route in the local RIB, and compares them. If they differ, it will use existing functions to send out the route to the peer.

6 Optimizations for Reducing Impact

Grafting a BGP session requires incrementally updating the remote end-point as well as the other routers in the AS. In this section, we present optimizations that can further reduce the traffic and processing load imposed on routers not directly involved in the grafting process. These optimizations capitalize on the knowledge that grafting is taking place and the routers' local copy of the routes previously learned from the remote end-point. First, we discuss how we can keep routers from sending unnecessary updates to their eBGP neighbors. Second, we then discuss how the majority of iBGP messages can be eliminated. Finally, we consider the intra-cluster router case where the routes do not change.

6.1 Reducing Impact on eBGP Sessions

Importing routes on the migrate-to router, and withdrawing routes on the migrate-from router, may trigger a flurry of update messages to other BGP neighbors. Consider the example in Figure 5, where before grafting router E had announced 192.168.0.0/16 to router A, which in turn announced the route to B and C. Eventually two things will happen: (i) the migrate-from router A will *remove* the 192.168.0.0/16 route from E and (ii) the migrate-to router B will *add* the 192.168.0.0/16 route from E. Without any special coordination, these two events could happen in either order.

If A removes the route before B imports it, then A's eBGP neighbors (like router C) may receive a withdrawal message, or briefly learn a different best route (should A have other candidate routes), only to have A reannounce the route upon (re)learning it from B. Alternatively, if B adds the route before A sends the withdrawal message to C, then A may have both a withdrawal message and the subsequent (re)announcement queued to send to router C, perhaps leading to redundant BGP messages. In the first case, C may temporarily have no route at all, and in the second case C may receive redundant messages. In both cases these effects are temporary, but we would like to avoid them if possible.

To do so, rather than deleting the route, A can mark the route as "exported"—safe in the knowledge that, if this route should remain the best route, A will soon (re)learn it from the migrate-to router B. For example, suppose the route from E is the *only* route for the destination prefix—then A would certainly (re)learn the route from B, and could forgo withdrawing and reannouncing the route to its other neighbors. Of course, if A does not receive the announcement (either after some period of time or implicitly through receiving an update with a different route for that prefix), then it can proceed with deleting the exported route.

So far we only considered the eBGP messages the migrate-from router would send. A similar situation can occur on the eBGP sessions of the other routers in the AS (e.g., router F). This is because these other routers must be notified (via iBGP) to no longer go through A for the routes learned over the migrating session (i.e., with E). Therefore, the migrate-from router must send out withdrawal messages to its iBGP neighbors and the migrate-to router must send out announcements to its iBGP neighbors. This may result in the other routers in the AS (e.g., router F) temporarily withdrawing a route, temporarily sending a different best route, or sending a redundant update to their eBGP neighbors. Because of this, we have the migrate-from router send the marked list to each of its iBGP neighbors and a notification that these all migrated to the migrate-to router – this list is simply the list of prefixes, not the associated attributes. We expect this list to be relatively small in terms of total bytes. With this list, the other routers in the AS can perform the same procedure, and eliminate any unnecessary external messages.

6.2 Reducing Impact on iBGP Sessions

While using iBGP unmodified is sufficient for dealing with the change in topology brought about by migration, it is still desirable to reduce the impact migration has on the iBGP sessions. Here, since the route-selection policy will likely be consistent throughout an ISP's network, we can reduce the number of update messages sent by extending iBGP (an easier task than modifying eBGP). When the migrate-from and migrate-to routers select the same routes, the act of migration will not change the decision. Since all routers are informed of the migration, the iBGP updates can be suppressed (the migrate-from router withdrawing the route and the migrate-to router announcing the route). When the migrate-from and migrate-to routers select different routes, it is most likely due to differences in IGP distances. For the migrate-to router, the act of migration will cause all routes learned from the remote end-point router to become directly learned routes, as opposed to some distance away, and therefore the migrate-to router will now prefer those routes (except when the migrate-to router's currently selected route is also directly learned). This change in route selection causes the migrate-to router to send updates to its iBGP neighbors notifying them of the change. However, since it is more common to change routes, we can reduce the number of updates that need to be sent with a modification to iBGP where updates are sent when the migrate-to router keeps a route instead of when it changes a route. Other routers will be notified of the migration and will assume the routes being migrated will be selected unless told otherwise.

6.3 Eliminating Processing Entirely

Re-running the route-selection processes is essential as migration can change the topology, and therefore change the best route. When migrating within a cluster router, the topology does not change, and therefore we should be able to eliminate processing entirely. The selected best route will be a consistent selection on every blade. Therefore, even when migrating, while the internal data structures might need to be adjusted, no decision process needs to be run and no external messages need to be sent. In fact, there is no need for any internal messages to be sent either. With the modified iBGP used for communication between route processor blades, the next hop field is the next router, not the next processor blade – i.e., iBGP messages are only used to exchange routes learned externally and do not affect how packets are forwarded internally. Therefore, upon migration, there is no need to send an update as the routes learned externally have already been exchanged.

While exchanging messages and running the decision process can be eliminated, transferring the routing state from the exporting blade to the importing blade is still needed. Being the blade responsible for a particular BGP session requires that the local RIB have all of the routes learned over that session. While some may have been previously announced by the migrate-from blade, not all of them were. Therefore, we need to send over the Adj-RIB-in for the migrating session in order to know all routes learned over that session as well as which subset of routes the migrate-from blade announced were associated with that session.

7 Performance Evaluation

In this section, we evaluate router grafting through experiments with our prototype system and realistic traces of BGP update messages. We focus primarily on control-plane overhead, since data-plane performance depends primarily on the latency for link migration—where our solution simply leverages recent innovations in programmable transport networks. First, we evaluate our prototype implementation from Section 5 to measure the grafting time and CPU utilization on the migrate-from and migrate-to routers. Then we evaluate the effectiveness of our optimizations from Section 6 in reducing the number of update messages received by other routers.

7.1 Grafting Delay and Overhead

The first experiment measures the impact of BGP session grafting on the migrate-from and migrate-to routers. To do this we supplemented the topology shown in Figure 5 with a router adjacent to E (in a different AS) and a router adjacent to B (in a different AS). These two extra routers were fed a BGP update message trace taken from RouteViews [14]. This essentially fills the RIB of B and E with routes that have the same set of prefixes, but different paths. We used Emulab [15] to run the experiment on servers with 3GHz processors and 2GB RAM.[4]

The time it takes to complete the migration process is a function of the size of the routing table. The larger it is, the larger the state that needs to be transferred and the more routes that need to be compared. To capture this relationship, we varied the RIB size by replaying multiple traces. The results, shown in Figure 7, include both the case where migration occurs between routers (when the migrate-to router must run the BGP decision process) and the case where migration is between blades (where the decision process does not need to run because the underlying topology is not changed). The "between blades" curve, then, illustrates the time required to transfer the BGP routes and import them into the internal data

[4]This is roughly comparable to the route processors used in commercially available high-end routers.

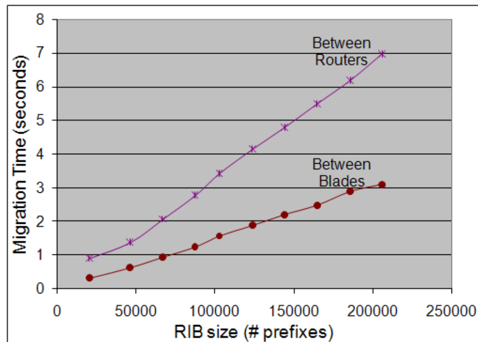


Figure 7: BGP session grafting time vs. RIB size.

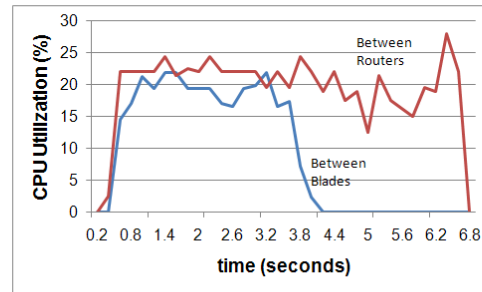


Figure 8: The CPU utilization at the migrate-to router during migration, with a 200k prefix RIB.

structures. Note that these results do not imply that TCP needs to be able to handle this long of an outage where packets go unacknowledged – the TCP migration process takes less than a millisecond. Instead, when compared to rehoming a customer today, where there is downtime measured in minutes, the migration time is small. In fact, since in our setup AS100 and AS200 have a peering agreement, the actual migration time would be less if AS100 were a customer of AS200 (since AS100 would announce fewer routes to AS200).

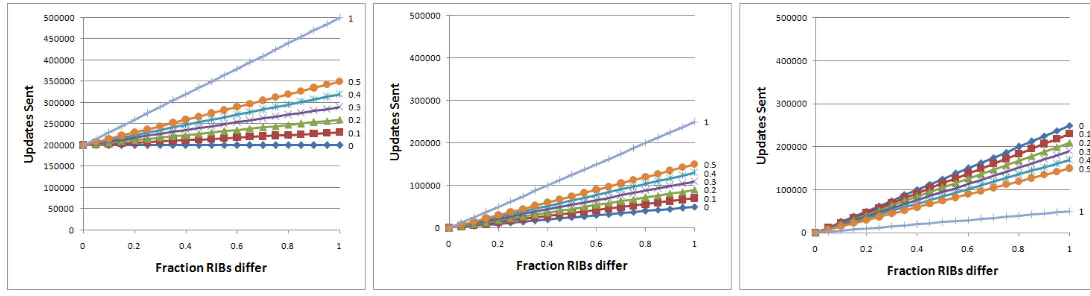
The CPU utilization during the grafting process is also important. The BGP process on the migrate-from router experienced only a negligible increase in CPU utilization for dumping the BGP RIBs. The migrate-to router needs to import the routing entries and compare routing tables. For each prefix in the received routing information, the migrate-to router must perform a lookup to find the routing table entry for that prefix. Figure 8 shows the CPU utilization at 0.2 second intervals, as reported by *top*, for the case where the RIB consists of 200,000 prefixes. There are three things to note. First, the CPU utilization is roughly constant. This is perhaps due to the implementation where the data is received, placed in a file, then iteratively read from the file and processed before reading the next. This keeps the CPU utilization at only a fraction as computation is mixed with reads from disk. Second, the CPU utilization is the same for both migrating between routers and migrating between blades. The case between routers merely takes longer because of the additional work involved in running the BGP decision process. Third, migration can be run as a lower priority task and use less CPU but take longer – preventing the migration from effecting the performance of the router during spikes in routing updates, which commonly results in intense CPU usage during the spikes.

7.2 Optimizations for Reducing Impact

While the impact on the migrate-from and migrate-to routers is important, perhaps a more important metric is the impact on the routers *not* involved in the migration, including other routers within the same AS as well as the eBGP neighbors. If the overhead of grafting is relatively contained, network operators could more freely apply the technique to simplify network-management tasks.

First and foremost, the remote end-point experiences an overhead directly proportional to the number of additional BGP update messages it receives. The number of messages depends on how many best routes differ between the migrate-from and migrate-to router—the migrate-from router must send an update message for every route that differs. The exact amount depends heavily on the proximity of the migrate-from and migrate-to routers—if the two routers are in the same Point-of-Presence of the ISP, perhaps no routes would change. As such, we do not expect this overhead to be significant. Since the sources of overhead for the remote end-point are relatively well understood, and it is difficult to acquire the kinds of intra-ISP measurement data necessary to quantify the number of route changes, we do not present a plot for this case.

Perhaps the more significant impact is on the other routers, both within the AS and in other ASes, that may have to learn new routes for the prefixes announced by the remote end-point. To evaluate this, we measured the number of updates that would be sent as a function of the fraction of prefixes where the migrate-from router had selected a different route than the migrate-to router. By doing so, this covers the entire range of migration targets (i.e. it does not limit our evaluation to migration within a PoP). Recall that this difference is what needs to be corrected. Also recall that the prefixes being considered here are the ones learned from the router at the remote end-point of the session being migrated, not the entire routing table, as these are the routes that could impact what is sent to other routers. For our measurement, we use a fixed set of 100,000 prefixes. However, the results are directly proportional to the number of prefixes, and can therefore be scaled appropriately – for migrating a customer link, the number of prefixes would be significantly smaller, for migrating a peering link, the number



(a) Without optimization. (b) Reducing eBGP impact. (c) Reducing iBGP impact.

Figure 9: Updates sent as a result of migration.

of prefixes could be higher.

The results are shown in Figure 9, with the three graphs representing the three different cases as discussed in Section 6: (a) direct approach with no optimizations, (b) optimizations to reduce eBGP messages by capitalizing on redundant information in the network, and (c) optimizations to reduce iBGP messages by treating the route selection changing as the common case. For the graphs, each line represents a fixed fraction of differing routes that change the selected route as a result of the grafting. For example, consider where the migrate-from router selects a particular route different than the migrate-to router. In this case, after migration, the migrate-to router selects the route the migrate-from selected (i.e., it changes its own route). Each line represents the fraction of times this change occurs—for example, the line labeled 0.2 in Figure 9 is where 20% of the routes that differ will change to the routes selected by the migrate-from router.

There are several things of note from the graphs. First is that the direct (unoptimized) approach must send significantly more messages. In the case where the selected routes do not differ much, which we consider will be a most likely scenario, the optimized approaches hardly send any messages at all. Second, when comparing Figure 9(b) with Figure 9(c), we can see that depending on what would be considered the common case, we can choose a method that would result in the fewest updates. For (b), the assumption is that when the routes differ, the migrate-to router will not change to the routes the migrate-from selected. Whereas in (c), the assumption is that when the routes differ, the migrate-from router will change to the routes the migrate-from router selected. The reason they would change is that the routes learned from the remote end-point of the session being migrated will now be directly learned routes, rather than via iBGP. It is likely that the policy of route selection is consistent throughout the ISPs network, and therefore differences will be due to IGP distances and changing the router will change those routes to be more preferable. We are working on characterizing when these differences would occur in order to enable us to predict the impact a given migration might have. Third, and perhaps most important, migration can be performed with minimal disruption to other routers in the likely scenario where there are few differences in routes selected.

8 Related Work

High availability and ease of network management are goals of many systems, and therefore router grafting has much in common with them. In particular, ones that attempt to minimize disruptions during planned maintenance. One possibility is to reconfigure the routing protocols such that traffic will no longer be sent to the router about to undergo maintenance [16, 17]. Alternatively, others have decoupled the control plane and data plane such that the router can continue to forward packets while the control plane goes off-line (e.g., rebooted) [18, 19]. However, unlike router grafting, these require modifications to the remote end-point router and they are only useful for temporarily shutting down the session on a given physical router, rather than enabling the session to come back up on a different router as in router grafting.

In this regard, router grafting shares more in common with VROOM [3], which makes use of virtual machine migration [20] to ease network management. Maintenance could be performed without taking down the router simply by migrating the virtual router to another physical router. This requires the two physical routers to be compatible (running the same virtualization technology), a limitation router grafting does not have. In fact, router grafting does not rely on virtual machine technology. Kozuch showed the ability to migrate without the use of virtualization [21], but did so at the granularity of the entire operating system and all running processes. With a coarse granularity, the physical router where the virtual router is being migrated to must be able to handle the entire virtual router's load.

Router grafting is also similar to the RouterFarm work [6], which targeted re-homing a customer. How-

ever, it required restarting the session and is more disruptive than router grafting. Along similar lines, high-availability routers enable switching over to a different router or blade in a router [22]. This, however, is done either through periodically check-pointing, which preserves the memory image, or running two complete instances of the router software concurrently, which is an inefficient use of resources.

While we presented router grafting in the context of a BGP session, we envision it being more general. Along these lines, partitioning the prefix space across multiple routers or blades is a possibility.ViAggre [23] partitions the prefix space across multiple routers, however it is a static architecture not one which dynamically repartitions the prefix space as router grafting could.

Finally, we made use of TCP socket migration to handle change or disruption in end-points. One alternative is to modify the TCP protocol to include the ability to change IP addresses [24]. Since the IP address of the end-points in router grafting can remain the same, we do not need this capability, but could make use of it.

9 Conclusions

Router grafting is a new technique that opens many new possibilities for managing a network. It does this by enabling, without disruption, the migration of a routing session between (i) physical routers, (ii) blades in a cluster router, and (iii) routers from different vendors. We were able to do this while being transparent to the remote end-point. We handled the changes in topology through incremental updates, only sending out the necessary updates to convey the difference. Importantly, we did not affect the correctness of the network as the data plane will continue to forward packets and routing updates do not cause the migration to be aborted.

Going forward, we plan to explore the motivating applications for router grafting to further demonstrate the usefulness of our new technique. We are particularly interested in exploring the role of router grafting in traffic engineering. Finally, this work raises interesting questions about what exactly a router is, and the various ways routers can be "sliced and diced." We plan to explore these questions in our ongoing work.

References

[1] S. Agarwal, C. Chuah, S. Bhattacharyya, and C. Diot, "Impact of BGP dynamics on router CPU utilization," in *Passive and Active Measurement*, April 2004.

[2] J. Gottlieb, A. Greenberg, J. Rexford, and J. Wang, "Automated provisioning of BGP customers," *IEEE Network Magazine*, November/December 2003.

[3] Y. Wang, E. Keller, B. Biskeborn, J. van der Merwe, and J. Rexford, "Virtual Routers on the Move: Live Router Migration as a Network-Management Primitive," in *ACM SIGCOMM*, August 2008.

[4] J. Wei, K. Ramakrishnan, R. Doverspike, and J. Pastor, "Convergence through packet-aware transport," *Journal of Optical Networking*, vol. 5, April 2006.

[5] "Ciena CoreDirector Switch." `http://www.ciena.com`.

[6] M. Agrawal, S. Bailey, A. Greenberg, J. Pastor, P. Sebos, S. Seshan, J. van der Merwe, and J. Yates, "RouterFarm: Towards a dynamic, manageable network edge," in *Proc. ACM SIGCOMM Workshop on Internet Network Management (INM)*, September 2006.

[7] A. Rostami and E. Sargent, "An optical integrated system for implementation of NxM optical cross-connect, beam splitter, mux/demux and combiner." IJCSNS International Journal of Computer Science and Network Security, July 2006.

[8] M. Tahir, M. Ghattas, D. Birhanu, and S. N. Nawaz, *Cisco IOS XR Fundamentals*. Cisco Press, 2009.

[9] "IETF draft: MRT routing information export format," July 2009. `http://tools.ietf.org/id/draft-ietf-grow-mrt-10.txt`.

[10] R. Braden, "Requirements for Internet Hosts - Communication Layers." RFC 1122, October 1989.

[11] "Quagga software routing suite," `www.quagga.net`.

[12] M. Bernaschi, F. Casadei, and P. Tassotti, "SockMi: a solution for migrating TCP/IP connections," in *Proc. Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, 2007.

[13] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Kaashoek, "The Click modular router," in *ACM Trans. Comp. Sys.*, August 2000.

[14] "Route views project," `http://www.routeviews.org`.

[15] B. White, J. Lepreau, L. Stoller, R. Ricci, G. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," in *OSDI*, December 2002.

[16] R. Teixeira, A. Shaikh, T. Griffin, and J. Rexford, "Dynamics of hot-potato routing in IP networks," *IEEE/ACM Trans. Networking*, December 2008.

[17] P. Francois, M. Shand, and O. Bonaventure, "Disruption-free topology reconfiguration in OSPF networks," in *Proc. IEEE INFOCOM*, May 2007.

[18] A. Shaikh, R. Dube, and A. Varma, "Avoiding instability during graceful shutdown of multiple OSPF routers," *IEEE/ACM Trans. Networking*, vol. 14, pp. 532–542, June 2006.

[19] E. Chen, R. Fernando, J. Scudder, and Y. Rekhter, "Graceful Restart Mechanism for BGP." RFC 4724, January 2007.

[20] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live Migration of Virtual Machines," in *Proc. Networked Systems Design and Implementation*, May 2005.

[21] M. A. Kozuch, M. Kaminsky, and M. P. Ryan, "Migration without virtualization," in *Proc. Workshop on Hot Topics in Operating Systems*, May 2009.

[22] "Cisco IOS high availability curbs downtime with faster reloads and upgrades." `http://www.cisco.com/en/US/products/ps6550/prod_white_papers_list.html`.

[23] H. Ballani, P. Francis, T. Cao, and J. Wang, "Making Routers Last Longer with ViAggre," in *Proc. of USENIX Symposium on Networked Systems Design and Implementation*, April 2009.

[24] A. Snoeren and H. Balakrishnan, "An end-to-end approach to host mobility," in *Proc. ACM MOBICOM*, (Boston, MA), August 2000.

ElasticTree: Saving Energy in Data Center Networks

Brandon Heller⋆, Srini Seetharaman†, Priya Mahadevan⋄,
Yiannis Yiakoumis⋆, Puneet Sharma⋄, Sujata Banerjee⋄, Nick McKeown⋆
⋆ Stanford University, Palo Alto, CA USA
† Deutsche Telekom R&D Lab, Los Altos, CA USA
⋄ Hewlett-Packard Labs, Palo Alto, CA USA

ABSTRACT

Networks are a shared resource connecting critical IT infrastructure, and the general practice is to *always* leave them on. Yet, meaningful energy savings can result from improving a network's ability to scale up and down, as traffic demands ebb and flow. We present *ElasticTree*, a network-wide power[1] manager, which dynamically adjusts the set of active network elements — links and switches — to satisfy changing data center traffic loads.

We first compare multiple strategies for finding minimum-power network subsets across a range of traffic patterns. We implement and analyze ElasticTree on a prototype testbed built with production OpenFlow switches from three network vendors. Further, we examine the trade-offs between energy efficiency, performance and robustness, with real traces from a production e-commerce website. Our results demonstrate that for data center workloads, *ElasticTree* can save up to 50% of network energy, while maintaining the ability to handle traffic surges. Our fast heuristic for computing network subsets enables ElasticTree to scale to data centers containing thousands of nodes. We finish by showing how a network admin might configure ElasticTree to satisfy their needs for performance and fault tolerance, while minimizing their network power bill.

1. INTRODUCTION

Data centers aim to provide reliable and scalable computing infrastructure for massive Internet services. To achieve these properties, they consume huge amounts of energy, and the resulting operational costs have spurred interest in improving their efficiency. Most efforts have focused on servers and cooling, which account for about 70% of a data center's total power budget. Improvements include better components (low-power CPUs [12], more efficient power supplies and water-cooling) as well as better software (tickless kernel, virtualization, and smart cooling [30]).

With energy management schemes for the largest power consumers well in place, we turn to a part of the data center that consumes 10-20% of its total power: the network [9]. The total power consumed by networking elements in data centers in 2006 in the U.S. alone was 3 billion kWh and rising [7]; our goal is to significantly reduce this rapidly growing energy cost.

1.1 Data Center Networks

As services scale beyond ten thousand servers, inflexibility and insufficient bisection bandwidth have prompted researchers to explore alternatives to the traditional 2N tree topology (shown in Figure 1(a)) [1] with designs such as VL2 [10], PortLand [24], DCell [16], and BCube [15]. The resulting networks look more like a mesh than a tree. One such example, the fat tree [1][2], seen in Figure 1(b), is built from a large number of richly connected switches, and can support any communication pattern (i.e. full bisection bandwidth). Traffic from lower layers is spread across the core, using multipath routing, valiant load balancing, or a number of other techniques.

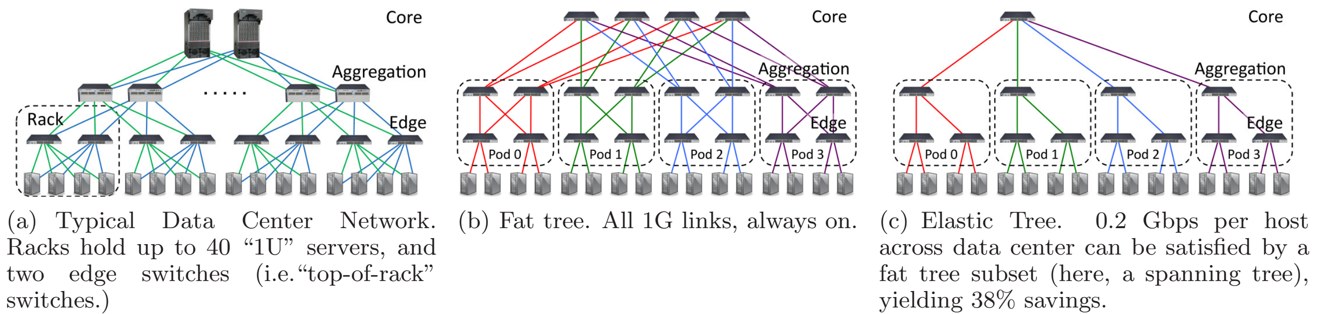
In a 2N tree, one failure can cut the effective bisection bandwidth in half, while two failures can disconnect servers. Richer, mesh-like topologies handle failures more gracefully; with more components and more paths, the effect of any individual component failure becomes manageable. This property can also help improve energy efficiency. In fact, dynamically varying the number of active (powered on) network elements provides a control knob to tune between energy efficiency, performance, and fault tolerance, which we explore in the rest of this paper.

1.2 Inside a Data Center

Data centers are typically provisioned for peak workload, and run well below capacity most of the time. Traffic varies daily (e.g., email checking during the day), weekly (e.g., enterprise database queries on weekdays), monthly (e.g., photo sharing on holidays), and yearly (e.g., more shopping in December). Rare events like cable cuts or celebrity news may hit the peak capacity, but most of the time traffic can be satisfied by a subset of the network links and

[1] We use power and energy interchangeably in this paper.

[2] Essentially a buffered Clos topology.



(a) Typical Data Center Network. Racks hold up to 40 "1U" servers, and two edge switches (i.e."top-of-rack" switches.)

(b) Fat tree. All 1G links, always on.

(c) Elastic Tree. 0.2 Gbps per host across data center can be satisfied by a fat tree subset (here, a spanning tree), yielding 38% savings.

Figure 1: Data Center Networks: (a), 2N Tree (b), Fat Tree (c), ElasticTree

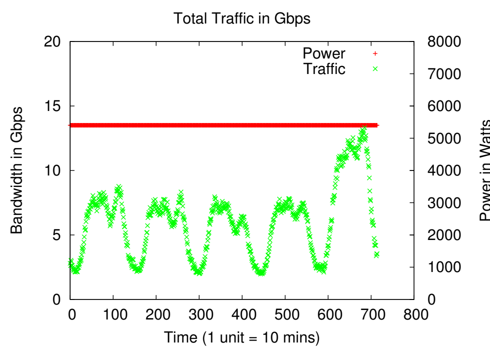
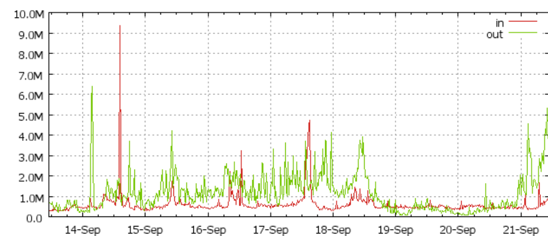
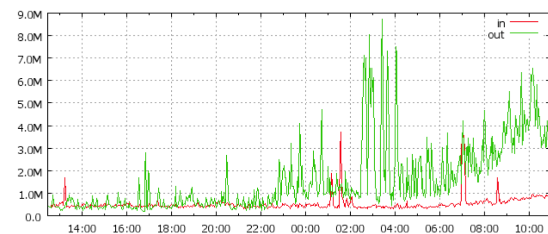


Figure 2: E-commerce website: 292 production web servers over 5 days. Traffic varies by day/weekend, power doesn't.



(a) Router port for 8 days. Input/output ratio varies.



(b) Router port from Sunday to Monday. Note marked increase and short-term spikes.

Figure 3: Google Production Data Center

switches. These observations are based on traces collected from two production data centers.

Trace 1 (Figure 2) shows aggregate traffic collected from 292 servers hosting an e-commerce application over a 5 day period in April 2008 [22]. A clear diurnal pattern emerges; traffic peaks during the day and falls at night. Even though the traffic varies significantly with time, the rack and aggregation switches associated with these servers draw constant power (secondary axis in Figure 2).

Trace 2 (Figure 3) shows input and output traffic at a router port in a production Google data center in September 2009. The Y axis is in Mbps. The 8-day trace shows diurnal and weekend/weekday variation, along with a constant amount of background traffic. The 1-day trace highlights more short-term bursts. Here, as in the previous case, the power consumed by the router is fixed, irrespective of the traffic through it.

1.3 Energy Proportionality

An earlier power measurement study [22] had presented power consumption numbers for several data center switches for a variety of traffic patterns and switch configurations. We use switch power measurements from this study and summarize relevant results in Table 1. In all cases, turning the switch on consumes most of the power; going from zero to full traffic increases power by less than 8%. Turning off a switch yields the most power benefits, while turning off an unused port saves only 1-2 Watts. Ideally, an unused switch would consume no power, and energy usage would grow with increasing traffic load. Consuming energy in proportion to the load is a highly desirable behavior [4, 22].

Unfortunately, today's network elements are not energy proportional: fixed overheads such as fans, switch chips, and transceivers waste power at low loads. The situation is improving, as competition encourages more efficient products, such as closer-to-energy-proportional links and switches [19, 18, 26, 14]. However, maximum efficiency comes from a

Ports Enabled	Port Traffic	Model A power (W)	Model B power (W)	Model C power (W)
None	None	151	133	76
All	None	184	170	97
All	1 Gbps	195	175	102

Table 1: Power consumption of various 48-port switches for different configurations

combination of improved components and improved component management.

Our choice – as presented in this paper – is to manage today's non energy-proportional network components more intelligently. By zooming out to a whole-data-center view, a network of on-or-off, non-proportional components can act as an energy-proportional ensemble, and adapt to varying traffic loads. The strategy is simple: turn off the links and switches that we don't need, right now, to keep available only as much networking capacity as required.

1.4 Our Approach

ElasticTree is a network-wide energy optimizer that continuously monitors data center traffic conditions. It chooses the set of network elements that must stay active to meet performance and fault tolerance goals; then it powers down as many unneeded links and switches as possible. We use a variety of methods to decide which subset of links and switches to use, including a formal model, greedy bin-packer, topology-aware heuristic, and prediction methods. We evaluate ElasticTree by using it to control the network of a purpose-built cluster of computers and switches designed to represent a data center. Note that our approach applies to currently-deployed network devices, as well as newer, more energy-efficient ones. It applies to single forwarding boxes in a network, as well as individual switch chips within a large chassis-based router.

While the energy savings from powering off an individual switch might seem insignificant, a large data center hosting hundreds of thousands of servers will have tens of thousands of switches deployed. The energy savings depend on the traffic patterns, the level of desired system redundancy, and the size of the data center itself. Our experiments show that, on average, savings of 25-40% of the network energy in data centers is feasible. Extrapolating to all data centers in the U.S., we estimate the savings to be about 1 billion KWhr annually (based on 3 billion kWh used by networking devices in U.S. data centers [7]). Additionally, reducing the energy consumed by networking devices also results in a proportional reduction in cooling costs.

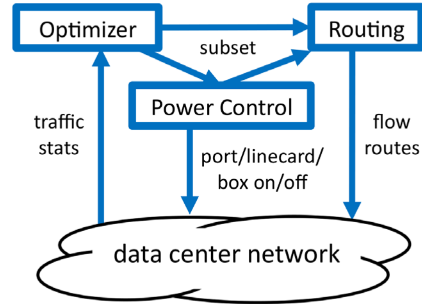


Figure 4: System Diagram

The remainder of the paper is organized as follows: §2 describes in more detail the ElasticTree approach, plus the modules used to build the prototype. §3 computes the power savings possible for different communication patterns to understand best and worse-case scenarios. We also explore power savings using real data center traffic traces. In §4, we measure the potential impact on bandwidth and latency due to ElasticTree. In §5, we explore deployment aspects of ElasticTree in a real data center. We present related work in §6 and discuss lessons learned in §7.

2. ELASTICTREE

ElasticTree is a system for dynamically adapting the energy consumption of a data center network. ElasticTree consists of three logical modules - optimizer, routing, and power control - as shown in Figure 4. The optimizer's role is to find the minimum-power network subset which satisfies current traffic conditions. Its inputs are the topology, traffic matrix, a power model for each switch, and the desired fault tolerance properties (spare switches and spare capacity). The optimizer outputs a set of active components to both the power control and routing modules. Power control toggles the power states of ports, linecards, and entire switches, while routing chooses paths for all flows, then pushes routes into the network.

We now show an example of the system in action.

2.1 Example

Figure 1(c) shows a worst-case pattern for network locality, where each host sends one data flow halfway across the data center. In this example, 0.2 Gbps of traffic per host must traverse the network core. When the optimizer sees this traffic pattern, it finds which subset of the network is sufficient to satisfy the traffic matrix. In fact, a minimum spanning tree (MST) is sufficient, and leaves 0.2 Gbps of extra capacity along each core link. The optimizer then

informs the routing module to compress traffic along the new sub-topology, and finally informs the power control module to turn off unneeded switches and links. We assume a 3:1 idle:active ratio for modeling switch power consumption; that is, 3W of power to have a switch port, and 1W extra to turn it on, based on the 48-port switch measurements shown in Table 1. In this example, 13/20 switches and 28/48 links stay active, and ElasticTree reduces network power by 38%.

As traffic conditions change, the optimizer continuously recomputes the optimal network subset. As traffic increases, more capacity is brought online, until the full network capacity is reached. As traffic decreases, switches and links are turned off. Note that when traffic is increasing, the system must wait for capacity to come online before routing through that capacity. In the other direction, when traffic is decreasing, the system must change the routing - by moving flows off of soon-to-be-down links and switches - before power control can shut anything down.

Of course, this example goes too far in the direction of power efficiency. The MST solution leaves the network prone to disconnection from a single failed link or switch, and provides little extra capacity to absorb additional traffic. Furthermore, a network operated close to its capacity will increase the chance of dropped and/or delayed packets. Later sections explore the tradeoffs between power, fault tolerance, and performance. Simple modifications can dramatically improve fault tolerance and performance at low power, especially for larger networks. We now describe each of ElasticTree modules in detail.

2.2 Optimizers

We have developed a range of methods to compute a minimum-power network subset in ElasticTree, as summarized in Table 2. The first method is a formal model, mainly used to evaluate the solution quality of other optimizers, due to heavy computational requirements. The second method is greedy bin-packing, useful for understanding power savings for larger topologies. The third method is a simple heuristic to quickly find subsets in networks with regular structure. Each method achieves different tradeoffs between scalability and optimality. All methods can be improved by considering a data center's past traffic history (details in §5.4).

2.2.1 Formal Model

We desire the optimal-power solution (subset and flow assignment) that satisfies the traffic constraints,

[3]Bounded percentage from optimal, configured to 10%.

Type	Quality	Scalability	Input	Topo
Formal	Optimal [3]	Low	Traffic Matrix	Any
Greedy	Good	Medium	Traffic Matrix	Any
Topo-aware	OK	High	Port Counters	Fat Tree

Table 2: Optimizer Comparison

but finding the optimal flow assignment alone is an NP-complete problem for integer flows. Despite this computational complexity, the formal model provides a valuable tool for understanding the solution quality of other optimizers. It is flexible enough to support arbitrary topologies, but can only scale up to networks with less than 1000 nodes.

The model starts with a standard multi-commodity flow (MCF) problem. For the precise MCF formulation, see Appendix A. The constraints include link capacity, flow conservation, and demand satisfaction. The variables are the flows along each link. The inputs include the topology, switch power model, and traffic matrix. To optimize for power, we add binary variables for every link and switch, and constrain traffic to only active (powered on) links and switches. The model also ensures that the full power cost for an Ethernet link is incurred when either side is transmitting; there is no such thing as a half-on Ethernet link.

The optimization goal is to minimize the total network power, while satisfying all constraints. Splitting a single flow across multiple links in the topology might reduce power by improving link utilization overall, but reordered packets at the destination (resulting from varying path delays) will negatively impact TCP performance. Therefore, we include constraints in our formulation to (optionally) prevent flows from getting split.

The model outputs a subset of the original topology, plus the routes taken by each flow to satisfy the traffic matrix. Our model shares similar goals to Chabarek *et al.* [6], which also looked at power-aware routing. However, our model (1) focuses on data centers, not wide-area networks, (2) chooses a subset of a fixed topology, not the component (switch) configurations in a topology, and (3) considers individual flows, rather than aggregate traffic.

We implement our formal method using both MathProg and General Algebraic Modeling System (GAMS), which are high-level languages for optimization modeling. We use both the GNU Linear Programming Kit (GLPK) and CPLEX to solve the formulation.

2.2.2 Greedy Bin-Packing

For even simple traffic patterns, the formal model's solution time scales to the 3.5^{th} power as a function of the number of hosts (details in §5). The greedy bin-packing heuristic improves on the formal model's scalability. Solutions within a bound of optimal are not guaranteed, but in practice, high-quality subsets result. For each flow, the greedy bin-packer evaluates possible paths and chooses the leftmost one with sufficient capacity. By leftmost, we mean in reference to a single layer in a structured topology, such as a fat tree. Within a layer, paths are chosen in a deterministic left-to-right order, as opposed to a random order, which would evenly spread flows. When all flows have been assigned (which is not guaranteed), the algorithm returns the active network subset (set of switches and links traversed by some flow) plus each flow path.

For some traffic matrices, the greedy approach will not find a satisfying assignment for all flows; this is an inherent problem with any greedy flow assignment strategy, even when the network is provisioned for full bisection bandwidth. In this case, the greedy search will have enumerated all possible paths, and the flow will be assigned to the path with the lowest load. Like the model, this approach requires knowledge of the traffic matrix, but the solution can be computed incrementally, possibly to support on-line usage.

2.2.3 Topology-aware Heuristic

The last method leverages the regularity of the fat tree topology to quickly find network subsets. Unlike the other methods, it does not compute the set of flow routes, and assumes perfectly divisible flows. Of course, by splitting flows, it will pack every link to full utilization and reduce TCP bandwidth — not exactly practical.

However, simple additions to this "starter subset" lead to solutions of comparable quality to other methods, but computed with less information, and in a fraction of the time. In addition, by decoupling power optimization from routing, our method can be applied alongside *any* fat tree routing algorithm, including OSPF-ECMP, valiant load balancing [10], flow classification [1] [2], and end-host path selection [23]. Computing this subset requires only port counters, not a full traffic matrix.

The intuition behind our heuristic is that to satisfy traffic demands, an edge switch doesn't care *which* aggregation switches are active, but instead, *how many* are active. The "view" of every edge switch in a given pod is identical; all see the same number of aggregation switches above. The number of required switches in the aggregation layer is then equal to the number of links required to support the traffic of the most active source above or below (whichever is higher), assuming flows are perfectly divisible. For example, if the most active source sends 2 Gbps of traffic up to the aggregation layer and each link is 1 Gbps, then two aggregation layer switches must stay on to satisfy that demand. A similar observation holds between each pod and the core, and the exact subset computation is described in more detail in §5. One can think of the topology-aware heuristic as a cron job for that network, providing periodic input to *any* fat tree routing algorithm.

For simplicity, our computations assume a homogeneous fat tree with one link between every connected pair of switches. However, this technique applies to full-bisection-bandwidth topologies with any number of layers (we show only 3 stages), bundled links (parallel links connecting two switches), or varying speeds. Extra "switches at a given layer" computations must be added for topologies with more layers. Bundled links can be considered single faster links. The same computation works for other topologies, such as the aggregated Clos used by VL2 [10], which has 10G links above the edge layer and 1G links to each host.

We have implemented all three optimizers; each outputs a network topology subset, which is then used by the control software.

2.3 Control Software

ElasticTree requires two network capabilities: traffic data (current network utilization) and control over flow paths. NetFlow [27], SNMP and sampling can provide traffic data, while policy-based routing can provide path control, to some extent. In our ElasticTree prototype, we use OpenFlow [29] to achieve the above tasks.

OpenFlow: OpenFlow is an open API added to commercial switches and routers that provides a flow table abstraction. We first use OpenFlow to validate optimizer solutions by directly pushing the computed set of application-level flow routes to each switch, then generating traffic as described later in this section. In the live prototype, OpenFlow also provides the traffic matrix (flow-specific counters), port counters, and port power control. OpenFlow enables us to evaluate ElasticTree on switches from different vendors, with no source code changes.

NOX: NOX is a centralized platform that provides network visibility and control atop a network of OpenFlow switches [13]. The logical modules in ElasticTree are implemented as a NOX application. The application pulls flow and port counters,

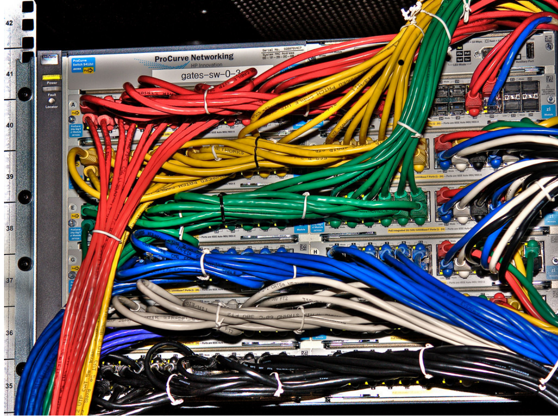


Figure 5: Hardware Testbed (HP switch for $k = 6$ fat tree)

Vendor	Model	k	Virtual Switches	Ports	Hosts
HP	5400	6	45	270	54
Quanta	LB4G	4	20	80	16
NEC	IP8800	4	20	80	16

Table 3: Fat Tree Configurations

directs these to an optimizer, and then adjusts flow routes and port status based on the computed subset. In our current setup, we do not power off inactive switches, due to the fact that our switches are virtual switches. However, in a real data center deployment, we can leverage any of the existing mechanisms such as command line interface, SNMP or newer control mechanisms such as power-control over OpenFlow in order to support the power control features.

2.4 Prototype Testbed

We build multiple testbeds to verify and evaluate ElasticTree, summarized in Table 3, with an example shown in Figure 5. Each configuration multiplexes many smaller virtual switches (with 4 or 6 ports) onto one or more large physical switches. All communication between virtual switches is done over direct links (not through any switch backplane or intermediate switch).

The smaller configuration is a complete $k = 4$ three-layer homogeneous fat tree[4], split into 20 independent four-port virtual switches, supporting 16 nodes at 1 Gbps apiece. One instantiation comprised 2 NEC IP8800 24-port switches and 1 48-port switch, running OpenFlow v0.8.9 firmware provided by NEC Labs. Another comprised two Quanta LB4G 48-port switches, running the OpenFlow Reference Broadcom firmware.

[4]Refer [1] for details on fat trees and definition of k

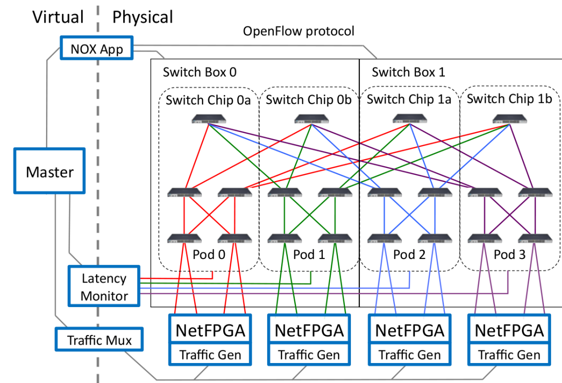


Figure 6: Measurement Setup

The larger configuration is a complete $k = 6$ three-layer fat tree, split into 45 independent six-port virtual switches, supporting 54 hosts at 1 Gbps apiece. This configuration runs on one 288-port HP ProCurve 5412 chassis switch or two 144-port 5406 chassis switches, running OpenFlow v0.8.9 firmware provided by HP Labs.

2.5 Measurement Setup

Evaluating ElasticTree requires infrastructure to generate a small data center's worth of traffic, plus the ability to concurrently measure packet drops and delays. To this end, we have implemented a NetFPGA based traffic generator and a dedicated latency monitor. The measurement architecture is shown in Figure 6.

NetFPGA Traffic Generators. The NetFPGA Packet Generator provides deterministic, line-rate traffic generation for all packet sizes [28]. Each NetFPGA emulates four servers with 1GE connections. Multiple traffic generators combine to emulate a larger group of independent servers: for the k=6 fat tree, 14 NetFPGAs represent 54 servers, and for the k=4 fat tree,4 NetFPGAs represent 16 servers.

At the start of each test, the traffic distribution for each port is packed by a weighted round robin scheduler into the packet generator SRAM. All packet generators are synchronized by sending one packet through an Ethernet control port; these control packets are sent consecutively to minimize the start-time variation. After sending traffic, we poll and store the transmit and receive counters on the packet generators.

Latency Monitor. The latency monitor PC sends tracer packets along each packet path. Tracers enter and exit through a different port on the same physical switch chip; there is one Ethernet port on the latency monitor PC per switch chip. Packets are

logged by Pcap on entry and exit to record precise timestamp deltas. We report median figures that are averaged over all packet paths. To ensure measurements are taken in steady state, the latency monitor starts up after 100 ms. This technique captures all but the last-hop egress queuing delays. Since edge links are never oversubscribed for our traffic patterns, the last-hop egress queue should incur no added delay.

3. POWER SAVINGS ANALYSIS

In this section, we analyze ElasticTree's network energy savings when compared to an always-on baseline. Our comparisons assume a homogeneous fat tree for simplicity, though the evaluation also applies to full-bisection-bandwidth topologies with aggregation, such as those with 1G links at the edge and 10G at the core. The primary metric we inspect is *% original network power*, computed as:

$$= \frac{\text{Power consumed by ElasticTree} \times 100}{\text{Power consumed by original fat-tree}}$$

This percentage gives an accurate idea of the overall power saved by turning off switches and links (i.e., savings equal 100 - *% original power*). We use power numbers from switch model A (§1.3) for both the baseline and ElasticTree cases, and only include active (powered-on) switches and links for ElasticTree cases. Since all three switches in Table 1 have an idle:active ratio of 3:1 (explained in §2.1), using power numbers from switch model B or C will yield similar network energy savings. Unless otherwise noted, optimizer solutions come from the greedy bin-packing algorithm, with flow splitting disabled (as explained in Section 2). We validate the results for all $k = \{4, 6\}$ fat tree topologies on multiple testbeds. For all communication patterns, the measured bandwidth as reported by receive counters matches the expected values. We only report energy saved directly from the network; extra energy will be required to power on and keep running the servers hosting ElasticTree modules. There will be additional energy required for cooling these servers, and at the same time, powering off unused switches will result in cooling energy savings. We do not include these extra costs/savings in this paper.

3.1 Traffic Patterns

Energy, performance and robustness all depend heavily on the traffic pattern. We now explore the possible energy savings over a wide range of communication patterns, leaving performance and robustness for §4.

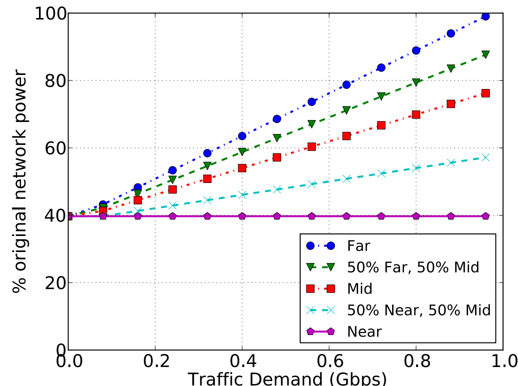


Figure 7: Power savings as a function of demand, with varying traffic locality, for a 28K-node, k=48 fat tree

3.1.1 Uniform Demand, Varying Locality

First, consider two extreme cases: *near* (highly localized) traffic matrices, where servers communicate only with other servers through their edge switch, and *far* (non-localized) traffic matrices where servers communicate only with servers in other pods, through the network core. In this pattern, all traffic stays within the data center, and none comes from outside. Understanding these extreme cases helps to quantify the range of network energy savings. Here, we use the formal method as the optimizer in ElasticTree.

Near traffic is a best-case — leading to the largest energy savings — because ElasticTree will reduce the network to the minimum spanning tree, switching off all but one core switch and one aggregation switch per pod. On the other hand, *far* traffic is a worst-case — leading to the smallest energy savings — because every link and switch in the network is needed. For *far* traffic, the savings depend heavily on the network utilization, $u = \frac{\sum_i \sum_j \lambda_{ij}}{\text{Total hosts}}$ (λ_{ij} is the traffic from host i to host j, $\lambda_{ij} < 1$ Gbps). If u is close to 100%, then all links and switches must remain active. However, with lower utilization, traffic can be concentrated onto a smaller number of core links, and unused ones switch off. Figure 7 shows the potential savings as a function of utilization for both extremes, as well as traffic to the aggregation layer *Mid*), for a $k = 48$ fat tree with roughly 28K servers. Running ElasticTree on this configuration, with *near* traffic at low utilization, we expect a network energy reduction of 60%; we cannot save any further energy, as the active network subset in this case is the MST. For *far* traffic and u=100%, there are no energy savings. This graph highlights the power benefit of local communications, but more im-

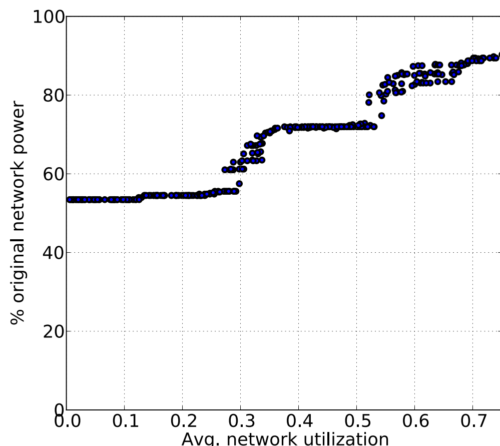


Figure 8: Scatterplot of power savings with random traffic matrix. Each point on the graph corresponds to a pre-configured average data center workload, for a $k = 6$ fat tree

portantly, shows potential savings in all cases. Having seen these two extremes, we now consider more realistic traffic matrices with a mix of both *near* and *far* traffic.

3.1.2 *Random Demand*

Here, we explore how much energy we can expect to save, on average, with random, admissible traffic matrices. Figure 8 shows energy saved by ElasticTree (relative to the baseline) for these matrices, generated by picking flows uniformly and randomly, then scaled down by the most oversubscribed host's traffic to ensure admissibility. As seen previously, for low utilization, ElasticTree saves roughly 60% of the network power, regardless of the traffic matrix. As the utilization increases, traffic matrices with significant amounts of *far* traffic will have less room for power savings, and so the power saving decreases. The two large steps correspond to utilizations at which an extra aggregation switch becomes necessary across *all* pods. The smaller steps correspond to individual aggregation or core switches turning on and off. Some patterns will densely fill all available links, while others will have to incur the entire power cost of a switch for a single link; hence the variability in some regions of the graph. Utilizations above 0.75 are not shown; for these matrices, the greedy bin-packer would sometimes fail to find a complete satisfying assignment of flows to links.

3.1.3 *Sine-wave Demand*

As seen before (§1.2), the utilization of a data center will vary over time, on daily, seasonal and annual

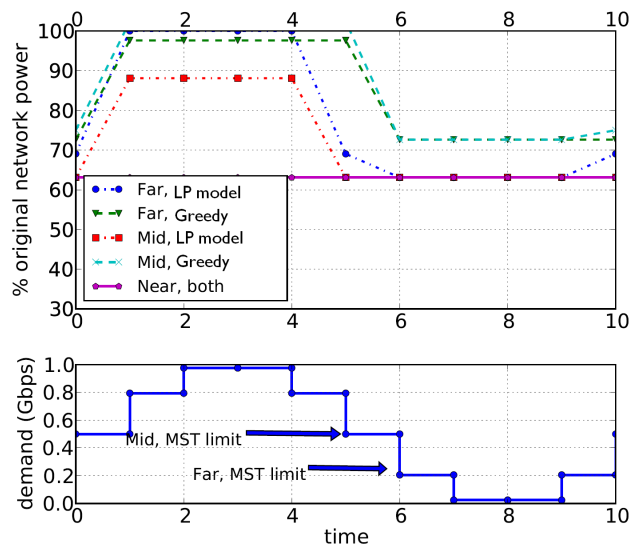


Figure 9: Power savings for sinusoidal traffic variation in a $k = 4$ fat tree topology, with 1 flow per host in the traffic matrix. The input demand has 10 discrete values.

time scales. Figure 9 shows a time-varying utilization; power savings from ElasticTree that follow the utilization curve. To crudely approximate diurnal variation, we assume $u = 1/2(1 + \sin(t))$, at time t, suitably scaled to repeat once per day. For this sine wave pattern of traffic demand, the network power can be reduced up to 64% of the original power consumed, without being over-subscribed and causing congestion.

We note that most energy savings in all the above communication patterns comes from powering off switches. Current networking devices are far from being energy proportional, with even completely idle switches (0% utilization) consuming 70-80% of their fully loaded power (100% utilization) [22]; thus powering off switches yields the most energy savings.

3.1.4 *Traffic in a Realistic Data Center*

In order to evaluate energy savings with a real data center workload, we collected system and network traces from a production data center hosting an e-commerce application (Trace 1, §1). The servers in the data center are organized in a tiered model as application servers, file servers and database servers. The System Activity Reporter (sar) toolkit available on Linux obtains CPU, memory and network statistics, including the number of bytes transmitted and received from 292 servers. Our traces contain statistics averaged over a 10-minute interval and span 5 days in April 2008. The aggregate traffic through all the servers varies between 2 and 12 Gbps at any given time instant (Figure 2). Around 70% of the

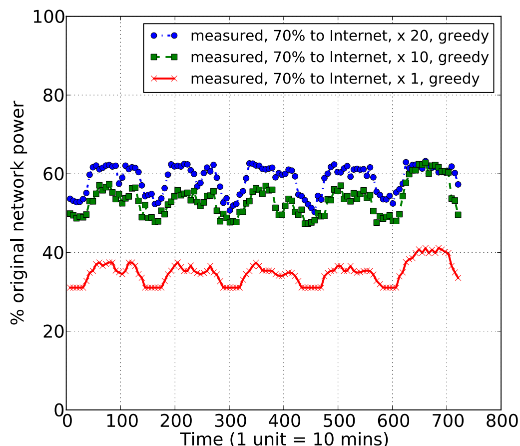


Figure 10: Energy savings for production data center (e-commerce website) traces, over a 5 day period, using a k=12 fat tree. We show savings for different levels of overall traffic, with 70% destined outside the DC.

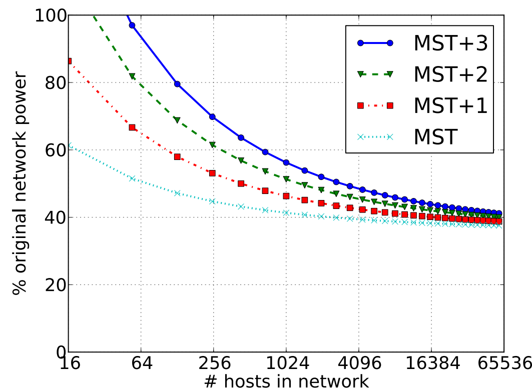


Figure 11: Power cost of redundancy

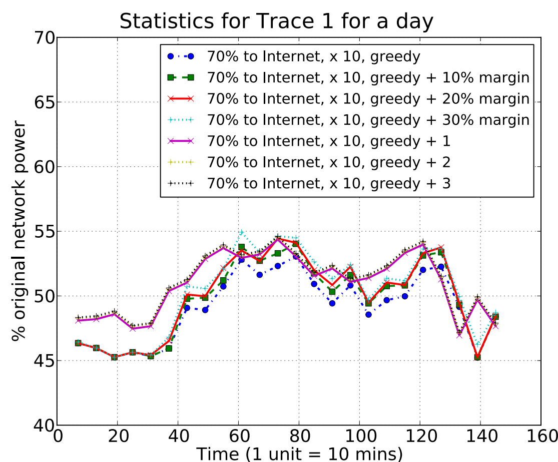


Figure 12: Power consumption in a robust data center network with safety margins, as well as redundancy. Note "greedy+1" means we add a MST over the solution returned by the greedy solver.

traffic leaves the data center and the remaining 30% is distributed to servers within the data center.

In order to compute the energy savings from ElasticTree for these 292 hosts, we need a $k = 12$ fat tree. Since our testbed only supports $k = 4$ and $k = 6$ sized fat trees, we simulate the effect of ElasticTree using the greedy bin-packing optimizer on these traces. A fat tree with $k = 12$ can support up to 432 servers; since our traces are from 292 servers, we assume the remaining 140 servers have been powered off. The edge switches associated with these powered-off servers are assumed to be powered off; we do not include their cost in the baseline routing power calculation.

The e-commerce service does not generate enough network traffic to require a high bisection bandwidth topology such as a fat tree. However, the time-varying characteristics are of interest for evaluating ElasticTree, and should remain valid with proportionally larger amounts of network traffic. Hence, we scale the traffic up by a factor of 20.

For different scaling factors, as well as for different intra data center versus outside data center (external) traffic ratios, we observe energy savings ranging from 25-62%. We present our energy savings results in Figure 10. The main observation when visually comparing with Figure 2 is that the power consumed by the network follows the traffic load curve. Even though individual network devices are not energy-proportional, ElasticTree introduces energy proportionality into the network.

We stress that network energy savings are workload dependent. While we have explored savings in the best-case and worst-case traffic scenarios as well as using traces from a production data center, a highly utilized and "never-idle" data center network would not benefit from running ElasticTree.

3.2 Robustness Analysis

Typically data center networks incorporate some level of capacity margin, as well as redundancy in the topology, to prepare for traffic surges and network failures. In such cases, the network uses more switches and links than essential for the regular production workload.

Consider the case where only a minimum spanning

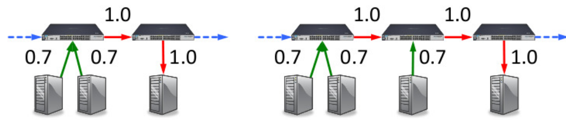


Figure 13: Queue Test Setups with one (left) and two (right) bottlenecks

Bottlenecks	Median	Std. Dev
0	36.00	2.94
1	473.97	7.12
2	914.45	10.50

Table 4: Latency baselines for Queue Test Setups

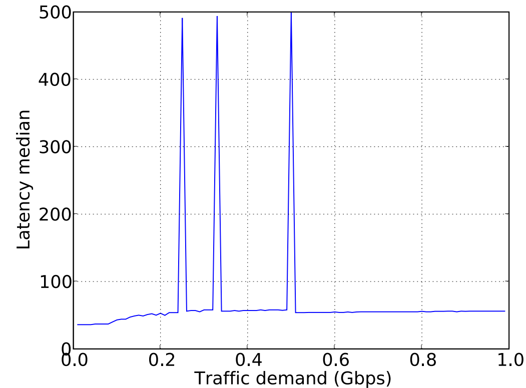


Figure 14: Latency vs demand, with uniform traffic.

tree (MST) in the fat tree topology is turned on (all other links/switches are powered off); this subset certainly minimizes power consumption. However, it also throws away all path redundancy, and with it, *all* fault tolerance. In Figure 11, we extend the MST in the fat tree with additional active switches, for varying topology sizes. The MST+1 configuration requires one additional edge switch per pod, and one additional switch in the core, to enable any single aggregation or core-level switch to fail without disconnecting a server. The MST+2 configuration enables any two failures in the core or aggregation layers, with no loss of connectivity. As the network size increases, the incremental cost of additional fault tolerance becomes an insignificant part of the total network power. For the largest networks, the savings reduce by only 1% for each additional spanning tree in the core aggregation levels. Each +1 increment in redundancy has an *additive* cost, but a *multiplicative* benefit; with MST+2, for example, the failures would have to happen in the same pod to disconnect a host. This graph shows that the added cost of fault tolerance is low.

Figure 12 presents power figures for the k=12 fat tree topology when we add safety margins for accommodating bursts in the workload. We observe that the additional power cost incurred is minimal, while improving the network's ability to absorb unexpected traffic surges.

4. PERFORMANCE

The power savings shown in the previous section are worthwhile only if the performance penalty is negligible. In this section, we quantify the performance degradation from running traffic over a network subset, and show how to mitigate negative effects with a safety margin.

4.1 Queuing Baseline

Figure 13 shows the setup for measuring the buffer depth in our test switches; when queuing occurs, this knowledge helps to estimate the number of hops where packets are delayed. In the congestion-free case (not shown), a dedicated latency monitor PC sends tracer packets into a switch, which sends it right back to the monitor. Packets are timestamped by the kernel, and we record the latency of each received packet, as well as the number of drops. This test is useful mainly to quantify PC-induced latency variability. In the single-bottleneck case, two hosts send 0.7 Gbps of constant-rate traffic to a single switch output port, which connects through a second switch to a receiver. Concurrently with the packet generator traffic, the latency monitor sends tracer packets. In the double-bottleneck case, three hosts send 0.7 Gbps, again while tracers are sent.

Table 4 shows the latency distribution of tracer packets sent through the Quanta switch, for all three cases. With no background traffic, the baseline latency is 36 us. In the single-bottleneck case, the egress buffer fills immediately, and packets experience 474 us of buffering delay. For the double-bottleneck case, most packets are delayed twice, to 914 us, while a smaller fraction take the single-bottleneck path. The HP switch (data not shown) follows the same pattern, with similar minimum latency and about 1500 us of buffer depth. All cases show low measurement variation.

4.2 Uniform Traffic, Varying Demand

In Figure 14, we see the latency totals for a uniform traffic series where all traffic goes through the core to a different pod, and every hosts sends one flow. To allow the network to reach steady state, measurements start 100 ms after packets are sent,

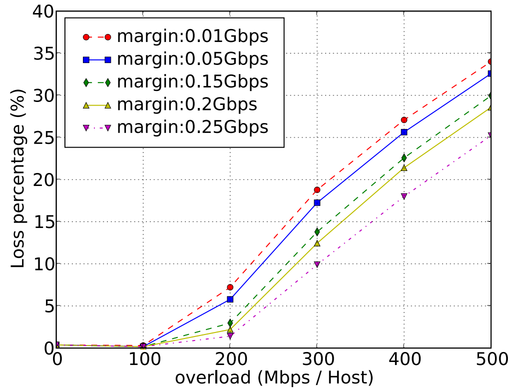


Figure 15: Drops vs overload with varying safety margins

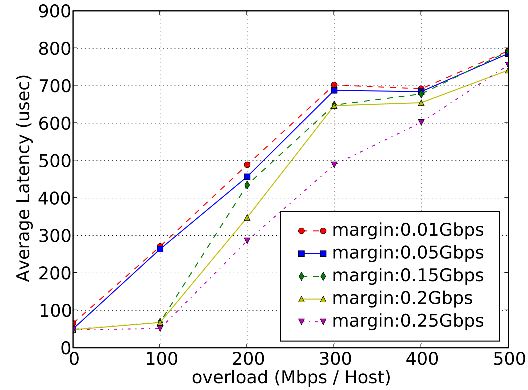


Figure 16: Latency vs overload with varying safety margins

and continue until the end of the test, 900 ms later. All tests use 512-byte packets; other packet sizes yield the same results. The graph covers packet generator traffic from idle to 1 Gbps, while tracer packets are sent along every flow path. If our solution is feasible, that is, all flows on each link sum to less than its capacity, then we will see no dropped packets, with a consistently low latency.

Instead, we observe sharp spikes at 0.25 Gbps, 0.33 Gbps, and 0.5 Gbps. These spikes correspond to points where the available link bandwidth is exceeded, even by a small amount. For example, when ElasticTree compresses four 0.25 Gbps flows along a single 1 Gbps link, Ethernet overheads (preamble, inter-frame spacing, and the CRC) cause the egress buffer to fill up. Packets either get dropped or significantly delayed.

This example motivates the need for a safety margin to account for processing overheads, traffic bursts, and sustained load increases. The issue is not just that drops occur, but also that *every* packet on an overloaded link experiences significant delay. Next, we attempt to gain insight into how to set the safety margin, or capacity reserve, such that performance stays high up to a known traffic overload.

4.3 Setting Safety Margins

Figures 15 and 16 show drops and latency as a function of traffic overload, for varying safety margins. *Safety margin* is the amount of capacity reserved at *every* link by the optimizer; a higher safety margin provides performance insurance, by delaying the point at which drops start to occur, and average latency starts to degrade. *Traffic overload* is the amount each host sends and receives beyond the original traffic matrix. The overload for a host is spread evenly across all flows sent by that host. For example, at zero overload, a solution with a safety margin of 100 Mbps will prevent more than 900 Mbps of combined flows from crossing each link. If a host sends 4 flows (as in these plots) at 100 Mbps overload, each flow is boosted by 25 Mbps. Each data point represents the average over 5 traffic matrices. In all matrices, each host sends to 4 randomly chosen hosts, with a total outgoing bandwidth selected uniformly between 0 and 0.5 Gbps. All tests complete in one second.

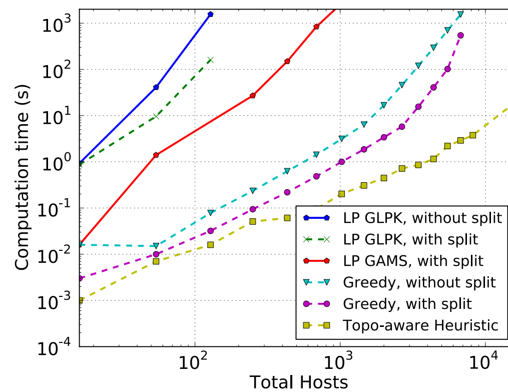


Figure 17: Computation time for different optimizers as a function of network size

Drops Figure 15 shows no drops for small overloads (up to 100 Mbps), followed by a steadily increasing drop percentage as overload increases. Loss percentage levels off somewhat after 500 Mbps, as some flows cap out at 1 Gbps and generate no extra traffic. As expected, increasing the safety margin defers the point at which performance degrades.

Latency In Figure 16, latency shows a trend similar to drops, except when overload increases to 200 Mbps, the performance effect is more pronounced. For the 250 Mbps margin line, a 200 Mbps overload results in 1% drops, however latency increases by 10x due to the few congested links. Some margin lines cross at high overloads; this is not to say that a smaller margin is outperforming a larger one, since drops increase, and we ignore those in the latency calculation.

Interpretation Given these plots, a network operator can choose the safety margin that best balances the competing goals of performance and energy efficiency. For example, a network operator might observe from past history that the traffic average never varies by more than 100 Mbps in any 10 minute span. She considers an average latency under 100 us to be acceptable. Assuming that ElasticTree can transition to a new subset every 10 minutes, the operator looks at 100 Mbps overload on each plot. She then finds the smallest safety margin with sufficient performance, which in this case is 150 Mbps. The operator can then have some assurance that if the traffic changes as expected, the network will meet her performance criteria, while consuming the minimum amount of power.

5. PRACTICAL CONSIDERATIONS

Here, we address some of the practical aspects of deploying ElasticTree in a live data center environment.

5.1 Comparing various optimizers

We first discuss the scalability of various optimizers in ElasticTree, based on solution time vs network size, as shown in Figure 17. This analysis provides a sense of the feasibility of their deployment in a real data center. The formal model produces solutions closest to optimal; however for larger topologies (such as fat trees with $k >= 14$), the time to find the optimal solution becomes intractable. For example, finding a network subset with the formal model with flow splitting enabled on CPLEX on a single core, 2 Ghz machine, for a $k = 16$ fat tree, takes about an hour. The solution time growth of this carefully optimized model is about $O(n^{3.5})$, where n is the number of hosts. We then ran the greedy-bin packer (written in unoptimized Python) on a single core of a 2.13 Ghz laptop with 3 GB of RAM. The no-split version scaled as about $O(n^{2.5})$, while the with-split version scaled slightly better, as $O(n^2)$. The topology-aware heuristic fares much better, scaling as roughly $O(n)$, as expected. Subset computation for 10K hosts takes less than 10 seconds for a single-core, unoptimized, Python implementation – faster than the fastest switch boot time we observed (30 seconds for the Quanta switch). This result implies that the topology-aware heuristic approach is not fundamentally unscalable, especially considering that the number of operations increases linearly with the number of hosts. We next describe in detail the topology-aware heuristic, and show how small modifications to its “starter subset” can yield high-quality, practical network solutions, in little time.

5.2 Topology-Aware Heuristic

We describe precisely how to calculate the subset of active network elements using only port counters.

Links. First, compute $LEdge^{up}_{p,e}$, the minimum number of active links exiting edge switch e in pod p to support up-traffic (edge → agg):

$$LEdge^{up}_{p,e} = \lceil (\sum_{a \in A_p} F(e \rightarrow a))/r \rceil$$

A_p is the set of aggregation switches in pod p, $F(e \rightarrow a)$ is the traffic flow from edge switch e to aggregation switch a, and r is the link rate. The total up-traffic of e, divided by the link rate, equals the minimum number of links from e required to satisfy the up-traffic bandwidth. Similarly, compute $LEdge^{down}_{p,e}$, the number of active links exiting edge switch e in pod p to support down-traffic (agg → edge):

$$LEdge^{down}_{p,e} = \lceil (\sum_{a \in A_p} F(a \rightarrow e))/r \rceil$$

The maximum of these two values (plus 1, to ensure a spanning tree at idle) gives $LEdge_{p,e}$, the minimum number of links for edge switch e in pod p:

$$LEdge_{p,e} = \max\{LEdge^{up}_{p,e}, LEdge^{down}_{p,e}, 1\}$$

Now, compute the number of active links from each pod to the core. $LAgg^{up}_{p}$ is the minimum number of links from pod p to the core to satisfy the up-traffic bandwidth (agg → core):

$$LAgg^{up}_{p} = \lceil (\sum_{c \in C, a \in A_p, a \rightarrow c} F(a \rightarrow c))/r \rceil$$

Hence, we find the number of up-*links*, $LAgg^{down}_{p}$ used to support down-traffic (core → agg) in pod p:

$$LAgg^{down}_{p} = \lceil (\sum_{c \in C, a \in A_p, c \rightarrow a} F(c \rightarrow a))/r \rceil$$

The maximum of these two values (plus 1, to ensure a spanning tree at idle) gives $LAgg_p$, the mini-

mum number of core links for pod p:

$$LAgg_p = \max\{LEdge_p^{up}, LEdge_p^{down}\}$$

Switches. For both the aggregation and core layers, the number of switches follows directly from the link calculations, as every active link must connect to an active switch. First, we compute $NAgg_p^{up}$, the minimum number of aggregation switches required to satisfy up-traffic (edge → agg) in pod p:

$$NAgg_p^{up} = \max_{e \in E_p}\{LEdge_{p,e}^{up}\}$$

Next, compute $NAgg_p^{down}$, the minimum number of aggregation switches required to support down-traffic (core → agg) in pod p:

$$NAgg_p^{down} = \lceil (LAgg_p^{down}/(k/2) \rceil$$

C is the set of core switches and k is the switch degree. The number of core links in the pod, divided by the number of links uplink in each aggregation switch, equals the minimum number of aggregation switches required to satisfy the bandwidth demands from all core switches. The maximum of these two values gives $NAgg_p$, the minimum number of active aggregation switches in the pod:

$$NAgg_p = \max\{NAgg_p^{up}, NAgg_p^{down}, 1\}$$

Finally, the traffic between the core and the most-active pod informs $NCore$, the number of core switches that must be active to satisfy the traffic demands:

$$NCore = \lceil \max_{p \in P}(LAgg_p^{up}) \rceil$$

Robustness. The equations above assume that 100% utilized links are acceptable. We can change r, the link rate parameter, to set the desired average link utilization. Reducing r reserves additional resources to absorb traffic overloads, plus helps to reduce queuing delay. Further, if hashing is used to balance flows across different links, reducing r helps account for collisions.

To add k-redundancy to the starter subset for improved fault tolerance, add k aggregation switches to each pod and the core, plus activate the links on all added switches. Adding k-redundancy can be thought of as adding k parallel MSTs that overlap at the edge switches. These two approaches can be combined for better robustness.

5.3 Response Time

The ability of ElasticTree to respond to spikes in traffic depends on the time required to gather statistics, compute a solution, wait for switches to boot, enable links, and push down new routes. We measured the time required to power on/off links and switches in real hardware and find that the dominant time is waiting for the switch to boot up, which ranges from 30 seconds for the Quanta switch to about 3 minutes for the HP switch. Powering individual ports on and off takes about $1-3$ seconds. Populating the entire flow table on a switch takes under 5 seconds, while reading all port counters takes less than 100 ms for both. Switch models in the future may support features such as going into various sleep modes; the time taken to wake up from sleep modes will be significantly faster than booting up. ElasticTree can then choose which switches to power off versus which ones to put to sleep.

Further, the ability to predict traffic patterns for the next few hours for traces that exhibit regular behavior will allow network operators to plan ahead and get the required capacity (plus some safety margin) ready in time for the next traffic spike. Alternately, a control loop strategy to address performance effects from burstiness would be to dynamically increase the safety margin whenever a threshold set by a service-level agreement policy were exceeded, such as a percentage of packet drops.

5.4 Traffic Prediction

In all of our experiments, we input the entire traffic matrix to the optimizer, and thus assume that we have complete prior knowledge of incoming traffic. In a real deployment of ElasticTree, such an assumption is unrealistic. One possible workaround is to predict the incoming traffic matrix based on historical traffic, in order to plan ahead for expected traffic spikes or long-term changes. While prediction techniques are highly sensitive to workloads, they are more effective for traffic that exhibit regular patterns, such as our production data center traces (§3.1.4). We experiment with a simple auto regressive AR(1) prediction model in order to predict traffic to and from each of the 292 servers. We use traffic traces from the first day to train the model, then use this model to predict traffic for the entire 5 day period. Using the traffic prediction, the greedy bin-packer can determine an active topology subset as well as flow routes.

While detailed traffic prediction and analysis are beyond the scope of this paper, our initial experimental results are encouraging. They imply that even simple prediction models can be used for data center traffic that exhibits periodic (and thus predictable) behavior.

5.5 Fault Tolerance

ElasticTree modules can be placed in ways that mitigate fault tolerance worries. In our testbed, the

routing and optimizer modules run on a single host PC. This arrangement ties the fate of the whole system to that of each module; an optimizer crash is capable of bringing down the system.

Fortunately, the topology-aware heuristic – the optimizer most likely to be deployed – operates independently of routing. The simple solution is to move the optimizer to a separate host to prevent slow computation or crashes from affecting routing. Our OpenFlow switches support a passive listening port, to which the read-only optimizer can connect to grab port statistics. After computing the switch/link subset, the optimizer must send this subset to the routing controller, which can apply it to the network. If the optimizer doesn't check in within a fixed period of time, the controller should bring all switches up. The reliability of ElasticTree should be no worse than the optimizer-less original; the failure condition brings back the original network power, plus a time period with reduced network capacity.

For optimizers tied to routing, such as the formal model and greedy bin-packer, known techniques can provide controller-level fault tolerance. In active standby, the primary controller performs all required tasks, while the redundant controllers stay idle. On failing to receive a periodic heartbeat from the primary, a redundant controller becomes to the new primary. This technique has been demonstrated with NOX, so we expect it to work with our system. In the more complicated full replication case, multiple controllers are simultaneously active, and state (for routing and optimization) is held consistent between them. For ElasticTree, the optimization calculations would be spread among the controllers, and each controller would be responsible for power control for a section of the network. For a more detailed discussion of these issues, see §3.5 "Replicating the Controller: Fault-Tolerance and Scalability" in [5].

6. RELATED WORK

This paper tries to extend the idea of power proportionality into the network domain, as first described by Barroso *et al.* [4]. Gupta *et al.* [17] were amongst the earliest researchers to advocate conserving energy in networks. They suggested putting network components to sleep in order to save energy and explored the feasibility in a LAN setting in a later paper [18]. Several others have proposed techniques such as putting idle components in a switch (or router) to sleep [18] as well as adapting the link rate [14], including the IEEE 802.3az Task Force [19].

Chabarek *et al.* [6] use mixed integer programming to optimize router power in a wide area network, by choosing the chassis and linecard configuration to best meet the expected demand. In contrast, our formulation optimizes a data center local area network, finds the power-optimal network subset and routing to use, and includes an evaluation of our prototype. Further, we detail the tradeoffs associated with our approach, including impact on packet latency and drops.

Nedevschi *et al.* [26] propose shaping the traffic into small bursts at edge routers to facilitate putting routers to sleep. Their research is complementary to ours. Further, their work addresses edge routers in the Internet while our algorithms are for data centers. In a recent work, Ananthanarayanan [3] *et al.* motivate via simulation two schemes - a lower power mode for ports and time window prediction techniques that vendors can implemented in future switches. While these and other improvements can be made in future switch designs to make them more energy efficient, most energy (70-80% of their total power) is consumed by switches in their idle state. A more effective way of saving power is using a traffic routing approach such as ours to maximize idle switches and power them off. Another recent paper [25] *et al.* discusses the benefits and deployment models of a network proxy that would allow end-hosts to sleep while the proxy keeps the network connection alive.

Other complementary research in data center networks has focused on scalability [24][10], switching layers that can incorporate different policies [20], or architectures with programmable switches [11].

7. DISCUSSION

The idea of disabling critical network infrastructure in data centers has been considered taboo. Any dynamic energy management system that attempts to achieve energy proportionality by powering off a subset of idle components must demonstrate that the active components can still meet the current offered load, as well as changing load in the immediate future. The power savings must be worthwhile, performance effects must be minimal, and fault tolerance must not be sacrificed. The system must produce a feasible set of network subsets that can route to all hosts, and be able to scale to a data center with tens of thousands of servers.

To this end, we have built ElasticTree, which through data-center-wide traffic management and control, introduces energy proportionality in today's non-energy proportional networks. Our initial results (covering analysis, simulation, and hardware prototypes) demonstrate the tradeoffs between per-

formance, robustness, and energy; the safety margin parameter provides network administrators with control over these tradeoffs. ElasticTree's ability to respond to sudden increases in traffic is currently limited by the switch boot delay, but this limitation can be addressed, relatively simply, by adding a sleep mode to switches.

ElasticTree opens up many questions. For example, how will TCP-based application traffic interact with ElasticTree? TCP maintains link utilization in sawtooth mode; a network with primarily TCP flows might yield measured traffic that stays below the threshold for a small safety margin, causing ElasticTree to never increase capacity. Another question is the effect of increasing network size: a larger network probably means more, smaller flows, which pack more densely, and reduce the chance of queuing delays and drops. We would also like to explore the general applicability of the heuristic to other topologies, such as hypercubes and butterflies.

Unlike choosing between cost, speed, and reliability when purchasing a car, with ElasticTree one doesn't have to pick just two when offered performance, robustness, and energy efficiency. During periods of low to mid utilization, and for a variety of communication patterns (as is often observed in data centers), ElasticTree can maintain the robustness and performance, while lowering the energy bill.

8. ACKNOWLEDGMENTS

The authors want to thank their shepherd, Ant Rowstron, for his advice and guidance in producing the final version of this paper, as well as the anonymous reviewers for their feedback and suggestions. Xiaoyun Zhu (VMware) and Ram Swaminathan (HP Labs) contributed to the problem formulation; Parthasarathy Ranganathan (HP Labs) helped with the initial ideas in this paper. Thanks for OpenFlow switches goes to Jean Tourrilhes and Praveen Yalagandula at HP Labs, plus the NEC IP8800 team. Greg Chesson provided the Google traces.

9. REFERENCES

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *ACM SIGCOMM*, pages 63–74, 2008.

[2] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *USENIX NSDI*, April 2010.

[3] G. Ananthanarayanan and R. Katz. Greening the Switch. In *Proceedings of HotPower*, December 2008.

[4] L. A. Barroso and U. Hölzle. The Case for Energy-Proportional Computing. *Computer*, 40(12):33–37, 2007.

[5] M. Casado, M. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, page 12. ACM, 2007.

[6] J. Chabarek, J. Sommers, P. Barford, C. Estan, D. Tsiang, and S. Wright. Power Awareness in Network Design and Routing. In *IEEE INFOCOM*, April 2008.

[7] U.S. Environmental Protection Agency's Data Center Report to Congress. `http://tinyurl.com/2jz3ft`.

[8] S. Even, A. Itai, and A. Shamir. On the Complexity of Time Table and Multi-Commodity Flow Problems. In *16th Annual Symposium on Foundations of Computer Science*, pages 184–193, October 1975.

[9] A. Greenberg, J. Hamilton, D. Maltz, and P. Patel. The Cost of a Cloud: Research Problems in Data Center Networks. In *ACM SIGCOMM CCR*, January 2009.

[10] A. Greenberg, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *ACM SIGCOMM*, August 2009.

[11] A. Greenberg, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Towards a Next Generation Data Center Architecture: Scalability and Commoditization. In *ACM PRESTO*, pages 57–62, 2008.

[12] D. Grunwald, P. Levis, K. Farkas, C. M. III, and M. Neufeld. Policies for Dynamic Clock Scheduling. In *OSDI*, 2000.

[13] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, and N. McKeown. NOX: Towards an Operating System for Networks. In *ACM SIGCOMM CCR*, July 2008.

[14] C. Gunaratne, K. Christensen, B. Nordman, and S. Suen. Reducing the Energy Consumption of Ethernet with Adaptive Link Rate (ALR). *IEEE Transactions on Computers*, 57:448–461, April 2008.

[15] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. In *ACM SIGCOMM*, August 2009.

[16] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. DCell: A Scalable and Fault-Tolerant Network Structure for Data Centers. In *ACM SIGCOMM*, pages 75–86, 2008.

[17] M. Gupta and S. Singh. Greening of the internet. In *ACM SIGCOMM*, pages 19–26, 2003.

[18] M. Gupta and S. Singh. Using Low-Power Modes for Energy Conservation in Ethernet LANs. In *IEEE INFOCOM*, May 2007.

[19] IEEE 802.3az. `ieee802.org/3/az/public/index.html`.

[20] D. A. Joseph, A. Tavakoli, and I. Stoica. A Policy-aware Switching Layer for Data Centers. *SIGCOMM Comput. Commun. Rev.*, 38(4):51–62, 2008.

[21] S. Kandula, D. Katabi, S. Sinha, and A. Berger. Dynamic Load Balancing Without Packet Reordering. *SIGCOMM Comput. Commun. Rev.*, 37(2):51–62, 2007.

[22] P. Mahadevan, P. Sharma, S. Banerjee, and P. Ranganathan. A Power Benchmarking Framework for Network Devices. In *Proceedings of IFIP Networking*, May 2009.

[23] J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. C. Mogul. SPAIN: COTS Data-Center Ethernet for Multipathing over Arbitrary Topologies. In *USENIX NSDI*, April 2010.

[24] R. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric. In *ACM SIGCOMM*, August 2009.

[25] S. Nedevschi, J. Chandrashenkar, B. Nordman, S. Ratnasamy, and N. Taft. Skilled in the Art of Being Idle: Reducing Energy Waste in Networked Systems. In *Proceedings Of NSDI*, April 2009.

[26] S. Nedevschi, L. Popa, G. Iannaccone, S. Ratnasamy, and D. Wetherall. Reducing Network Energy Consumption via Sleeping and Rate-Adaptation. In *Proceedings of the 5th USENIX NSDI*, pages 323–336, 2008.

[27] Cisco IOS NetFlow. `http://www.cisco.com/web/go/netflow`.

[28] NetFPGA Packet Generator. `http://tinyurl.com/ygcupdc`.

[29] The OpenFlow Switch. `http://www.openflowswitch.org`.

[30] C. Patel, C. Bash, R. Sharma, M. Beitelmam, and R. Friedrich. Smart Cooling of data Centers. In *Proceedings of InterPack*, July 2003.

APPENDIX

A. POWER OPTIMIZATION PROBLEM

Our model is a multi-commodity flow formulation, augmented with binary variables for the power state of links and switches. It minimizes the total network power by solving a mixed-integer linear program.

A.1 Multi-Commodity Network Flow

Flow network $G(V, E)$, has edges $(u, v) \in E$ with capacity $c(u, v)$. There are k commodities $K_1, K_2, \ldots, K_k$, defined by $K_i = (s_i, t_i, d_i)$, where, for commodity i, s_i is the source, t_i is the sink, and d_i is the demand. The flow of commodity i along edge (u, v) is $f_i(u, v)$. Find a flow assignment which satisfies the following three constraints [8]:

Capacity constraints: The total flow along each link must not exceed the edge capacity.

$$\forall (u,v) \in V, \sum_{i=1}^{k} f_i(u,v) \leq c(u,v)$$

Flow conservation: Commodities are neither created nor destroyed at intermediate nodes.

$$\forall i, \sum_{w \in V} f_i(u,w) = 0, \text{when } u \neq s_i \text{ and } u \neq t_i$$

Demand satisfaction: Each source and sink sends or receives an amount equal to its demand.

$$\forall i, \sum_{w \in V} f_i(s_i,w) = \sum_{w \in V} f_i(w,t_i) = d_i$$

A.2 Power Minimization Constraints

Our formulation uses the following notation:

S	Set of all switches
V_u	Set of nodes connected to a switch u
$a(u,v)$	Power cost for link (u,v)
$b(u)$	Power cost for switch u
$X_{u,v}$	Binary decision variable indicating whether link (u,v) is powered ON
Y_u	Binary decision variable indicating whether switch u is powered ON
E_i	Set of all unique edges used by flow i
$r_i(u,v)$	Binary decision variable indicating whether commodity i uses link (u,v)

The objective function, which minimizes the total network power consumption, can be represented as:

Minimize $\sum_{(u,v)\in E} X_{u,v} \times a(u,v) + \sum_{u \in V} Y_u \times b(u)$

The following additional constraints create a dependency between the flow routing and power states:

Deactivated links have no traffic: Flow is restricted to only those links (and consequently the switches) that are powered on. Thus, for all links (u, v) used by commodity i, $f_i(u, v) = 0$, when $X_{u,v} = 0$. Since the flow variable f is positive in our formulation, the linearized constraint is:

$$\forall i, \forall (u,v) \in E, \sum_{i=1}^{k} f_i(u,v) \leq c(u,v) \times X_{u,v}$$

The optimization objective inherently enforces the converse, which states that links with no traffic can be turned off.

Link power is bidirectional: Both "halves" of an Ethernet link must be powered on if traffic is flowing in either direction:

$$\forall (u,v) \in E, X_{u,v} = X_{v,u}$$

Correlate link and switch decision variable: When a switch u is powered off, all links connected to this switch are also powered off:

$$\forall u \in V, \forall w \in V_v, X_{u,w} = X_{w,u} \leq Y_u$$

Similarly, when all links connecting to a switch are off, the switch can be powered off. The linearized constraint is:

$$\forall u \in V, Y_u \leq \sum_{w \in V_u} X_{w,u}$$

A.3 Flow Split Constraints

Splitting flows is typically undesirable due to TCP packet reordering effects [21]. We can prevent flow splitting in the above formulation by adopting the following constraint, which ensures that the traffic on link (u, v) of commodity i is equal to either the full demand or zero:

$$\forall i, \forall (u,v) \in E, f_i(u,v) = d_i \times r_i(u,v)$$

The regularity of the fat tree, combined with restricted tree routing, helps to reduce the number of flow split binary variables. For example, each inter-pod flow must go from the aggregation layer to the core, with exactly $(k/2)^2$ path choices. Rather than consider binary variable r for all edges along every possible path, we only consider the set of "unique edges", those at the highest layer traversed. In the inter-pod case, this is the set of aggregation to edge links. We precompute the set of unique edges E_i usable by commodity i, instead of using all edges in E. Note that the flow conservation equations will ensure that a connected set of unique edges are traversed for each flow.

SPAIN: COTS Data-Center Ethernet for Multipathing over Arbitrary Topologies

Jayaram Mudigonda* Jayaram.Mudigonda@hp.com
Praveen Yalagandula* Praveen.Yalagandula@hp.com
Mohammad Al-Fares+ malfares@cs.ucsd.edu
Jeffrey C. Mogul* Jeff.Mogul@hp.com

*HP Labs, Palo Alto, CA 94304 +UC San Diego

Abstract

Operators of data centers want a scalable network fabric that supports high bisection bandwidth and host mobility, but which costs very little to purchase and administer. Ethernet *almost* solves the problem – it is cheap and supports high link bandwidths – but traditional Ethernet does not scale, because its spanning-tree topology forces traffic onto a single tree. Many researchers have described "scalable Ethernet" designs to solve the scaling problem, by enabling the use of multiple paths through the network. However, most such designs require specific wiring topologies, which can create deployment problems, or changes to the network switches, which could obviate the commodity pricing of these parts.

In this paper, we describe SPAIN ("Smart Path Assignment In Networks"). SPAIN provides multipath forwarding using inexpensive, commodity off-the-shelf (COTS) Ethernet switches, over arbitrary topologies. SPAIN precomputes a set of paths that exploit the redundancy in a given network topology, then merges these paths into a set of trees; each tree is mapped as a separate VLAN onto the physical Ethernet. SPAIN requires only minor end-host software modifications, including a simple algorithm that chooses between pre-installed paths to efficiently spread load over the network. We demonstrate SPAIN's ability to improve bisection bandwidth over both simulated and experimental data-center networks.

1 Introduction

Data-center operators often take advantage of scale, both to amortize fixed costs, such as facilities and staff, over many servers, and to allow high-bandwidth, low-latency communications among arbitrarily large sets of machines. They thus desire scalable data-center networks. Data-center operators also must reduce costs for both equipment and operations; commodity off-the-shelf (COTS) components often provide the best total cost of ownership (TCO).

Ethernet is becoming the primary network technology for data centers, especially as protocols such as Fibre Channel over Ethernet (FCoE) begin to allow convergence of all data-center networking onto a single fabric. COTS Ethernet has many nice features, especially ubiquity, self-configuration, and high link bandwidth at low cost, but traditional Layer-2 (L2) Ethernet cannot scale to large data centers. Adding IP (Layer-3) routers "solves" the scaling problem via the use of subnets, but introduces new problems, especially the difficulty of supporting dynamic mobility of virtual machines: the lack of a single flat address space makes it much harder to move a VM between subnets. Also, the use of IP routers instead of Ethernet switches can increase hardware costs and complicate network management.

This is not a new problem; plenty of recent research papers have proposed scalable data-center network designs based on Ethernet hardware. All such proposals address the core scalability problem with traditional Ethernet, which is that to support self-configuration of switches, it forces all traffic into a single spanning tree [28] – even if the physical wired topology provides multiple paths that could, in principle, avoid unnecessary sharing of links between flows.

In Sec. 3, we discuss previous proposals in specific detail. Here, at the risk of overgeneralizing, we assert that SPAIN improves over previous work by providing multipath forwarding using inexpensive, COTS Ethernet switches, over arbitrary topologies, and supporting incremental deployment; we are not aware of previous work that does all four.

Support for COTS switches probably reduces costs, and certainly reduces the time before SPAIN could be deployed, compared to designs that require even small changes to switches. Support for arbitrary topologies is especially important because it allows SPAIN to be used without re-designing the entire physical network, in contrast to designs that require hypercubes, fat-trees, etc., and because there may be no single topology that best meets all needs. Together, both properties allow incremental deployment of SPAIN in an existing data-center network, without reducing its benefits in a purpose-built network. SPAIN can also function without a real-time

central controller, although it may be useful to exploit such a controller to guarantee specific QoS properties.

In SPAIN, an offline network controller system first pre-computes a set of paths that exploit the redundancy in a given network topology, with the goal of utilizing the redundancy in the physical wiring both to provide high bisection bandwidth (low over-subscription), and to support several failover paths between any given pair of hosts. The controller then merges these paths into a set of trees and maps each tree onto a separate VLAN, exploiting the VLAN support in COTS Ethernet switches. In most cases, only a small number of VLANs suffice to cover the physical network.

SPAIN does require modifications to end-host systems, including a simple algorithm that chooses between pre-installed paths to efficiently spread load over the network. These modifications are quite limited; in Linux, they consist of a loadable kernel module and a user-level controller.

We have evaluated SPAIN both in simulation and in an experimental deployment on a network testbed. We show that SPAIN adds virtually no end-host overheads, that it significantly improves bisection bandwidth on a variety of topologies, that it can be deployed incrementally with immediate performance benefits, and that it tolerates faults in the network.

2 Background and goals

Ethernet is known for its ease-of-use. Hosts come with preset addresses and simply need to be plugged in; each network switch automatically learns the locations of other switches and of end hosts. Switches organize themselves into a spanning tree to form loop-free paths between all source-destination pairs. Hence, not surprisingly, Ethernet now forms the basis of virtually all enterprise and data center networks. This popularity made many Ethernet switches an inexpensive commodity and led to continuous improvements. 10Gbps Ethernet is fast becoming commoditized [18], the 40Gbps standard is expected this year [21], and the standardization of 100Gbps is already underway [6].

Network operators would like to be able to scale Ethernet to an entire data center, but it is very difficult to do so, as we detail in section 2.2. Hence, today most such networks are designed as several modest-sized Ethernets (IP subnets), connected by one or two layers of IP routers [2, 3, 10].

2.1 Why we want Ethernet to scale

The use of multiple IP subnets, especially within a data center, creates significant management complexity. In particular, it requires a network administrator to simultaneously and consistently manage the L2 and L3 layers, even though these are based on very different forwarding, control, and administrative mechanisms.

Consider a typical network composed of Ethernet-based IP subnets. This not only requires the configuration of IP subnets and routing protocols—which is considered a hard problem in itself [22]—but also sacrifices the simplicity of Ethernet's plug-and-play operation. For instance, as explained in [2, 3], in such a hybrid network, to allow the end hosts to efficiently reach the IP-routing layer, all Ethernet switches must be configured such that their automatic forwarding table computation is forced to pick only the shortest paths between the IP-routing layer and the hosts.

Dividing a data center into a set of IP subnets has other drawbacks. It imposes the need to configure DHCP servers for each subnet; to design an IP addressing assignment that does not severely fragment the IP address space (especially with IPv4); and makes it hard to deal with topology changes [22]. For example, migrating a live virtual machine from one side of the data center to another, to deal with a cooling imbalance, requires changing that VM's IP address – this can disrupt existing connections.

For these reasons, it becomes very attractive to scale a single Ethernet to connect an entire data center or enterprise.

2.2 Why Ethernet is hard to scale

Ethernet's lack of scalability stems from three main problems:

1. Its use of the Spanning Tree Protocol to automatically ensure a loop-free topology.
2. Packet floods for learning host locations.
3. Host-generated broadcasts, especially for ARP.

We discuss each of these issues.

Spanning tree: Spanning Tree Protocol (STP) [28] was a critical part of the initial success of Ethernet; it allows automatic self-configuration of a set of relatively simple switches. Using STP, all the switches in an L2 domain agree on a subset of links between them, so as to form a spanning tree over all switches. By forwarding packets only on those links, the switches ensure connectivity while eliminating packet-forwarding loops. Otherwise, Ethernet would have had to carry a hop-count or TTL field, which would have created compatibility and implementation challenges.

STP, however, creates significant problems for scalable data-center networks:

- **Limited bisection bandwidth**: Since there is (by definition) only one path through the spanning tree between any pair of hosts, a source-destination pair cannot use multiple paths to achieve the best possible bandwidth. Also, since links on any path are probably shared by many other host pairs, conges-

tion can arise, especially near the designated (high-bandwidth) root switch of the tree. The aggregate throughput of the network can be much lower than the sum of the NIC throughputs.

- **High-cost core switches**: Aggregate throughput can be improved by use of a high-fanout, high-bandwidth switch at the root of the tree. Scaling root-switch bandwidth can be prohibitively expensive [10], especially since this switch must be replicated to avoid a single point of failure for the entire data center. Also, the STP must be properly configured to ensure that the spanning tree actually gets rooted at this expensive switch.
- **Low reliability**: Since the spanning tree leads to lots of sharing at links closer to the root, a failure can affect an unnecessarily large fraction of paths.
- **Reduced flexibility in node placement**: Generally, for a given source-destination pair, the higher the common ancestor in the spanning tree, the higher the number of competing source-destination pairs that share links in the subtree, and thus the lower the throughput that this given pair can achieve. Hence, to ensure adequate throughput, frequently-communicating source-destination pairs must be connected to the same switch, or to neighboring switches with the lowest possible common ancestor. Such restrictions, particularly in case of massive-scale applications that require high server-to-server bandwidth, inhibit flexibility in workload placement or cause substantial performance penalties [10, 18].

SPAIN avoids these problems by employing multiple spanning trees, which can fully exploit the path redundancy in the physical topology, especially if the wiring topology is not a simple tree.

Packet floods: Ethernet's automatic self-configuration is often a virtue: a host can be plugged into a port anywhere in the network, and the switches discover its location by observing the packets it transmits [32]. A switch learns the location of a MAC address by recording, in its learning table, the switch port on which it first sees a packet sent from that address. To support host mobility, switches periodically forget these bindings and re-learn them.

If a host has not sent packets in the recent past, therefore, switches will not know its location. When forwarding a packet whose destination address is not in its learning table, a switch must "flood" the packet on all of its ports in the spanning tree (except on the port the packet arrived on). This flooding traffic can be a serious limit to scalability [1, 22].

In SPAIN, we use a mechanism called *chirping* (see Sec. 6) which avoids most timeout-related flooding.

Host broadcasts: Ethernet's original shared-bus design made broadcasting easy; not surprisingly, protocols such as the Address Resolution Protocol (ARP) and the Dynamic Host Configuration Protocol (DHCP) were designed to exploit broadcasts. Since broadcasts consume resources throughout a layer-2 domain, broadcasting can limit the scalability of an Ethernet domain [3, 15, 22, 26]. Greenberg *et al.* [16] observe that "...the overhead of broadcast traffic (e.g., ARP) limits the size of an IP subnet to *a few hundred servers*..."

SPAIN does not eliminate broadcasts, but we can exploit certain aspects of both the data-center environment and our willingness to modify end-host implementations. See [25] for more discussion.

3 Related Work

Spanning trees in Ethernet have a long history. The original algorithm was first proposed in 1985 [28], and was adapted as the IEEE 802.1D standard in 1990. Since then it has been improved and adapted along several dimensions. While the Rapid Spanning Tree Protocol (802.1s) reduces convergence time, the Per-VLAN Spanning Tree (802.1Q) improves link utilization by allowing each VLAN to have its own spanning tree. Sharma *et al.* exploit these multiple spanning trees to achieve improved fault recovery [33]. In their work, Viking manager, a central entity, communicates and pro-actively manages both switches and end hosts. Based on its global view of the network, the manager selects (and, if needed, dynamically re-configures) the spanning trees.

Most proposals for improving Ethernet scalability focus on eliminating the restrictions to a single spanning tree.

SmartBridge, proposed by Rodeheffer *et al.* [30], completely eliminated the use of a spanning tree. SmartBridges learn, based on the principles of diffused computation, locations of switches as well as hosts to forward packets along the shortest paths. STAR, a subsequent architecture by Lui *et al.* [24] achieves similar benefits while also facilitating interoperability with the 802.1D standard. Perlman's RBridges, based on an IS-IS routing protocol, allow shortest paths, can inter-operate with existing bridges, and can also be optimized for IP [29]. Currently, this work is being standardized by the TRILL working group of IETF [7]. Note that TRILL focuses only on shortest-path or equal-cost routes, and does not support multiple paths of different lengths.

Myers *et al.* [26] proposed eliminating the basic reason for the spanning tree, the reliance on broadcast as the basic primitive, by combining link-state routing with a directory for host information.

More recently, Kim *et al.* [22] proposed the SEATTLE architecture for very large and dynamic Ethernets. Their switches combine link-state routing with a DHT to achieve broadcast elimination and shortest-path forwarding, without suffering the large space requirements

Table 1: Comparing SPAIN against related work

	SPAIN	SEATTLE [22]	TRILL [7]	VL2 [17]	PortLand [27]	MOOSE [31]
Wiring Topology	Arbitrary	Arbitrary	Arbitrary	Fat-tree	Fat-tree	Arbitrary
Usable paths	Arb. multiple paths	Single Path	ECMP	ECMP	ECMP	Single Path
Deploy incrementally?	YES	NO	YES	NO	NO	YES
Uses COTS switches?	YES (L2)	NO	NO	YES (L3)	NO	NO
Needs end-host mods?	YES	NO	NO	YES	NO	NO

Fat-tree = multi-rooted tree; ECMP = Equal Cost Multi-Path.

of some of the prior approaches; otherwise, SEATTLE is quite similar to TRILL.

Greenberg *et al.* [18] proposed an architecture that scales to 100,000 or more servers. They exploit programmable commodity layer-2 switches, allowing them to modify the data and control planes to support hot-spot-free multipath routing. A sender host for each flow consults a central directory and determines a random intermediary switch; it then bounces the flow via this intermediary. When all switches know of efficient paths to all other switches, going via a random intermediary is expected to achieve good load spreading.

Several researchers have proposed specific regular topologies that support scalability. Fat trees, in particular, have received significant attention. Al-Fares *et al.* [10] advocate combining fat trees with a specific IP addressing assignment, thereby supporting novel switch algorithms that provide high bisection bandwidth without expensive core switches. Mysore *et al.* [27] update this approach in their PortLand design, which uses MAC-address re-writing instead of IP addressing, thus creating a flat L2 network. Scott *et al.* [31] similarly use MAC-address re-writing in MOOSE, but without imposing a specific topology; however, MOOSE uses shortest-path forwarding, rather than multipath.

VL2 [17] provides the illusion of a large L2 network on top of an IP network with a Clos [14] topology, using a logically centralized directory service. VL2 achieves Equal-Cost Multipath (ECMP) forwarding in Clos topologies by assigning a single IP anycast address to all core switches. It is not obvious how one could assign such IP anycast addresses to make multipath forwarding work in non-Clos topologies.

The commercial switch vendor Woven Systems [8] also used a fat tree for the interconnect inside their switch chassis, combing their proprietary vScale chips with Ethernet switch chips that include specific support for fat-trees [4]. The vScale chips use a proprietary algorithm to spread load across the fat-tree paths.

In contrast to fat-tree topologies, others have proposed recursive topologies such as hypercubes. These include DCell [20] and BCube [19].

As summarized in Tab. 1, SPAIN differs from all of this prior work because it provides multipath forwarding, uses unmodified COTS switches, works with arbitrary topologies, supports incremental deployment, and requires no centralized controllers.

4 The design of SPAIN

We start with our specific goals for SPAIN, including the context in which it operates. Our goals are to:

- Deliver more bandwidth and better reliability than spanning tree.
- Support arbitrary topologies, not just fat-tree or hypercube, and extract the best bisection bandwidth from any topology.
- Utilize unmodified, off-the-shelf, commodity-priced (COTS) Ethernet switches.
- Minimize end host software changes, and be incrementally deployable.

In particular, we want to support flat Layer-2 addressing and routing, so as to:

- Simplify network manageability by retaining the plug-and-play properties of Ethernet at larger scales.
- Facilitate non-routable protocols, such as Fibre Channel over Ethernet (FCoE), that are required for "fabric convergence" within data centers [23]. Fabric convergence, the replacement of special-purpose interconnects such as Fibre Channel with standard Ethernet, can reduce hardware costs, management costs, and rack space.
- Improve the flexibility of virtual server and storage placement within data centers, by reducing the chances that arbitrary placement could create bandwidth problems, and by avoiding the complexity of VM migration between IP subnets.

We explicitly limit the focus of SPAIN to data-center networks, rather than trying to solve the general problem of how to scale Ethernet. Also, while we believe that SPAIN will scale to relatively large networks, our goal is not to scale to arbitrary sizes, but to support typical-sized data-centers.

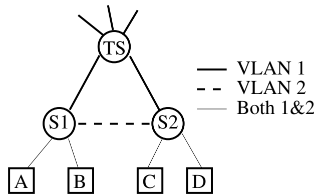


Figure 1: Example of VLANs used for multipathing

4.1 Overview of SPAIN

In SPAIN, we pre-compute a set of paths that utilizes the redundancy in the physical wiring, both to provide high bisection bandwidth and to improve fault tolerance. We then merge these paths into a set of trees, map each tree to a separate VLAN, and install these VLANs on the switches. We usually need only a few VLANs to cover the physical network, since a single VLAN ID can be re-used for multiple disjoint subtrees.

SPAIN allows a pair of end hosts to use different VLANs, potentially traversing different links at different times, for different flows; hence, SPAIN can achieve higher throughput and better fault-tolerance than traditional spanning-tree Ethernet.

SPAIN reserves VLAN 1 to include all nodes. This default VLAN is thus always available as a fallback path, or if we need to broadcast or multicast to all nodes. We believe that we can support multicast more efficiently by mapping multicast trees onto special VLANs, but this is future work.

SPAIN requires only a few switch features: MAC-address learning and VLAN support; these are already present in most COTS switches. Optionally, SPAIN can exploit other switch features to improve performance, scale, and fault tolerance, or to reduce manual configuration: LLDP; SNMP queries to get LLDP information; and the Per-VLAN Spanning Tree Protocol or the Multiple Spanning Tree Protocol (see Sec. 5.5).

SPAIN requires a switch to store multiple table entries (one per VLAN tree) for each destination, in the worst case where flows are active for all possible (VLAN, destination) pairs. (Table overflows lead to packet flooding; they are not fatal.) This could limit SPAIN's applicability to very large networks with densely populated traffic matrices, but even inexpensive merchant-silicon switches have sufficiently large tables for moderately-large networks.

For data centers where MAC addresses are known a priori, we have designed another approach called *FIB-pinning*, but do not describe it here due to space constraints. See [25] for more details.

Fig. 1 illustrates SPAIN with a toy example, which could be a fragment of a larger data-center network. Although there is a link between switches S1 and S2, the standard STP does not forward traffic via that link. SPAIN creates two VLANs, with VLAN1 covering the normal spanning tree, and VLAN2 covering the alternate link. Once the VLANs have been configured, end-host A could (for example) use VLAN1 for flows to C while end-host B uses VLAN2 for flows to D, thereby doubling the available bandwidth versus traditional Ethernet. (SPAIN allows more complex end-host algorithms, to support fault tolerance and load balancing.)

Note that TRILL or SEATTLE, both of are shortest-path (or equal-cost multi-path) protocols, would only use the path corresponding to VLAN2.

SPAIN requires answers to three questions:

1. Given an arbitrary topology of links and switches, with finite switch resources, how should we compute the possible paths to use between host pairs?
2. How can we set up the switches to utilize these paths?
3. How do pairs of end hosts choose which of several possible paths to use?

Thus, SPAIN includes three key components, for *path computation*, *path setup*, and *path selection*. The first two can run offline (although online reconfiguration could help improve network-wide QoS and failure resilience); the path selection process runs online at the end hosts for each flow.

5 Offline configuration of the network

In this section, we describe the centralized algorithms SPAIN uses for offline network configuration: path computation and path setup. (Sec. 6 discusses the online, end-host-based path selection algorithms.)

These algorithms address several challenges:

- **Which set of paths to use?**: The goal is to compute smallest set of paths that exploit all of the redundancy in the network.
- **How to map paths to VLANs?**: We must minimize the number of VLANs used, since Ethernet only allows 4096 VLANs, and some switches support fewer. Also, each VLAN consumes switch resources – a switch needs to cache a learning-table entry for each known MAC on each VLAN.
- **How to handle unplanned topology changes?**: Physical topologies (links and switches) change either due to failures and repairs of links and switches, or due to planned upgrades. Our approach is to recompute and re-install paths only during upgrades, which should be infrequent, and depend on dynamic fault-tolerance techniques to handle unplanned changes.

Because of space constraints, we omit many details of the path computation algorithms; these may be found in the Technical Report version of the paper [25].

5.1 Practical issues

SPAIN's centralized configuration mechanism must address two practical issues: learning the actual topology, and configuring the individual switches with the correct VLANs.

Switches use the Link-Layer Discovery Protocol (LLDP) (IEEE Standard 802.1AB) to advertise their identities and capabilities. They collect the information they receive from their neighbors and store it in their SNMP MIB. We can leverage this support to programmatically determine the topology of the entire L2 network.

Switches maintain a *VLAN-map* table, to track the VLANs allowed on each physical interface, along with information about whether packets will arrive with a VLAN header or not. Each interface can be set in *untagged mode* or *tagged mode* for each VLAN.[1] If a port is in *tagged* mode for a VLAN v, packets received on that interface with VLAN tag v in the Ethernet header are accepted for forwarding. If a port is in *untagged* mode for VLAN v, all packets received on that port without a VLAN tag are assumed to be part of VLAN v. Any packet with VLAN v received on a port not configured for VLAN v are simply dropped. For SPAIN, we assume that this VLAN assignment can be performed programmatically using SNMP.

For each graph computed by the path layout program, SPAIN's switch configuration module instantiates a VLAN corresponding to that graph onto the switches covered by that VLAN. For a graph $G(V, E)$ with VLAN number v, this module contacts the switch corresponding to each vertex in V and sets all ports of that switch whose corresponding edges appear in E in tagged mode for VLAN v. Also, all ports facing end-hosts are set to tagged mode for VLAN v, so that tagged packets from end-hosts are accepted.

5.2 Path-set computation

Our first goal is to compute a *path set*: a set of link-by-link loop-free paths connecting pairs of end hosts through the topology.

A good path set achieves two simultaneous objectives. First, it exploits the available topological redundancy. That is, the path set includes enough paths to ensure that any source-destination pair, at any given time, can find at least one *usable* path between them. By "usable path", we mean a path that does not go through bottlenecked or failed links. Hence a path set that includes *all* possible paths is trivially the best, in terms of exploiting the redundancy. However, such a path set might be impractical, because switch resources (especially on COTS switches) might be insufficient to instantiate so many paths. Thus, the second objective for a good path set is that it has a limited number of paths.

[1] This is the terminology used by HP ProCurve. Cisco uses the terms *access mode* and *trunk mode*.

Algorithm 1 Algorithm for Path Computation

1: Given:
2: $G_{full} = (V_{full}, E_{full})$: The full topology,
3: w: Edge weights,
4: s: Source, d: Destination
5: k: Desired number of paths per s, d pair
6:
7: Initialize: $\forall e \in E : w(e) = 1$
8: /* `shortest` computes weighted shortest path */
9: Path p = `shortest`(G, s, d, w) ;
10: **for** $e \in p$ **do**
11: $w(e) += |E|$
12:
13: **while** $(|P| < k)$ **do**
14: p = `shortest`(G, s, d, w)
15: **if** $p \in P$ **then**
16: /* no more useful paths */
17: *break ;*
18: $P = P \bigcup \{p\}$
19: **for** $e \in p$ **do**
20: $w(e) += |E|$
21:
22: return P

We accomplish this in steps shown in Algorithm 1. (This algorithm has been simplified to assume unit edge capacities; the extension to non-uniform weights is straightforward.)

First, (lines 7–11), we initialize the set of paths for each source-destination pair to include the shortest path. Shortest paths are attractive because in general, they minimize the network resources needed for each packet, and have a higher probability of staying usable after failures. That is, under the simplifying assumption that each link independently fails (either bottlenecks or goes down) with a constant probability f, then a path p of length $|p|$ will be usable with probability $P_u(p) = (1 - (1 - f)^{|p|})$. (We informally refer to this probability, that a path will be usable, as its "usability," and similarly for the probability of a set of paths between a source-destination pair.) Clearly, since the shortest path has the smallest length, it will have the highest P_u.

Then (lines 13–20), we grow the path set to meet the desired degree (k) of path diversity between any pair of hosts. Note that a path set is usable if at least one of the paths is usable. We denote the usability of a path set *ps* as $PS_u(\text{ps})$. This probability depends not only on the lengths of the paths in the set, but also on the degree of shared links between the paths. A best path set of size k has the maximum $PS_u(\cdot)$ of all possible path sets of size k. However, it is computationally infeasible to find the best path set of size k. Hence, we use a greedy algorithm that adds one path at a time, and that prefers the path that

has the minimum number of links in common with paths that are already in the path set.

We prefer adding a link-disjoint path, because a single link failure can not simultaneously take down both the new path and the existing paths. As shown in [25], in most networks with realistic topologies and operating conditions, a link-disjoint path improves the usability of a path set by the largest amount.

As shown in lines 10–11 and 19–20, we implement our preference for link-disjoint paths by incrementing the edge weights of the path we have added to the path set by a large number (number of edges). This ensures that the subsequent shortest-path computation picks a link that is already part of the path set only if it absolutely has to.

5.3 Mapping path sets to VLANs

Given a set of paths with the desired diversity, SPAIN must then map them onto a minimal set of VLANs. (Remember that Ethernet switches support 4096 VLANs, sometimes fewer.)

We need to ensure that the subgraphs formed by the paths of each VLAN are loop-free, so that the switches work correctly in the face of forwarding-table lookup misses. On such a lookup miss for a packet on a VLAN v, a switch will flood the packet to all outgoing interfaces of VLAN v – if the VLAN has a loop, the packet will circulate forever. (We could run the spanning-tree protocol on each VLAN to ensure there are no loops, but then there would be no point in adding links to the SPAIN VLANs that the STP would simply remove from service.)

Problem 1. VLAN Minimization: *Given a set of paths $P = \{p_1, p_2, ..., p_n\}$ in a graph $G = (V, E)$, find an assignment of paths to VLANs, with minimal number of VLANs, such that the subgraph formed by the paths of each VLAN is loop-free.*

We prove in [25] that Problem 1 is NP-hard. Therefore, we employ a greedy VLAN-packing heuristic, Algorithm 2. Given the set of all paths P computed in Algorithm 1, Algorithm 2 processes the paths serially, constructing a set of subgraphs SG that include those paths. For each path p, if p is not covered by any subgraph in the current set SG, the algorithm tries to greedily pack that path p into any one of the subgraphs in the current set (lines 6–12). If the greedy packing step fails for a path, a new graph is created with this path, and is added to SG (lines 13–15).

Running this algorithm just once might not yield a solution near the optimum. Therefore, we use the best solution from N runs, randomizing the order in which paths are chosen for packing, and the order in which the current set of subgraphs SG are examined.

The serial nature of Algorithm 2 does not scale well; its complexity is $O(mkn^3)$, where m is the VLANs, k is the number of paths, and n is the number of switches. We have designed a parallel algorithm, described in [25], based on graph-coloring heuristics, which yields speedup linear in the number of edge switches.

Algorithm 2 Greedy VLAN Packing Heuristic

```
1:  Given: G = (V, E), k
2:  SG = ∅ /* set of loop-free subgraphs*/
3:  for v ∈ V do
4:    for u ∈ V do
5:      P = ComputePaths(G, v, u, k) ;
6:      for p ∈ P /* in a random order */ do
7:        if p not covered by any graph in SG then
8:          Success = FALSE;
9:          for S ∈ SG /* in a random order */ do
10:           if p does not create loop in S then
11:             Add p to S
12:             Success = TRUE ;
13:         if Success == FALSE then
14:           S' = new graph with p
15:           SG = SG ⋃ {S'}
16: return SG
```

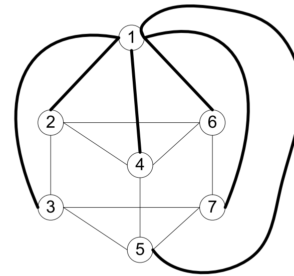


Figure 2: 7-switch topology; original tree in bold

5.3.1 An example

Fig. 2 shows a relatively simple wiring topology with seven switches. One can think of this as a 1-level tree (with switch #1 as the root), augmented by adding three cross-connect links to each non-root switch.

Fig. 3 shows how the heuristic greedy algorithm (Alg. 2) chooses seven VLANs to cover this topology. VLAN #1 is the original tree (and is used as the default spanning tree).

5.4 Algorithm performance

Since our algorithm is an approximate solution to an NP-hard problem, we applied it to a variety of different topologies that have been suggested for data-center networks, to see how many VLANs it requires. Where possible, we also present the optimal number of VLANs. (See [25] for more details about this analysis.)

These topologies include FatTree (p) [10], a 2-ary 3-tree, where p is the number of ports per switch; BCube (p, l) [19], where p is the number of ports per switch, and l is the number of levels in the recursive construction of

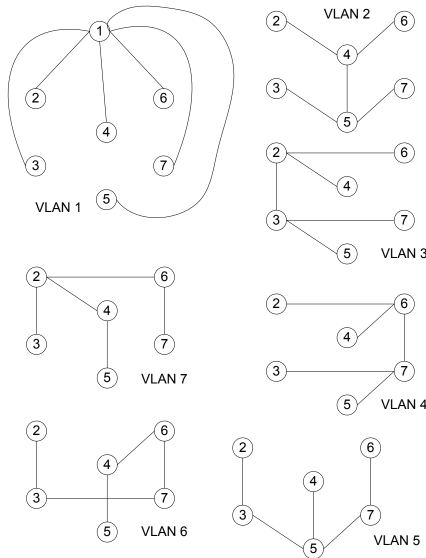


Figure 3: VLANs covering 7-switch topology of Fig. 2

the topology; 2-D HyperX (k) [9], where k is the number of switches in each dimension of a 2-D mesh; and CiscoDC, Cisco's recommended data center network [12], a three-layer tree with two core switches, and with parameters (m, a) where m is the number of aggregation modules, and a the number of access switch pairs associated with each aggregation module.

Table 2: Performance of VLAN mapping heuristic

Topology	**Minimal # of VLANs**	**SPAIN's heuristic**	**Trials (N) for best**
FatTree (p)	$(p/2)^2$	$(p/2)^2$	1 for all p
BCube (p, l)	$p^{l-1}l$	$p^{l-1}l$	290 for (2,3) 6 for (3,2)
2-D HyperX (k)	Unknown $O(k^3)$	12 for k=3 38 for k=4	475 304
CiscoDC (m, a)	Unknown	9 for (2,2) 12 for (3,2) 18 for (4,3)	1549 52 39
our testbed	4	4	4

Table 2 shows the performance of SPAIN's VLAN mapping heuristic on different topologies. The heuristic matches the optimal mapping on FatTree and BCube. We don't yet know the optimal value for CiscoDC or 2-D HyperX, although for 2-D HyperX, k^3 is a loose upper bound. The table also shows that, for the Open Cirrus subset used in our experiments (Sec. 10), the heuristic uses the optimal number (4) of VLANs.

The last column in the table shows the number of trials (N) it took for SPAIN's VLAN packing algorithm to generate its best result; we show the worst case over five runs, and the averages are much smaller. In some cases, luck seems to play a role in how many trials are required. Each row took less than 60 sec., using a single CPU (for these computations, we used the serial algorithm, not the parallel algorithm).

5.5 Fault tolerance in SPAIN

A SPAIN-based network must disable the normal STP behavior on all switches; otherwise, they will block the use of their non-spanning-tree ports, preventing SPAIN from using those links in its VLANs. (SPAIN configures its VLANs to avoid loops, of course.) Disabling STP means that we lose its automatic fault tolerance.

Instead, SPAIN's fault tolerance is based on the pre-provisioning of multiple paths between pairs of hosts, and on end-host detection and recovery from link and switch failures; see Sec. 6.6 for details.

However, SPAIN could use features like Cisco's proprietary Per-VLAN Spanning Tree (PVST) or the IEEE 802.1s standard Multiple Spanning Tree (MST) to improve fault tolerance. SPAIN could configure switches so that, for each VLAN, PVST or MST would prefer the ports in that VLAN over other ports (using per-port spanning tree priorities or weights). This allows a switch to fail over to the secondary ports if PVST or MST detects a failure. SPAIN would still use its end-host failure mechanisms for rapid repair of flows as the spanning tree protocols have higher convergence time, and in case some switches do not support PVST/MST.

6 End-host algorithms

Once the off-line algorithms have computed the paths and configured the switches with the appropriate VLANs, all of the online intelligence in SPAIN lies in the end hosts.[2] SPAIN's end-host algorithms are designed to meet five goals: (1) effectively spread load across the pre-computed paths, (2) minimize the overheads of broadcasting and flooding, (3) efficiently detect and react to failures in the network, (4) facilitate end-point mobility (e.g., VM migration), and (5) enable incremental deployment. We generically refer to the end-host implementation as the "SPAIN driver," although (as described in Sec. 9), some functions run in a user-mode daemon rather than in the kernel-mode driver.

The SPAIN driver has four major functions: boot-time initialization, sending a packet, receiving a packet, and re-initializing a host after it moves to a new edge switch.

An end host uses the following data structures and parameters:

- $ES(m)$: the ID of the edge switch to which MAC address m is currently connected.

[2]SPAIN could support the use of a centralized service to help end-hosts optimize their load balancing, but we have not yet implemented this service, nor is it a necessary feature.

- $V_{reach}(es)$: the set of VLANs that reach the edge switch es.
- R: the *reachability* VLAN map, a bit map encoding the union of $V_{reach}(\bullet)$ over all es, computed by the algorithms in Section 5.
- $V_{usable}(es)$: the set of VLANs that have recently tested as usable to reach es.
- T_{repin} is the length of time after which non-TCP flows go through the VLAN re-pinning process.
- T_{sent} is the minimum amount of time since last send on a VLAN that triggers a *chirp* (see below).
- $V_{sent}(es)$: the set of VLANs that we sent a packet via es within the last T_{sent} seconds.

SPAIN uses a protocol we call *chirping* for several functions: to avoid most timeout-related flooding, to test VLANs for usability, and to support virtual-machine migration. For VM migration, chirping works analogously to the Gratuitous ARP (GARP) mechanism, in which a host broadcasts an ARP request for its own IP → MAC binding.

An end-host A sends a unicast *chirp* packet to another host B if B has just started sending a flow to A, and if A has not sent any packets (including chirps) in the recent past (T_{sent}) to any host connected to the same edge switch as B. An end-host (possibly a virtual machine) broadcasts a chirp packet when it reboots and when it moves to a different switch. A chirp packet carries the triple <IP Address, MAC address, Edge-switch ID>. Chirp packets also carry a $want_reply$ flag to trigger a unicast chirp in response; broadcast chirps never set this flag. All hosts that receive a chirp update their ARP tables with this $IP \rightarrow MAC$ address binding; they also update the $ES(m)$ table. SPAIN sends unicast chirps often enough to preempt most of the flooding that would arise from entries timing out of switch learning tables.

6.1 Host initialization

After a host boots and initializes its NIC drivers, SPAIN must do some initialization. The first step is to download the VLAN reachability map R from a repository. (The repository could be found via a new DHCP option.) While this map could be moderately large (about 5MB for a huge network with 500K hosts and 10K edge switches using all 4K possible VLANs), it is compressible and cachable, and since it changes rarely, a re-download could exploit differential update codings.

Next, the driver determines the ID of the edge switch to which it is connected, by listening for Link Layer Discovery Protocol (LLDP) messages, which switches periodically send on each port. The LLDP rate (typically, once per 30 sec.) is low enough to avoid significant end-host loads, but fast enough that a SPAIN driver that listens for LLDP messages in parallel with other host-booting steps should not suffer much delay.

Finally, the host broadcasts a chirp packet, on the default VLAN (VLAN 1). Although broadcasts are unreliable, a host Y that fails to receive the broadcast chirp from host X will later recover by sending a unicast chirp (with the $wants_response$ flag set) when it needs to select a VLAN for communicating with host X.

Algorithm 3 Selecting a VLAN

```
1:  /* determine the edge switch of the destination */
2:  m = get_dest_mac(flow)
3:  es = get_es(m)
4:  /* candidate VLANs: those that reach es */
5:  if candidate_vlans is empty then
6:      /* No candidate VLANs; */
7:      /* Either es is on a different SPAIN cloud or m is a non-
        SPAIN host */
8:      return the default VLAN (VLAN 1)
9:  /* see if any of the candidates are usable */
10: usable_vlans = candidate_vlans ⋂ V_usable(es)
11: if usable_vlans is empty then
12:     return the default VLAN (VLAN 1)
13: init_probe(candidate_vlans − usable_vlans)
14: return a random v ∈ usable_vlans.
```

6.2 Sending a Packet

SPAIN achieves high bisection bandwidth by spreading traffic across multiple VLANs. The SPAIN driver must choose which VLAN to use for each flow (we normally avoid changing VLANs during a flow, to limit packet reordering). Therefore, the driver must decide which VLAN to use when a flow starts, and must also decide whether to change VLANs (for reasons such as routing around a fault, or improving load-balance for long flows, or to support VM mobility). We divide these into two algorithms: for VLAN selection, and for triggering re-selection of the VLAN for a flow (which we call *re-pinning*).

Algorithm 3 shows the procedure for VLAN selection for a flow to a destination MAC m. The driver uses the $ES(m)$ to find the edge switch es and then uses the reachability map R to find the set of VLANs that reach es. If m does not appear in the ES table, then the driver uses the default VLAN for this flow, and sends a unicast chirp to m to determine if it is a SPAIN host. The driver then computes the *candidate set* by removing VLANs that are not in $V_{usable}(es)$ (which is updated during processing of incoming packets; see Algorithm 5).

If the candidate set is non-empty, the driver selects a member at random and uses this VLAN for the flow.[3] If the set is empty (there are no known-usable VLANs), the flow is instead assigned to VLAN 1. The driver initiates *probing* of a subset of all the VLANs that reach the es but are currently not usable.

SPAIN probes a VLAN v to determine whether it can be used to reach a given destination MAC m (or its ES)

[3] SPAIN with a dynamic centralized controller could bias this choice to improve global load balance; see Sec. 9.

by sending a unicast chirp message to m on v. If the path through VLAN v is usable and if the chirp reaches m, the receiving SPAIN driver responds with its own unicast chirp message on v, which in turn results in v being marked as usable in the probing host (the bit $V_{usable}(es)$ is set to 1).

When to re-pin?: Occasionally, SPAIN must change the VLAN assigned to a flow, or *re-pin* the flow. Re-pinning helps to solve several problems:

1. Fault tolerance: when a VLAN fails (that is, a link or switch on the VLAN fails), SPAIN must rapidly move the flow to a usable VLAN, if one is available.
2. VM migration: if a VM migrates to a new edge switch, SPAIN may have to re-assign the flow to a VLAN that reaches that switch.
3. Improving load balance: in the absence of an online global controller to optimize the assignment of flows to VLANs, it might be useful to shift a long-lived flow between VLANs at intervals, so as to avoid pathological congestion accidents for the entire lifetime of a flow.
4. Better VLAN probing: the re-pinning process causes VLAN probing, which can detect that a "down" VLAN has come back up, allowing SPAIN to exploit the revived VLAN for better load balance and resilience.

When the SPAIN driver detects either of the first two conditions, it immediately initiates re-pinning for the affected flows.

However, re-pinning for the last two reasons should not be done *too* frequently, since this causes problems of its own, especially for TCP flows: packet reordering, and (if re-pinning changes the available bandwidth for a flow) TCP slow-start effects. Hence, the SPAIN driver distinguishes between TCP and non-TCP flows. For non-TCP flows, SPAIN attempts re-pinning at regular intervals.

For TCP flows, re-pinning is done only to address failure or serious performance problems. The SPAIN driver initiates re-pinning for these flows only when the congestion window has become quite small, and the current (outgoing) packet is a retransmission. Together, these two conditions ensure that we do not interfere with TCP's own probing for available bandwidth, and also eliminate the possibility of packet reordering.

Algorithm 4 illustrates the decision process for re-pinning a flow; it is invoked whenever the flow attempts to send a packet.

One risk of re-pinning based on decreases in the congestion window is that it could lead to instability if many flows are sharing a link that suddenly becomes overloaded. SPAIN tries to prevent oscillations by spreading out the re-pinning operations. Also, pinning a flow to a new VLAN does not cause the original VLAN to be marked as unusable, so new flow arrivals could still be assigned to that VLAN, which should damp oscillations. However, we lack solid evidence that these techniques guarantee stability; resolving this issue is future work.

Algorithm 4 Determine if a flow needs VLAN selection

```
1:  if last_move_time >= last_pin_time then
2:      /* we moved since last VLAN selection - re-pin flow */
3:      return true;
4:  current_es = get_es(dst_mac)
5:  if saved_es (from the flow state) != current_es then
6:      /* destination moved – update flow state & re-pin */
7:      saved_es = current_es;
8:      return true
9:  if current_vlan(flow) ≤ 0 then
10:     return true /* new flows need VLAN selection */
11: if proto_of(flow) ≠ TCP then
12:     if (now − last_pin_time) ≥ T_repin then
13:         return true /* periodic re-pin */
14: else
15:     if cwnd(flow) ≤ W_repin_thresh && is_rxmt(flow)
        then
16:         return true /* TCP flow might prefer another path */
17: return false /* no need to repin */
```

Algorithm 5 Receiving a Packet

```
1:  vlan = get_vlan(packet)
2:  m = get_src_mac(packet)
3:  if is_chirp(packet) then
4:      update_ARP_table(packet)
5:      update_ES_table(packet, vlan)
6:      if wants_chirp_response(packet) then
7:          send_unicast_chirp(m, vlan)
8:  es = get_es(m) /* determine sender's edge switch */
9:  /* mark packet-arrival VLAN as usable for es */
10: V_usable(es) = V_usable(es) ⋃ vlan
11: /* chirp if we haven't sent to es via vlan recently */
12: if the vlan bit in V_sent(es) is not set then
13:     send_unicast_chirp(m, vlan)
14:     /* V_sent(es) is cleared every T_sent sec. */
15: deliver packet to protocol stack
```

6.3 Receiving a Packet

Algorithm 5 shows pseudo-code for SPAIN's packet reception processing. All chirp packets are processed to update the host's ARP table and ES table (which maps MAC addresses to edge switches); if the chirp packet requests a response, SPAIN replies with its own unicast chirp on the same VLAN.

The driver treats any incoming packet (including chirps) as proof of the health of the path to its source edge switch es via the arrival VLAN.[4] It records this observation in the $V_{usable}(es)$ bitmap, for use by Algorithm 3.

Finally, before delivering the received packet to the protocol stack, the SPAIN driver sends a unicast chirp

[4] In the case of an asymmetrical failure in which our host's packets are lost, SPAIN will ultimately declare the path dead after our peer gives up on the path and stops using it to send chirps to us.

to the source host if one has not been sent recently. (The pseudo-code omits a few details, including the case where the mapping $ES(m)$ is unknown. The code also omits details of deciding which chirps should request a chirp in response.)

6.4 Table housekeeping

The SPAIN driver must do some housekeeping functions to maintain some of its tables. First, every time a packet is sent, SPAIN sets the corresponding VLAN's bit in $V_{sent}(es)$.

Periodically, the $V_{sent}(es)$ and $V_{usable}(es)$ tables must be cleared, at intervals of T_{sent} seconds. To avoid chirp storms, our driver performs these table-clearing steps in evenly-spaced chunks, rather than clearing the entire table at once.

6.5 Support for end-host mobility

SPAIN makes a host that moves (e.g., for VM migration) responsible for informing all other hosts about its new location. In SPAIN, a VLAN is used to represent a collection of paths. Most failures only affect a subset of those paths. Hence, the usability of a VLAN to reach a given destination is a function of the location of the sender. When the sender moves, it has to re-learn this usability, so it flushes its usability map $V_{usable}(es)$.

Also, peer end-hosts and Ethernet switches must learn where the host is now connected. Therefore, after a host has finished its migration, it broadcasts a chirp, which causes the recipient hosts to update their ARP and ES tables, and which causes Ethernet switches to update their learning tables.

6.6 Handling failures

Failure detection, for a SPAIN end host, consists of detecting a VLAN failure and selecting a new VLAN for the affected flows; we have already described VLAN selection (Algorithm 3).

While we do not have a formal proof, we believe that SPAIN can almost always detect that a VLAN has failed with respect to an edge switch es, because most failures result in observable symptoms, such as a lack of incoming packets (including chirp responses) from es, or from severe losses on TCP flows to hosts on es.

SPAIN's design improves the chances for rapid failure detection because it treats all received packets as probes (to update V_{usable}), and because it aggregates path-health information per edge switch, rather than per destination host. However, because switch or link failures usually do not fully break an entire VLAN, SPAIN does not discard an entire VLAN upon failure detection; it just stops using that VLAN for the affected edge switch(es).

SPAIN also responds rapidly to fault repairs; the receipt of any packet from a host connected to an edge switch will re-establish the relevant VLAN as a valid choice. The SPAIN driver also initiates re-probing of a failed VLAN if a flow that could have used the VLAN is either starting or being re-pinned. At other times, the SPAIN driver re-probes less aggressively, to avoid unnecessary network overhead.

7 How SPAIN meets its goals

We can now summarize how the design of SPAIN addresses the major goals we described in Sec. 4.

Efficiently exploit multiple paths in arbitrary topologies: SPAIN's use of multiple VLANs allows it to spread load over all physical links in the network, not just those on a single spanning tree. SPAIN's use of end-host techniques to spread load over the available VLANs also contributes to this efficiency.

Support COTS Ethernet switches: SPAIN requires only standard features from Ethernet switches. Also, because SPAIN does not require routing all non-local traffic through a single core switch, it avoids the need for expensive switches with high port counts or high aggregate bandwidths.

Tolerate faults: SPAIN pre-computes multiple paths through the network, so that when a path fails, it can immediately switch flows to alternate paths. Also, by avoiding the need for expensive core switches, it decreases the need to replicate expensive components, or to rely on a single component for a large subset of paths.

SPAIN constantly checks path quality (through active probing, monitoring incoming packets, and monitoring the TCP congestion window), thereby allowing it to rapidly detect path failures.

Support incremental deployment: The correctness of SPAIN's end-host processing does not depend on an assumption that all end hosts implement SPAIN. (Our experiments in Sec. 10.4, showing the performance of SPAIN in incremental deployments, did not require any changes to either the SPAIN code or the non-SPAIN hosts.) Traffic to and from non-SPAIN hosts automatically follows the default VLAN, because these hosts never send chirp messages and so the SPAIN hosts never update their $ES(m)$ maps for these hosts.

8 Simulation results

We first evaluate SPAIN using simulations of a variety of network topologies. Later, in Sec. 10, we will show experimental measurements using a specific topology, but simulations are the only feasible way to explore a broader set of network topologies and scales.

We use simulations to (i) show how SPAIN increases link coverage and potential reliability; (ii) quantify the switch-resource requirements for SPAIN's VLAN-based approach; and (iii) show how SPAIN increases the potential aggregate throughput for a network.

We simulated a variety of regular topologies, as de-

Table 3: Summary of simulation results

Topology	#Switches	#Links	#Hosts	Coverage		NCP		#VLANs	Throughput gain	
				STP	SPAIN	STP	SPAIN		$PS = 1$	$PS = \alpha$
FatTree(4)	20	32	16	37.50	100.00	35.00	0.00	4	1.00	2.00
FatTree(8)	80	256	128	15.62	100.00	17.00	0.00	16	1.00	4.00
FatTree(16)	320	2048	1024	7.03	100.00	21.00	0.00	64	1.00	8.00
FatTree(48)	2880	55296	27648	2.17	100.00	17.00	0.00	576	1.00	24.00
HyperX(3)	9	18	216	44.44	100.00	10.83	0.00	12	3.02	1.81
HyperX(4)	16	48	384	31.25	100.00	21.41	0.00	38	4.38	2.49
HyperX(8)	64	448	1536	14.06	100.00	16.31	0.00	290	9.46	5.18
HyperX(16)	256	3840	6144	6.64	100.00	17.86	0.00	971	19.37	10.49
CiscoDC(2,2)	14	31	192	32.26	90.32	12.14	0.71	9	2.20	2.00
CiscoDC(3,2)	20	46	288	34.15	92.68	12.88	0.00	12	2.22	2.00
CiscoDC(4,3)	34	81	576	35.53	94.74	14.64	0.30	18	2.23	2.00
CiscoDC(8,8)	146	361	3072	37.67	97.51	17.91	0.23	38	2.24	2.00
BCube(8,2)	16	128	64	56.25	100.00	23.81	0.14	16	1.44	1.17
BCube(48,2)	96	4608	2304	51.04	100.00	22.50	0.36	96	1.04	1.04
BCube(8,4)	2048	16384	4096	31.82	100.00	43.19	0.00	2048	1.68	1.59

Key: *Coverage*= % of links covered by STP; *NCP*= % of node pairs with no connectivity, for link-failure probability = 0.04; *VLANs*= # VLANs required; *Throughput gain*= aggregate throughput, normalized to STP, for sufficient flows to saturate the network. *PS*= Path-set size; α = Maximum number of edge-disjoint paths between any two switches; $(p/2)^2$ for FatTree(p) topologies, $2k - 2$ for HyperX, 3 for CiscoDC, and l for BCube.

fined in Section 5.4: FatTree (p), BCube (p, l), 2-D HyperX (k), and CiscoDC (m, a).

Table 3 summarizes some of our simulation results for these topologies. We show results where SPAIN's path-set size PS (the number of available paths per source-destination pair) is set to the maximum number α of edge-disjoint paths possible in each topology. $\alpha = (p/2)^2$ for the FatTree topologies, $\alpha = 2k - 2$ for HyperX, $\alpha = 3$ for CiscoDC, and $\alpha = l$ for BCube (l is the number of levels in a BCube(p, l) topology). For throughput experiments, we also present results for $PS = 1$.

The *Coverage* column shows the fraction of links covered by a spanning tree and SPAIN. Except for the CiscoDC topologies, SPAIN always covers 100% of the links. In case of the CiscoDC topologies, our computed edge-disjoint paths do not utilize links between the pairs of aggregation switches, nor the link between the two core switches. Hence, SPAIN's VLANs do not cover these links.

The *NCP* (no-connectivity pairs) column is indicative of the fault tolerance of a SPAIN network; it shows the expected fraction of source-destination pairs that lack connectivity, with a simulated link-failure probability of 0.04, averaged over 10 randomized trials. (These are for PS set to the maximum edge-disjoint paths; even for $PS = 1$, SPAIN would be somewhat more fault-tolerant than STP.)

The *VLANs* column shows the number of VLANs required for each topology. For all topologies considered, the number is below Ethernet's 4K limit.

The *Throughput gain* columns show the aggregate throughput achieved through the network, normalized so that STP = 1. We assume unit per-link capacity and fair sharing of links between flows. We also assume that SPAIN chooses at random from the α available paths (for the $PS = \alpha$ column), and we report the mean of 10 randomized trials.

Our throughput simulation is very simple: it starts by queueing all N flows (e.g., 1 million) spread across H hosts, and then measures the time until all have completed. This models a pessimistic case where all host-to-host paths are fully loaded. Real-world data-center networks never operate like this; the experiments in Sec. 10 reflect a case in which a only subset of host-to-host paths are fully loaded. For example, SPAIN's throughput gain over STP for our BCube(48,2) simulations peaks at about 10x when the number of flows is approximately the number of hosts (this case is not shown in the table). Also, the simulations for SPAIN sometimes favor the $PS = 1$ configuration, which avoids the congestion that is caused by loading too many paths at once (as with $PS = \alpha$ case).

In summary, SPAIN's paths cover more than twice the links, and with significantly more reliability, than spanning-tree's paths, and, for many topologies and workloads, SPAIN significantly improves throughput over spanning tree.

9 Linux end-host implementation

Our end-host implementation for Linux involves two components: a dynamically-loadable kernel-module ("driver") that implements all data-plane functionality, and a user-level controller, mostly composed of shell and Perl scripts.

On boot-up (or whenever SPAIN functionality needs to be initialized) the user-level controller first determines

the MAC address of the network interface. It also determines the ID of the edge-switch to which the NIC is connected, by listening to the LLDP messages sent by the switch. It then contacts a central repository, via a pre-configured IP address; currently, we hard-code the IP address of the repository, but it could be supplied to each host via DHCP options. The controller then downloads the reachability map V_{reach}, and optionally a table that provides bias weights for choosing between VLANs (to support traffic engineering).

The controller then loads the SPAIN kernel driver, creating a `spain` virtual Ethernet device. Next, the controller configures the `spain` virtual device, using the following three major steps.

First, the controller attaches the `spain` device, as a master, to the underlying real `eth` device (as a slave). This master-slave relationship causes all packets that arrive over the `eth` device to be diverted to the `spain` device. That allows SPAIN's chirping protocol to see all incoming packets before they are processed by higher-layer handlers.

Second, the controller configures the `spain` device with the same IP and MAC addresses as the underlying `eth` device. The controller adjusts the routing table so that all the entries that were pointing to the `eth` device are now pointing to `spain`. This re-routing allows SPAIN's chirping protocol to see all outgoing packets.

Third, the controller supplies the driver with the maps it downloaded from the central repository, via a set of special `/proc` files exposed by the driver.

The `spain` driver straightforwardly implements the algorithms described in Section 6, while accounting for certain idiosyncrasies of the underlying NIC hardware. For instance, with NICs that do not support hardware acceleration for VLAN tagging on transmitted packets, the driver must assemble the VLAN header, insert it between the existing Ethernet header fields according to the 802.1Q specification, and then appropriately update the packet meta-data to allow the NIC to correctly compute the CRC field. Similarly, NICs with hardware acceleration for VLAN reception may sometimes deliver a received packet directly to the layer-3 protocol handlers, bypassing the normal driver processing. For these NICs, the `spain` driver must install an explicit packet handler to intercept incoming packets. (Much of this code is borrowed directly from the existing 802.1q module.)

Data structures: The SPAIN driver maintains several tables to support VLAN selection and chirping. To save space, we only discuss a few details. First, we maintain multiple bitmaps (for V_{usable} and V_{sent}) representing several time windows, rather than one bitmap; this spreads out events, such as chirping, to avoid large bursts of activity. Our current implementation ages out the stored history after about 20 seconds, which is fast enough to avoid FIB timeouts in the switches, without adding too much chirping overhead.

We avoid the use of explicit timers (by letting packet events drive the timing, as in "soft timers" [11]), and the use of multiprocessor locks, since inconsistent updates to these bitmaps do not create incorrect behavior.

Overall, the driver maintains about 4KB of state for each known edge switch, which is reasonable even for fairly large networks.

Limitations: Our current implementation can only handle one NIC per server. Data-center servers typically support between two and four NICs, mostly to provide fault tolerance. We should be able to borrow techniques from the existing `bonding` driver to support simultaneous use of multiple NICs. Also, the current implementation does not correctly handle NICs that support TCP offload, since this feature is specifically intended to hide layer-2 packets from the host software.

10 Experimental evaluation

In our experiments, we evaluate four aspects of SPAIN: overheads added by the end-host software; how SPAIN improves over a traditional spanning tree and shortest-path routing; support for incremental deployment; and tolerance of network faults.

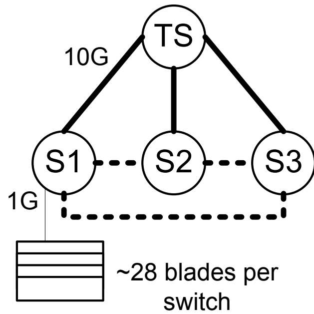
We do not compare SPAIN's performance against other proposed data-center network designs, such as PortLand [27] or VL2 [17], because these require specific network topologies. SPAIN's support for arbitrary topology is an advantage: one can evaluate or use it on the topology one has access to. (We plan to rebuild our testbed network to support fat-tree topologies, but this is hard to do at scale.) However, in Section 10.5, we compare SPAIN's performance against shortest-path routing, as is done in SEATTLE [22], TRILL [7], and IEEE 802.1aq [5].

10.1 Configuration and workloads

We conducted our evaluation on three racks that are part of the (larger) Open Cirrus testbed [13]. All our experiments are run on 80 servers spread across these three racks (rack 1 has 23 servers, rack 2 has 28, and rack 3 has 29). These servers have quad-core 2GHz Intel Xeon CPUs, 8GB RAM and run Ubuntu 9.04.

Each server is connected to a 3500-series HP ProCurve switch using a 1-GigE link, and these rack switches are connected to a central 5406-series ProCurve switch via 10-GigE links.

The Open Cirrus cluster was originally wired using a traditional two-tiered tree, with the core 5406 switch (TS) connected to the 3500 switches (S1, S2, and S3) in each logical rack. To demonstrate SPAIN's benefits, we added 10-GigE cross-connects between the 3500 switches, so that each such switch is connected to another



Dashed lines represent the non-spanning-tree links that we added.

Figure 4: Wiring topology used in our experiments

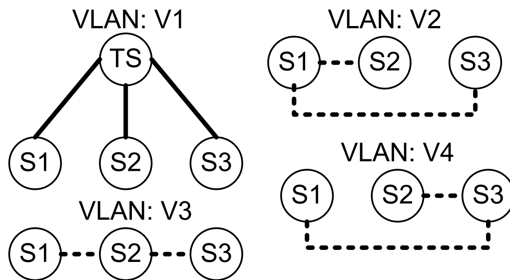


Figure 5: VLANs used by SPAIN for our topology

switch in each physical rack. Fig. 4 shows the resulting wired topology, and Fig. 5 shows the four VLANs computed by the SPAIN offline configuration algorithms.

In our tests, we used a "shuffle" workload (similar to that used in [17]), an all-to-all memory-to-memory bulk transfer among N participating hosts. This communication pattern occurs in several important applications, such as the shuffle phase between Map and Reduce phases of MapReduce, and in join operations in large distributed databases. In our workload, each host transfers 500MB to every other host using 10 simultaneous threads; the order in which hosts choose destinations is randomized to avoid deterministic hot spots. With each machine sending 500MB to all other machines, this experiment transfers about 3.16TB.

10.2 End-host overheads

We measured end-host overheads of several configurations, using ping (100 trials) to measure latency, and NetPerf (10 seconds, 50 trials) to measure both uni-directional and simultaneous bi-directional TCP throughput between a pair of hosts.

We found that we could not get optimal bi-directional TCP throughput, even for unmodified Linux in a two-host configuration, without using "Jumbo" (9000-byte) Ethernet packets. (We were able to get optimal one-way throughput using 1500-byte packets.) We are not entirely sure of the reason for this problem. The TCP experiments described in this paper all use Jumbo packets.

Table 4: End-host overheads

Configuration	ping RTT (usec)	TCP throughput (Mbit/sec) 1-way	TCP throughput (Mbit/sec) 2-way, [min,max]
Unmodified Linux	98	990	1866 [1858,1872]
1st pkt, cold start	4.03 ms		
SPAIN, no chirping	98	988	1860 [1852,1871]
1st pkt, cold start	3.94 ms		
SPAIN w/chirping	99	988	1866 [1857,1876]
1st pkt, cold start	4.84 ms		

ping results: mean of 100 warm-start trials;
throughput: mean, min, max of 50 trials

Table 4 shows the overhead measurements. The SPAIN driver does not appear to measurably affect TCP throughput or "ping" latency. Note that the chirping protocol does not measurably change either throughput or warm-start latency, even though it adds some data-structure updates on every packet transmission and reception. However, it appears to increase cold-start latency slightly, probably because of the CPU costs of allocating and initializing some data structures.

Table 4 shows throughputs for a single TCP flow in each direction. The shuffle workload, described in Sec. 10.1, sometimes leads to an imbalance in the number of TCP flows entering and leaving a node. We discovered that (even in unmodified Linux) this imbalance can lead to a significant throughput drop for the direction with fewer flows. For example, with 9 flows in one direction and 1 flow in the other, the 1-flow direction only gets 274 Mbps, while the 9-flow direction gets 984 Mbps. We are not sure what causes this.

10.3 SPAIN vs. spanning tree

Table 5 shows how SPAIN compares to spanning tree when running the shuffle workload on the Open Cirrus testbed.

Table 5: Spanning-tree vs. SPAIN

	Spanning Tree	SPAIN
Mean goodput/host (Mb/s)	449.25	834.51
Aggregate goodput (Gb/s)	35.60	66.68
Mean completion time/host	744.57 s	397.50 s
Total shuffle time	831.95 s	431.12 s

Results are means of 10 trials, 500 MBytes/trial, 80 hosts

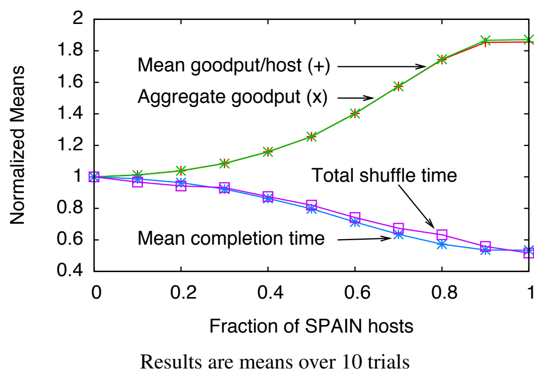
Our SPAIN trials yielded an aggregate goodput of 66.68 Gbps, which is 83.35% of the ideal 80-node goodput of 80 Gbps. This is an improvement of 87.30% over the spanning tree topology for the same experiment.

Based on the two-node bidirectional TCP transfer measurement shown in Table 4, the SPAIN goodput should have been (80*1860/2) Mbps or 74.4 Gbps. Thus the observed goodput is about 10% less than this expected goodput. We suspect that the discrepancy is

the result of the decreased throughput, described in Sec. 10.2, caused by occasional flow-count imbalances during these experiments. Note that we monitored the utilization of all links during the SPAIN experiments, and did not see any saturated links.

10.4 Incremental deployability

One of the key features of SPAIN is incremental deployability. To demonstrate this, we randomly assigned a fraction f of hosts as SPAIN nodes, and disabled SPAIN on the remaining hosts. We measured the goodput achieved with the shuffle experiment. To account for variations in node placement, we ran 10 trials for each fraction f, doing different random node assignments for each trial.



Results are means over 10 trials

Figure 6: Incremental deployability

In Fig. 6, we show how several metrics (per-host goodput, aggregate goodput, mean per-host completion time, and total shuffle run-time) vary as we change the fraction f of nodes on which SPAIN is deployed. The y-values for each curve are normalized to the 0%-SPAIN results. We saw very little trial-to-trial variation (less than 2.5%) in these experiments, with the exception of total shuffle time, which varies up to 13% between trials. This variation does not seem to depend on the use of SPAIN.

As expected, the aggregate goodput increases and the mean completion times decreases as the fraction of SPAIN nodes increases. The curve for the aggregate goodput increases until and flattens at about $f = 0.9$ at which point none of the links in our network are bottlenecked. Hence, at $f = 0.9$, even flows from or to non-SPAIN nodes do not experience any congestion.

10.5 SPAIN vs. Shortest-Path Routing

Protocols such as SEATTLE [22], TRILL [7], and IEEE 802.1aq [5] improve over the spanning-tree protocol by using Shortest-Path Routing (SPR). Although we did not test SPAIN directly against those three protocols, we can compare SPAIN's performance to SPR-based paths by emulation: we restrict the paths employed

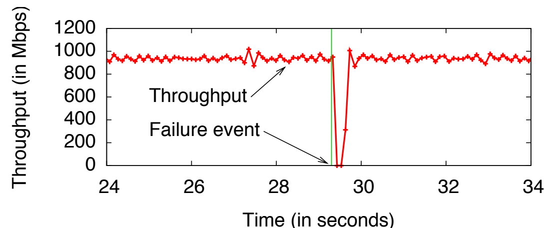


Figure 7: Fault-tolerance experiment

by SPAIN to only those that use the shortest paths between the switches in our test network. In this network, as shown in Fig. 4, the shortest paths between switches S1, S2, and S3 do not go through the core switch (TS); i.e., they do not include the links shown as "VLAN V1" in Fig. 5. (TRILL supports equal-cost multipath, but this would be hard to apply to the topology of Fig. 4.)

We then re-ran the shuffle experiment using the SPR topology, and achieved an aggregate goodput of 62.28 Gbps, vs. SPAIN's 66.61 Gbps goodput. The total shuffle time for SPR is 512.73 s, vs. SPAIN's 430 s. As mentioned in Sec. 10.3, SPAIN's throughput is limited by CPU overheads, so the relatively minor improvement of SPAIN over SPR may be a result of these overheads.

We note that, regardless of the relative performance of SPAIN and SPR, SPAIN retains the advantage of being deployable without any changes to switches. TRILL, SEATTLE, and IEEE 802.1aq Shortest Path Bridging will all require switch upgrades.

10.6 Fault tolerance

We implemented a simple fault detection and repair module that runs at user-level, periodically (100 msec) monitoring the performance of flows. It detects that a VLAN has failed for a destination if the throughput drops by more than 87.5% (equivalent to three halvings of the congestion window), in which case it re-pins the flow to an alternate VLAN.

To demonstrate fault-tolerance in SPAIN, we ran a simple experiment. We used NetPerf to generate a 50-second TCP flow, and measured its throughput every 100 msec. Fig. 7 shows a partial time-line. At 29.3 sec., we removed a link that was in use by this connection. SPAIN detects the failure and repairs the end-to-end path; the TCP throughput returns to normal within 200–300 msec.

11 Summary and conclusions

Our goal for SPAIN was to provide multipath forwarding using inexpensive, COTS Ethernet switches, over arbitrary topologies, and support incremental deployment. We have demonstrated, both in simulations and in experiments, that SPAIN meets those goals. In particular, SPAIN improves aggregate goodput over spanning-tree by 87% on a testbed that would not support most other scalable-Ethernet designs.

We recognize that significant additional work could be required to put SPAIN into practice in a large-scale network. This work includes the design and implementation of a real-time central controller, to support dynamic global re-balancing of link utilizations, and also improvements to SPAIN's end-host mechanisms for assigning flows to VLANs. We also do not fully understand how SPAIN will affect broadcast loads in very large networks.

Acknowledgements

We would like to thank our HP colleagues Peter Haddad, Jean Tourrilhes, Yoshio Turner, Sujata Banerjee, Paul Congdon, Dwight Barron, and Bob Tarjan; the reviewers, and our shepherd, Jennifer Rexford.

References

[1] ARP Flooding Attack. http://www.trendmicro.com/vinfo/secadvisories/default6.asp?VNAME=ARP+Fl%ooding+Attack.

[2] Campus network for high availability: Design guide. Cisco Systems, http://tinyurl.com/d3e6dj.

[3] Enterprise campus 3.0 architecture: Overview and framework. Cisco Systems, http://tinyurl.com/4bwr33.

[4] FocalPoint in Large-Scale Clos Switches. White Paper, Fulcrum Microsystems. http://www.fulcrummicro.com/documents/applications/clos.pdf.

[5] IEEE 802.1aq - Shortest Path Bridging. http://www.ieee802.org/1/pages/802.1aq.html.

[6] IEEE P802.3ba 40Gb/s and 100Gb/s Ethernet Task Force. http://www.ieee802.org/3/ba/public/index.html.

[7] IETF TRILL Working Group. http://www.ietf.org/html.charters/trill-charter.html.

[8] Woven Systems unveils 10G Ethernet switch. Network World, http://tinyurl.com/ajtr4b.

[9] J. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber. HyperX: Topology, Routing, and Packaging of Efficient Large-Scale Networks. In *Proc. Supercomputing*, Nov. 2009.

[10] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proc. SIGCOMM*, pages 63–74, 2008.

[11] M. Aron and P. Druschel. Soft timers: efficient microsecond software timer support for network processing. In *Proc. SOSP*, 1999.

[12] M. Arregoces and M. Portolani. *Data Center Fundamentals*. Cisco Press, 2003.

[13] R. Campbell et al. Open Cirrus Cloud Computing Testbed: Federated Data Centers for Open Source Systems and Services Research. In *Proc. HotCloud*, 2009.

[14] C. Clos. A Study of Non-Blocking Switching Networks. *Bell System Technical Journal*, 32(2):406–424, 1953.

[15] K. Elmeleegy and A. L. Cox. EtherProxy: Scaling Ethernet By Suppressing Broadcast Traffic. In *Proc. INFOCOM*, 2009.

[16] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel. The Cost of a Cloud: Research Problems in Data Center Networks. *SIGCOMM CCR*, 2009.

[17] A. Greenberg, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *Proc. SIGCOMM*, Barcelona, 2009.

[18] A. Greenberg, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Towards a Next Generation Data Center Architecture: Scalability and Commoditization. In *Proc. PRESTO*, pages 57–62, 2008.

[19] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. In *Proc. SIGCOMM*, Barcelona, 2009.

[20] C. Guo, H. Wu, K. Tan, L. Shiy, Y. Zhang, and S. Luz. DCell: A Scalable and Fault-Tolerant Network Structure for Data Centers. In *Proc. SIGCOMM*, Aug. 2008.

[21] John D'Ambrosia. IEEE P802.3ba Task Force Timeline. General Information Session at Interim Meeting http://www.ieee802.org/3/ba/public/jan10/agenda_01_0110.pdf, January 2010.

[22] C. Kim, M. Caesar, and J. Rexford. Floodless in SEATTLE: A Scalable Ethernet Architecture for Large Enterprises. In *Proc. SIGCOMM*, pages 3–14, 2008.

[23] M. Ko, D. Eisenhauer, and R. Recio. A Case for Convergence Enhanced Ethernet: Requirements and Applications. In *Proc. IEEE ICC*, 2008.

[24] K.-S. Lui, W. C. Lee, and K. Nahrstedt. STAR: a transparent spanning tree bridge protocol with alternate routing. *SIGCOMM CCR*, 32(3), 2002.

[25] J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. C. Mogul. SPAIN: Design and Algorithms for Constructing Large Data-Center Ethernets from Commodity Switches. Tech. Rep. HPL-2009-241, HP Labs, 2009.

[26] A. Myers, T. S. E. Ng, and H. Zhang. Rethinking the Service Model: Scaling Ethernet to a Million Nodes. In *Proc. HOTNETS-III*, 2004.

[27] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric. In *Proc. SIGCOMM*, 2009.

[28] R. Perlman. An Algorithm for Distributed Computation of a Spanning Tree in an Extended LAN. *SIGCOMM CCR*, 15(4), 1985.

[29] R. J. Perlman. Rbridges: Transparent Routing. In *Proc. INFOCOM*, 2004.

[30] T. L. Rodeheffer, C. A. Thekkath, and D. C. Anderson. SmartBridge: A Scalable Bridge Architecture. In *Proc. SIGCOMM*, 2000.

[31] M. Scott, A. Moore, and J. Crowcroft. Addressing the Scalability of Ethernet with MOOSE. In *Proc. DC CAVES Workshop*, Sept. 2009.

[32] R. Seifert and J. Edwards. *The All-New Switch Book: The Complete Guide to LAN Switching Technology*. Wiley, 2008.

[33] S. Sharma, K. Gopalan, S. Nanda, and T. Chiueh. Viking: a Multi-spanning-tree Ethernet Architecture for Metropolitan Area and Cluster Networks. In *Proc. INFOCOM*, 2004.

Hedera: Dynamic Flow Scheduling for Data Center Networks

Mohammad Al-Fares* Sivasankar Radhakrishnan*
Barath Raghavan† Nelson Huang* Amin Vahdat*
*{*malfares, sivasankar, nhuang, vahdat*}*@cs.ucsd.edu* †*barath@cs.williams.edu*

**Department of Computer Science and Engineering*
University of California, San Diego

†*Department of Computer Science*
Williams College

Abstract

Today's data centers offer tremendous aggregate bandwidth to clusters of tens of thousands of machines. However, because of limited port densities in even the highest-end switches, data center topologies typically consist of multi-rooted trees with many equal-cost paths between any given pair of hosts. Existing IP multipathing protocols usually rely on per-flow static hashing and can cause substantial bandwidth losses due to long-term collisions.

In this paper, we present Hedera, a scalable, dynamic flow scheduling system that adaptively schedules a multi-stage switching fabric to efficiently utilize aggregate network resources. We describe our implementation using commodity switches and unmodified hosts, and show that for a simulated 8,192 host data center, Hedera delivers bisection bandwidth that is 96% of optimal and up to 113% better than static load-balancing methods.

1 Introduction

At a rate and scale unforeseen just a few years ago, large organizations are building enormous data centers that support tens of thousands of machines; others are moving their computation, storage, and operations to cloud-computing hosting providers. Many applications—from commodity application hosting to scientific computing to web search and MapReduce—require substantial *intra-cluster* bandwidth. As data centers and their applications continue to scale, scaling the capacity of the network fabric for potential all-to-all communication presents a particular challenge.

There are several properties of cloud-based applications that make the problem of data center network design difficult. First, data center workloads are *a priori* unknown to the network designer and will likely be variable over both time and space. As a result, static resource allocation is insufficient. Second, customers wish to run their software on commodity operating systems; therefore, the network must deliver high bandwidth without requiring software or protocol changes. Third, virtualization technology—commonly used by cloud-based hosting providers to efficiently multiplex customers across physical machines—makes it difficult for customers to have guarantees that virtualized instances of applications run on the same physical rack. Without this physical locality, applications face inter-rack network bottlenecks in traditional data center topologies [2].

Applications alone are not to blame. The routing and forwarding protocols used in data centers were designed for very specific deployment settings. Traditionally, in ordinary enterprise/intranet environments, communication patterns are relatively predictable with a modest number of popular communication targets. There are typically only a handful of paths between hosts and secondary paths are used primarily for fault tolerance. In contrast, recent data center designs *rely* on the path multiplicity to achieve horizontal scaling of hosts [3, 16, 17, 19, 18]. For these reasons, data center topologies are very different from typical enterprise networks.

Some data center applications often initiate connections between a diverse range of hosts and require significant aggregate bandwidth. Because of limited port densities in the highest-end commercial switches, data center topologies often take the form of a multi-rooted tree with higher-speed links but decreasing aggregate bandwidth moving up the hierarchy [2]. These multi-rooted trees have many paths between all pairs of hosts. A key challenge is to simultaneously and dynamically forward flows along these paths to minimize/reduce link oversubscription and to deliver acceptable aggregate bandwidth.

Unfortunately, existing network forwarding protocols are optimized to select a single path for each source/destination pair in the absence of failures. Such static single-path forwarding can significantly underutilize multi-rooted trees with any fanout. State of the art forwarding in enterprise and data center environments

uses ECMP [21] (Equal Cost Multipath) to statically stripe flows across available paths using flow hashing. This static mapping of flows to paths does not account for either current network utilization or flow size, with resulting collisions overwhelming switch buffers and degrading overall switch utilization.

This paper presents Hedera, a dynamic flow scheduling system for multi-stage switch topologies found in data centers. Hedera collects flow information from constituent switches, computes non-conflicting paths for flows, and instructs switches to re-route traffic accordingly. Our goal is to maximize aggregate network utilization—bisection bandwidth—and to do so with minimal scheduler overhead or impact on active flows. By taking a global view of routing and traffic demands, we enable the scheduling system to see bottlenecks that switch-local schedulers cannot.

We have completed a full implementation of Hedera on the PortLand testbed [29]. For both our implementation and large-scale simulations, our algorithms deliver performance that is within a few percent of optimal—a hypothetical non-blocking switch—for numerous interesting and realistic communication patterns, and deliver in our testbed up to 4X more bandwidth than state of the art ECMP techniques. Hedera delivers these bandwidth improvements with modest control and computation overhead.

One requirement for our placement algorithms is an accurate view of the demand of individual flows under ideal conditions. Unfortunately, due to constraints at the end host or elsewhere in the network, measuring current TCP flow bandwidth may have no relation to the bandwidth the flow could achieve with appropriate scheduling. Thus, we present an efficient algorithm to estimate idealized bandwidth share that each flow would achieve under max-min fair resource allocation, and describe how this algorithm assists in the design of our scheduling techniques.

2 Background

The recent development of powerful distributed computing frameworks such as MapReduce [8], Hadoop [1] and Dryad [22] as well as web services such as search, e-commerce, and social networking have led to the construction of massive computing clusters composed of commodity-class PCs. Simultaneously, we have witnessed unprecedented growth in the size and complexity of datasets, up to several petabytes, stored on tens of thousands of machines [14].

These cluster applications can often be bottlenecked on the network, not by local resources [4, 7, 9, 14, 16]. Hence, improving application performance may hinge on improving network performance. Most traditional data center network topologies are hierarchical trees with small, cheap edge switches connected to the end-hosts [2]. Such networks are interconnected by two or three layers of switches to overcome limitations in port densities available from commercial switches. With the push to build larger data centers encompassing tens of thousands of machines, recent research advocates the horizontal—rather than vertical—expansion of data center networks [3, 16, 17]; instead of using expensive core routers with higher speeds and port-densities, networks will leverage a larger number of parallel paths between any given source and destination edge switches, so-called *multi-rooted tree* topologies (e.g. Figure 1).

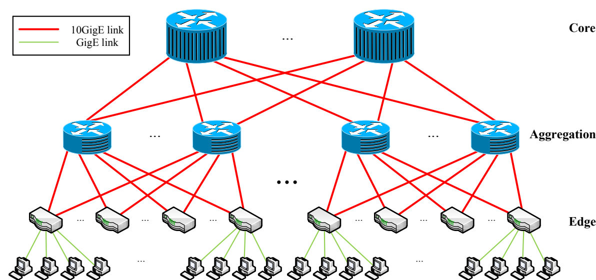


Figure 1: A common multi-rooted hierarchical tree.

Thus we find ourselves at an impasse—with network designs using multi-rooted topologies that have the potential to deliver full bisection bandwidth among all communicating hosts, but without an efficient protocol to forward data within the network or a scheduler to appropriately allocate flows to paths to take advantage of this high degree of parallelism. To resolve these problems we present the architecture of Hedera, a system that exploits path diversity in data center topologies to enable near-ideal bisection bandwidth for a range of traffic patterns.

2.1 Data Center Traffic Patterns

Currently, since no data center traffic traces are publicly available due to privacy and security concerns, we generate patterns along the lines of traffic distributions in published work to emulate typical data center workloads for evaluating our techniques. We also create synthetic communication patterns likely to stress data center networks. Recent data center traffic studies [4, 16, 24] show tremendous variation in the communication matrix over space and time; a typical server exhibits many small, transactional-type RPC flows (e.g. search results), as well as few large transfers (e.g. backups, backend operations such as MapReduce jobs). We believe that the network fabric should be robust to a range of communication patterns and that application developers should not be forced to match their communication patterns to

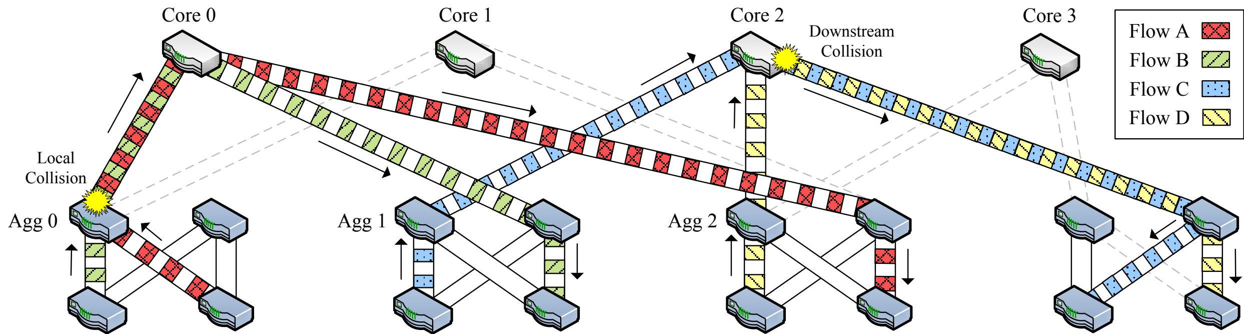


Figure 2: Examples of ECMP collisions resulting in reduced bisection bandwidth. Unused links omitted for clarity.

what may achieve good performance in a particular network setting, both to minimize development and debugging time and to enable easy porting from one network environment to another.

Therefore we focus in this paper on generating traffic patterns that stress and saturate the network, and comparing the performance of Hedera to current hash-based multipath forwarding schemes.

2.2 Current Data Center Multipathing

To take advantage of multiple paths in data center topologies, the current state of the art is to use Equal-Cost Multi-Path forwarding (ECMP) [2]. ECMP-enabled switches are configured with several possible forwarding paths for a given subnet. When a packet with multiple candidate paths arrives, it is forwarded on the one that corresponds to a hash of selected fields of that packet's headers modulo the number of paths [21], splitting load to each subnet across multiple paths. This way, a flow's packets all take the same path, and their arrival order is maintained (TCP's performance is significantly reduced when packet reordering occurs because it interprets that as a sign of packet loss due to network congestion).

A closely-related method is Valiant Load Balancing (VLB) [16, 17, 34], which essentially guarantees equal-spread load-balancing in a mesh network by bouncing individual packets from a source switch in the mesh off of randomly chosen intermediate "core" switches, which finally forward those packets to their destination switch. Recent realizations of VLB [16] perform randomized forwarding on a per-flow rather than on a per-packet basis to preserve packet ordering. Note that per-flow VLB becomes effectively equivalent to ECMP.

A key limitation of ECMP is that two or more large, long-lived flows can collide on their hash and end up on the same output port, creating an avoidable bottleneck as illustrated in Figure 2. Here, we consider a sample communication pattern among a subset of hosts in a multi-rooted, 1 Gbps network topology. We identify two types of collisions caused by hashing. First, TCP flows A and B interfere locally at switch *Agg0* due to a hash collision and are capped by the outgoing link's 1Gbps capacity to *Core0*. Second, with downstream interference, *Agg1* and *Agg2* forward packets independently and cannot foresee the collision at *Core2* for flows C and D.

In this example, all four TCP flows could have reached capacities of 1Gbps with improved forwarding; flow A could have been forwarded to *Core1*, and flow D could have been forwarded to *Core3*. But due to these collisions, all four flows are bottlenecked at a rate of 500Mbps each, a 50% bisection bandwidth loss.

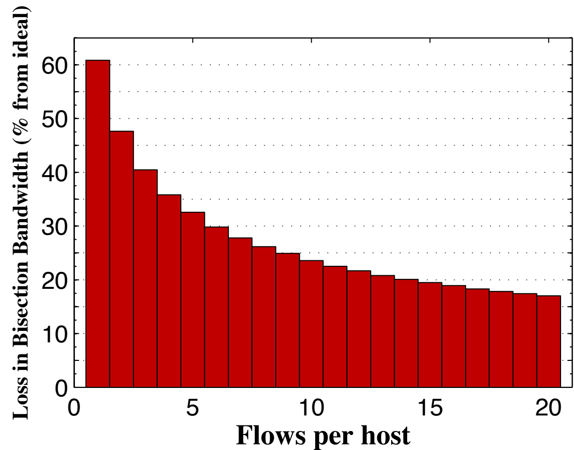


Figure 3: Example of ECMP bisection bandwidth losses vs. number of TCP flows per host for a k=48 fat-tree.

Note that the performance of ECMP and flow-based VLB intrinsically depends on flow size and the number of flows per host. Hash-based forwarding performs well in cases where hosts in the network perform all-to-all communication with one another simultaneously, or with individual flows that last only a few RTTs. Non-uniform communication patterns, especially those involving transfers of large blocks of data, require more careful scheduling of flows to avoid network bottlenecks.

We defer a full evaluation of these trade-offs to Section 6, however we can capture the intuition behind performance reduction of hashing with a simple Monte Carlo simulation. Consider a 3-stage fat-tree composed of 1GigE 48-port switches, with 27k hosts performing a data shuffle. Flows are hashed onto paths and each link is capped at 1GigE. If each host transfers an equal amount of data to all remote hosts one at a time, hash collisions will reduce the network's bisection bandwidth by an average of 60.8% (Figure 3). However, if each host communicates to remote hosts in parallel across 1,000 simultaneous flows, hash collisions will only reduce total bisection bandwidth by 2.5%. The intuition here is that if there are many simultaneous flows from each host, their individual rates will be small and collisions will not be significantly costly: each link has 1,000 slots to fill and performance will only degrade if substantially more than 1,000 flows hash to the same link. Overall, Hedera *complements* ECMP, supplementing default ECMP behavior for communication patterns that cause ECMP problems.

2.3 Dynamic Flow Demand Estimation

Figure 2 illustrates another important requirement for any dynamic network scheduling mechanism. The straightforward approach to find a good network-wide schedule is to measure the utilization of all links in the network and move flows from highly-utilized links to less utilized links. The key question becomes which flows to move. Again, the straightforward approach is to measure the bandwidth consumed by each flow on constrained links and move a flow to an alternate path with sufficient capacity for that flow. Unfortunately, a flow's current bandwidth may not reflect actual demand. We define a TCP flow's *natural* demand to mean the rate it would grow to in a fully non-blocking network, such that eventually it becomes limited by either the sender or receiver NIC speed. For example, in Figure 2, all flows communicate at 500Mbps, though all could communicate at 1Gbps with better forwarding. In Section 4.2, we show how to efficiently estimate the natural demands of flows to better inform Hedera's placement algorithms.

3 Architecture

Described at a high-level, Hedera has a control loop of three basic steps. First, it detects large flows at the edge switches. Next, it estimates the natural demand of large flows and uses placement algorithms to compute good paths for them. And finally, these paths are installed on the switches. We designed Hedera to support any general multi-rooted tree topology, such as the one in Figure 1, and in Section 5 we show our physical implementation using a fat-tree topology.

3.1 Switch Initialization

To take advantage of the path diversity in multi-rooted trees, we must spread outgoing traffic to or from any host as evenly as possible among all the core switches. Therefore, in our system, a packet's path is non-deterministic and chosen on its way up to the core, and is deterministic returning from the core switches to its destination edge switch. Specifically, for multi-rooted topologies, there is exactly one active minimum-cost path from any given core switch to any destination host.

To enforce this determinism on the downward path, we initialize core switches with the prefixes for the IP address ranges of destination pods. A *pod* is any subgrouping down from the core switches (in our fat-tree testbed, it is a complete bipartite graph of aggregation and edge switches, see Figure 8). Similarly, we initialize aggregation switches with prefixes for downward ports of the edge switches in that pod. Finally, edge switches forward packets directly to their connected hosts.

When a new flow starts, the default switch behavior is to forward it based on a hash on the flow's 10-tuple along one of its equal-cost paths (similar to ECMP). This path is used until the flow grows past a threshold rate, at which point Hedera dynamically calculates an appropriate placement for it. Therefore, all flows are assumed to be small until they grow beyond a threshold, 100 Mbps in our implementation (10% of each host's 1GigE link). *Flows* are packet streams with the same 10-tuple of <src MAC, dst MAC, src IP, dst IP, EtherType, IP protocol, TCP src port, dst port, VLAN tag, input port>.

3.2 Scheduler Design

A central scheduler, possibly replicated for fail-over and scalability, manipulates the forwarding tables of the edge and aggregation switches dynamically, based on regular updates of current network-wide communication demands. The scheduler aims to assign flows to non-conflicting paths; more specifically, it tries to not place multiple flows on a link that cannot accommodate their combined natural bandwidth demands.

In this model, whenever a flow persists for some time and its bandwidth demand grows beyond a defined limit, we assign it a path using one of the scheduling algorithms described in Section 4. Depending on this chosen path, the scheduler inserts flow entries into the edge and aggregation switches of the source pod for that flow; these entries redirect the flow on its newly chosen path. The flow entries expire after a timeout once the flow terminates. Note that the state maintained by the scheduler is only soft-state and does not have to be synchronized with any replicas to handle failures. Scheduler state is not re-

$$\begin{bmatrix} & (\frac{1}{3})_1 & (\frac{1}{3})_1 & (\frac{1}{3})_1 \\ (\frac{1}{3})_2 & & (\frac{1}{3})_1 & 0_0 \\ (\frac{1}{2})_1 & 0_0 & & (\frac{1}{2})_1 \\ 0_0 & (\frac{1}{2})_2 & 0_0 & \end{bmatrix} \Rightarrow \begin{bmatrix} & [\frac{1}{3}]_1 & (\frac{1}{3})_1 & (\frac{1}{3})_1 \\ [\frac{1}{3}]_2 & & (\frac{1}{3})_1 & 0_0 \\ [\frac{1}{3}]_1 & 0_0 & & (\frac{1}{2})_1 \\ 0_0 & [\frac{1}{3}]_2 & 0_0 & \end{bmatrix} \Rightarrow \begin{bmatrix} & [\frac{1}{3}]_1 & (\frac{1}{3})_1 & (\frac{1}{3})_1 \\ [\frac{1}{3}]_2 & & (\frac{1}{3})_1 & 0_0 \\ [\frac{1}{3}]_1 & 0_0 & & (\frac{2}{3})_1 \\ 0_0 & [\frac{1}{3}]_2 & 0_0 & \end{bmatrix} \Rightarrow \begin{bmatrix} & [\frac{1}{3}]_1 & (\frac{1}{3})_1 & [\frac{1}{3}]_1 \\ [\frac{1}{3}]_2 & & (\frac{1}{3})_1 & 0_0 \\ [\frac{1}{3}]_1 & 0_0 & & [\frac{2}{3}]_1 \\ 0_0 & [\frac{1}{3}]_2 & 0_0 & \end{bmatrix}$$

Figure 4: An example of estimating demands in a network of 4 hosts. Each matrix element denotes demand per flow as a fraction of the NIC bandwidth. Subscripts denote the number of flows from that source (rows) to destination (columns). Entries in parentheses are yet to converge. Grayed out entries in square brackets have converged.

quired for correctness (connectivity); rather it aids as a performance optimization.

Of course, the choice of the specific scheduling algorithm is open. In this paper, we compare two algorithms, Global First Fit and Simulated Annealing, to ECMP. Both algorithms search for flow-to-core mappings with the objective of increasing the aggregate bisection bandwidth for current communication patterns, supplementing default ECMP forwarding for large flows.

4 Estimation and Scheduling

Finding flow routes in a general network while not exceeding the capacity of any link is called the MULTI-COMMODITY FLOW problem, which is NP-complete for integer flows [11]. And while simultaneous flow routing is solvable in polynomial time for 3-stage Clos networks, no polynomial time algorithm is known for 5-stage Clos networks (i.e. 3-tier fat-trees) [20]. Since we do not aim to optimize Hedera for a specific topology, this paper presents practical heuristics that can be applied to a range of realistic data center topologies.

4.1 Host- vs. Network-Limited Flows

A flow can be classified into two categories: network-limited (e.g. data transfer from RAM) and host-limited (e.g. limited by host disk access, processing, etc.). A network-limited flow will use all bandwidth available to it along its assigned path. Such a flow is limited by congestion in the network, not at the host NIC. A host-limited flow can theoretically achieve a maximum throughput limited by the "slower" of the source and destination hosts. In the case of non-optimal scheduling, a network-limited flow might achieve a bandwidth less than the maximum possible bandwidth available from the underlying topology. In this paper, we focus on network-limited flows, since host-limited flows are a symptom of intra-machine bottlenecks, which are beyond the scope of this paper.

4.2 Demand Estimation

A TCP flow's current sending rate says little about its natural bandwidth demand in an ideal non-blocking network (Section 2.3). Therefore, to make intelligent flow placement decisions, we need to know the flows' max-min fair bandwidth allocation as if they are limited only by the sender or receiver NIC. When network limited, a sender will try to distribute its available bandwidth fairly among all its outgoing flows. TCP's AIMD behavior combined with fair queueing in the network tries to achieve max-min fairness. Note that when there are multiple flows from a host A to another host B, each of the flows will have the same steady state demand. We now describe how to find TCP demands in a hypothetical equilibrium state.

The input to the demand estimator is the set F of source and destination pairs for all active large flows. The estimator maintains an $N \times N$ matrix M; N is the number of hosts. The element in the ith row, j^{th} column contains 3 values: (1) the number of flows from host i to host j, (2) the estimated demand of each of the flows from host i to host j, and (3) a "converged" flag that marks flows whose demands have converged.

The demand estimator performs repeated iterations of increasing the flow capacities from the sources and decreasing exceeded capacity at the receivers until the flow capacities converge; Figure 7 presents the pseudocode. Note that in each iteration of decreasing flow capacities at the receivers, one or more flows converge until eventually all flows converge to the natural demands. The estimation time complexity is $O(|F|)$.

Figure 4 illustrates the process of estimating flow demands with a simple example. Consider 4 hosts (H_0, H_1, H_2 and H_3) connected by a non-blocking topology. Suppose H_0 sends 1 flow each to H_1, H_2 and H_3; H_1 sends 2 flows to H_0 and 1 flow to H_2; H_2 sends 1 flow each to H_0 and H_3; and H_3 sends 2 flows to H_1. The figure shows the iterations of the demand estimator. The matrices indicate the flow demands during successive stages of the algorithm starting with an increase in flow capacity from the sender followed by a decrease in flow capacity at the receiver and so on. The last matrix indicates the final estimated natural demands of the flows.

For real communication patterns, the demand matrix for currently active flows is a sparse matrix since most hosts will be communicating with a small subset of remote hosts at a time. The demand estimator is also

```
GLOBAL-FIRST-FIT(f: flow)
1   if f.assigned then
2       return old path assignment for f
3   foreach p ∈ P_src→dst do
4       if p.used + f.rate < p.capacity then
5           p.used ← p.used + f.rate
6           return p
7   else
8       h = HASH(f)
9       return p = P_src→dst(h)
```

Figure 5: Pseudocode for Global First Fit. GLOBAL-FIRST-FIT is called for each flow in the system.

```
SIMULATED-ANNEALING(n : iteration count)
1   s ← INIT-STATE()
2   e ← E(s)
3   s_B ← s, e_B ← e
4   T_0 ← n
5   for T ← T_0 ... 0 do
6       s_N ← NEIGHBOR(s)
7       e_N ← E(s_N)
8       if e_N < e_B then
9           s_B ← s_N, e_B ← e_N
10      if P(e, e_N, T) > RAND() then
11          s ← s_N, e ← e_N
12  return s_B
```

Figure 6: Pseudocode for Simulated Annealing. s denotes the current state with energy $E(s) = e$. e_B denotes the best energy seen so far in state s_B. T denotes the temperature. e_N is the energy of a neighboring state s_N.

largely parallelizable, facilitating scalability. In fact, our implementation uses both parallelism and sparse matrix data structures to improve the performance and memory footprint of the algorithm.

4.3 Global First Fit

In a multi-rooted tree topology, there are several possible equal-cost paths between any pair of source and destination hosts. When a new large flow is detected, (e.g. 10% of the host's link capacity), the scheduler linearly searches all possible paths to find one whose link components can all accommodate that flow. If such a path is found, then that flow is "placed" on that path: First, a capacity reservation is made for that flow on the links corresponding to the path. Second, the scheduler creates forwarding entries in the corresponding edge and aggregation switches. To do so, the scheduler maintains the reserved capacity on every link in the network and uses that to determine which paths are available to carry new flows. Reservations are cleared when flows expire.

Note that this corresponds to a first fit algorithm; a flow is greedily assigned the *first* path that can accommodate it. When the network is lightly loaded, finding such a path among the many possible paths is likely to be easy; however, as the network load increases and links become saturated, this choice becomes more difficult. Global First Fit does not guarantee that all flows will be accommodated, but this algorithm performs relatively well in practice as shown in Section 6. We show the pseudocode for Global First Fit in Figure 5.

4.4 Simulated Annealing

Next we describe the Simulated Annealing scheduler, which performs a probabilistic search to efficiently compute paths for flows. The key insight of our approach is to assign a single core switch for each destination host rather than a core switch for each flow. This reduces the search space significantly. Simulated Annealing forwards all flows destined to a particular host A through the designated core switch for host A.

The input to the algorithm is the set of all large flows to be placed, and their flow demands as estimated by the demand estimator. Simulated Annealing searches through a solution state space to find a near-optimal solution (Figure 6). A function E defines the energy in the current state. In each iteration, we move to a neighboring state with a certain acceptance probability P, depending on the energies in the current and neighboring states and the current temperature T. The temperature is decreased with each iteration of the Simulated Annealing algorithm and we stop iterating when the temperature is zero. Allowing the solution to move to a higher energy state allows us to avoid local minima.

1. State s: A set of mappings from destination hosts to core switches. Each host in a pod is assigned a particular core switch that it receives traffic from.

2. Energy function E: The total exceeded capacity over all the links in the current state. Every state assigns a unique path to every flow. We use that information to find the links for which the total capacity is exceeded and sum up exceeded demands over these links.

3. Temperature T: The remaining number of iterations before termination.

4. Acceptance probability P for transition from state s to neighbor state s_n, with energies E and E_n.

$$P(E_n, E, T) = \begin{cases} 1 & \text{if } E_n < E \\ e^{c(E-E_n)/T} & \text{if } E_n \geq E \end{cases}$$

where c is a parameter that can be varied. We empirically determined that $c = 0.5 \times T_0$ gives best results for a 16 host cluster and $c = 1000 \times T_0$ is best for larger data centers.

5. Neighbor generator function NEIGHBOR(): Swaps the assigned core switches for a pair of hosts in any of the pods in the current state s.

While simulated annealing is a known technique, our contribution lies in an optimization to significantly reduce the search space and the choice of appropriate energy and neighbor selection functions to ensure rapid convergence to a near optimal schedule. A straightforward approach is to assign a core for each flow individually and perform simulated annealing. However this results in a huge search space limiting the effectiveness of simulated annealing. The diameter of the search space (maximum number of neighbor hops between any two states) with this approach is equal to the number of flows in the system. Our technique of assigning core switches to destination hosts reduces the diameter of the search space to the minimum of the number of flows and the number of hosts in the data center. This heuristic reduces the search space significantly: in a 27k host data center with 27k large flows, the search space size is reduced by a factor of 10^{12000}. Simulated Annealing performs better when the size of the search space and its diameter are reduced [12]. With the straightforward approach, the runtime of the algorithm is proportional to the number of flows and the number of iterations while our technique's runtime depends only on the number of iterations.

We implemented both the baseline and optimized version of Simulated Annealing. Our simulations show that for randomized communication patterns in a 8,192 host data center with 16k flows, our techniques deliver a 20% improvement in bisection bandwidth and a 10X reduction in computation time compared to the baseline. These gains increase both with the size of the data center as well as the number of flows.

Initial state: Each pod has some fixed downlink capacity from the core switches which is useful only for traffic destined to that pod. So an important insight here is that we should distribute the core switches among the hosts in a single pod. For a fat-tree, the number of hosts in a pod is equal to the number of core switches, suggesting a one-to-one mapping. We restrict our solution search space to such assignments, i.e. we assign cores not to individual flows, but to destination hosts. Note that this choice of initial state is only used when the Simulated Annealing scheduler is run for the first time. We use an optimization to handle the dynamics of the system which reduces the importance of this initial state over time.

```
ESTIMATE-DEMANDS()
1   for all i, j
2       M_{i,j} ← 0
3   do
4       foreach h ∈ H do EST-SRC(h)
5       foreach h ∈ H do EST-DST(h)
6   while some M_{i,j}.demand changed
7   return M

EST-SRC(src: host)
1   d_F ← 0
2   n_U ← 0
3   foreach f ∈ ⟨src → dst⟩ do
4       if f.converged then
5           d_F ← d_F + f.demand
6       else
7           n_U ← n_U + 1
8   e_S ← (1.0 − d_F) / n_U
9   foreach f ∈ ⟨src → dst⟩ and not f.converged do
10      M_{f.src,f.dst}.demand ← e_S

EST-DST(dst: host)
1   d_T, d_S, n_R ← 0
2   foreach f ∈ ⟨src → dst⟩
3       f.rl ← true
4       d_T ← d_T + f.demand
5       n_R ← n_R + 1
6   if d_T ≤ 1.0 then
7       return
8   e_S ← 1.0 / n_R
9   do
10      n_R ← 0
11      foreach f ∈ ⟨src → dst⟩ and f.rl do
12          if f.demand < e_S then
13              d_S ← d_S + f.demand
14              f.rl ← false
15          else
16              n_R ← n_R + 1
17      e_S ← (1.0 − d_S) / n_R
18  while some f.rl was set to false
19  foreach f ∈ ⟨src → dst⟩ and f.rl do
20      M_{f.src,f.dst}.demand ← e_S
21      M_{f.src,f.dst}.converged ← true
```

Figure 7: Demand estimator for TCP flows. M is the demand matrix and H is the set of hosts. d_F denotes "converged" demand, n_U is the number of unconverged flows, e_S is the computed equal share rate, and $\langle \text{src} \rightarrow \text{dst} \rangle$ is the set of flows from src to some dst. In EST-DST d_T is the total demand, d_S is sender limited demand, f.rl is a flag for a receiver limited flow and n_R is the number of receiver limited flows.

Neighbor generator: A well-crafted neighbor generator function intrinsically avoids deep local minima. Complying with the idea of restricting the solution search space to mappings with near-uniform mapping of hosts in a pod to core switches, our implementation employs three different neighbor generator functions: (1) swap the assigned core switches for any two randomly chosen hosts in a randomly chosen pod, (2) swap the assigned core switches for any two randomly chosen hosts in a

randomly chosen edge switch, (3) randomly choose an edge or aggregation switch with equal probability and swap the assigned core switches for a random pair of hosts that use the chosen edge or aggregation switch to reach their currently assigned core switches. Our neighbor generator function randomly chooses between the 3 described techniques with equal probability at runtime for each iteration. Using multiple neighbor generator functions helps us avoid deep local minima in the search spaces of individual neighbor generator functions.

Calculation of energy function: The energy function for a neighbor can be calculated incrementally based on the energy in the current state and the cores that were swapped in the neighbor. We need not recalculate exceeded capacities for all links. Swapping assigned cores for a pair of hosts only affects those flows destined to those two hosts. So we need to recalculate the difference in the energy function only for those specific links involved and update the value of the energy based on the energy in the current state. Thus, the time to calculate the energy only depends on the number of large flows destined to the two affected hosts.

Dynamically changing flows: With dynamically changing flow patterns, in every scheduling phase, a few flows would be newly classified as large flows and a few older ones would have completed their transfers. We have implemented an optimization where we set the initial state to the best state from the previous scheduling phase. This allows the route-placement of existing, continuing flows to be disrupted as little as possible if their current paths can still support their bandwidth requirements. Further, the initial state that is used when the Simulated Annealing scheduler first starts up becomes less relevant over time due to this optimization.

Search space: The key characteristic of Simulated Annealing is assigning unique core switches based on destination hosts in a pod, crucial to reducing the size of the search space. However, there are communication patterns where an optimal solution necessarily requires a single destination host to receive incoming traffic through multiple core switches. While we omit the details for brevity, we find that, at least for the fat tree topology, all communication patterns can be handled if: i) the maximum number of large flows to or from a host is at most $k/2$, where k is the number of ports in the network switches, or ii) the minimum threshold of each large flow is set to $2/k$ of the link capacity. Given that in practice data centers are likely to be built from relatively high-radix switches, e.g., $k \geq 32$, our search space optimization is unlikely to eliminate the potential for locating optimal flow assignments in practice.

Algorithm Complexity	Time	Space
Global First-Fit	$O((k/2)^2)$	$O(k^3 + \|F\|)$
Simulated Annealing	$O(f_{avg})$	$O(k^3 + \|F\|)$

Table 1: Time and Space Complexity of Global First Fit and Simulated Annealing. k is the number of switch ports, $|F|$ is the total number of large flows, and f_{avg} is the average number of large flows to a host. The k^3 factor is due to in-memory link-state structures, and the $|F|$ factor is due to the flows' state.

4.5 Comparison of Placement Algorithms

With Global First Fit, a large flow can be re-routed immediately upon detection and is essentially pinned to its reserved links. Whereas Simulated Annealing waits for the next scheduling tick, uses previously computed flow placements to optimize the current placement, and delivers even better network utilization on average due to its probabilistic search.

We chose the Global First Fit and Simulated Annealing algorithms for their simplicity; we take the view that more complex algorithms can hinder the scalability and efficiency of the scheduler while gaining only incremental bandwidth returns. We believe that they strike the right balance of computational complexity and delivered performance gains. Table 1 gives the time and space complexities of both algorithms. Note that the time complexity of Global First Fit is independent of $|F|$, the number of large flows in the network, and that the time complexity of Simulated Annealing is independent of k.

More to the point, the simplicity of our algorithms makes them both well-suited for implementation in hardware, such as in an FPGA, as they consist mainly of simple arithmetic. Such an implementation would substantially reduce the communication overhead of crossing the network stack of a standalone scheduler machine.

Overall, while Simulated Annealing is more conceptually involved, we show in Sec. 6 that it almost always outperforms Global First Fit, and delivers close to the optimal bisection bandwidth both for our testbed and in larger simulations. We believe the additional conceptual complexity of Simulated Annealing is justified by the bandwidth gains and tremendous investment in the network infrastructure of modern data centers.

4.6 Fault Tolerance

Any scheduler must account for switch and link failures in performing flow assignments. While we omit the details for brevity, our Hedera implementation augments the PortLand routing and fault tolerance protocols [29]. Hence, the Hedera scheduler is aware of failures using the standard PortLand mechanisms and can re-route flows mapped to failed components.

5 Implementation

To test our scheduling techniques on a real physical multi-rooted network, we built as an example the fat-tree network described abstractly in prior work [3]. In addition, to understand how our algorithms scale with network size, we implemented a simulator to model the behavior of large networks with many flows under the control of a scheduling algorithm.

5.1 Topology

For the rest of the paper, we adopt the following terminology: for a fat-tree network built from k-port switches, there are k *pods*, each consisting of two layers: lower pod switches (*edge* switches), and the upper pod switches (*aggregation* switches). Each edge switch manages $(k/2)$ hosts. The k pods are interconnected by $(k/2)^2$ *core* switches.

One of the main advantages of this topology is the high degree of available path diversity; between any given source and destination host pair, there are $(k/2)^2$ equal-cost paths, each corresponding to a core switch. Note, however, that these paths are not link-disjoint. To take advantage of this path diversity (to maximize the achievable bisection bandwidth), we must assign flows non-conflicting paths. A key requirement of our work is to perform such scheduling with no modifications to end-host network stacks or operating systems. Our testbed consists of 16 hosts interconnected using a fat-tree of twenty 4-port switches, as shown in Figure 8.

We deploy a parallel control plane connecting all switches to a 48-port non-blocking GigE switch. We emphasize that this control network is not required for the Hedera architecture, but is used in our testbed as a debugging and comparison tool. This network transports only traffic monitoring and management messages to and from the switches; however, these messages could also be transmitted using the data plane. Naturally, for larger networks of thousands of hosts, a control network could be organized as a traditional tree, since control traffic should be only a small fraction of the data traffic. In our deployment, the flow scheduler runs on a separate machine connected to the 48-port switch.

5.2 Hardware Description

The switches in the testbed are 1U dual-core 3.2 GHz Intel Xeon machines, with 3GB RAM, and NetFPGA 4-port GigE PCI card switches [26]. The 16 hosts are 1U quad-core 2.13 GHz Intel Xeon machines with 3GB of RAM. These hosts have two GigE ports, the first connected to the control network for testing and debugging, and the other to its NetFPGA edge switch. The control network is organized as a simple star topology. The central switch is a Quanta LB4G 48-port GigE switch. The scheduler machine has a dual-core 2.4 GHz Intel Pentium CPU and 2GB of RAM.

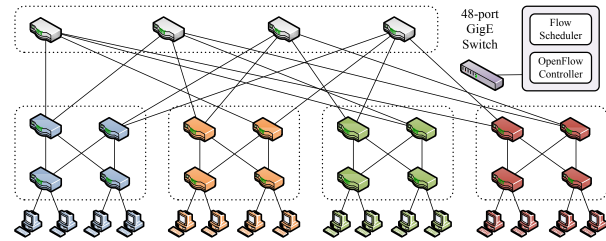


Figure 8: System Architecture. The interconnect shows the data-plane network, with GigE links throughout.

5.3 OpenFlow Control

The switches in the tree all run OpenFlow [27], which allows access to the forwarding tables for all switches. OpenFlow implementations have been ported to a variety of commercial switches, including those from Juniper, HP, and Cisco. OpenFlow switches match incoming packets to flow entries that specify a particular action such as duplication, forwarding on a specific port, dropping, and broadcast. The NetFPGA OpenFlow switches have 2 hardware tables: a 32-entry TCAM (that accepts variable-length prefixes) and a 32K entry SRAM that only accepts flow entries with fully qualified 10-tuples.

When OpenFlow switches start, they attempt to open a secure channel to a central controller. The controller can query, insert, modify flow entries, or perform a host of other actions. The switches maintain statistics per flow and per port, such as total byte counts, and flow durations. The default behavior of the switch is as follows: if an incoming packet does not match any of the flow entries in the TCAM or SRAM table, the switch inserts a new flow entry with the appropriate output port (based on ECMP) which allows any subsequent packets to be directly forwarded at line rate in hardware. Once a flow grows beyond the specified threshold, the Hedera scheduler may modify the flow entry for that flow to redirect it along a newly chosen path.

5.4 Scheduling Frequency

Our scheduler implementation polls the edge switches for flow statistics (to detect large flows), and performs demand estimation and scheduling once every five seconds. This period is due entirely to a register read-rate limitation of the OpenFlow NetFPGA implementation. However, our scalability measurements in Section 6 show that a modestly-provisioned machine can schedule

tens of thousands of flows in a few milliseconds, and that even at the 5 second polling rate, Hedera significantly outperforms the bisection bandwidth of current ECMP methods. In general, we believe that sub-second and potentially sub-100ms scheduling intervals should be possible using straightforward techniques.

5.5 Simulator

Since our physical testbed is restricted to 16 hosts, we also developed a simulator that coarsely models the behavior of a network of TCP flows. The simulator accounts for flow arrivals and departures to show the scalability of our system for larger networks with dynamic communication patterns. We examine our different scheduling algorithms using the flow simulator for networks with as many as 8,192 hosts. Existing packet-level simulators, such as *ns-2*, are not suitable for this purpose: e.g. a simulation with 8,192 hosts each sending at 1Gbps would have to process 2.5×10^{11} packets for a 60 second run. If a per-packet simulator were used to model the transmission of 1 million packets per second using TCP, it would take 71 hours to simulate just that one test case.

Our simulator models the data center topology as a network graph with directed edges. Each edge has a fixed capacity. The simulator accepts as input a communication pattern among hosts and uses it, along with a specification of average flow sizes and arrival rates, to generate simulated traffic. The simulator generates new flows with an exponentially distributed length, with start times based on a Poisson arrival process with a given mean. Destinations are based upon the suite in Section 6.

The simulation proceeds in discrete time ticks. At each tick, the simulation updates the rates of all flows in the network, generates new flows if needed. Periodically it also calls the scheduler to assign (new) routes to flows. When calling the Simulated Annealing and Global First Fit schedulers, the simulator first calls the demand estimator and passes along its results.

When updating flow rates, the simulator models TCP slow start and AIMD, but without performing per-packet computations. Each tick, the simulator shuffles the order of flows and computes the expected rate increase for each flow, constrained by available bandwidth on the flow's path. If a flow is in slow start, its rate is doubled. If it is in congestion avoidance, its rate is additively increased (using an additive increase factor of 15 MB/s to simulate a network with an RTT of 100 μs). If the flow's path is saturated, the flow's rate is halved and bandwidth is freed along the path. Each tick, we also compute the number of bytes sent by the flow and purge flows that have completed sending all their bytes.

Since our simulator does not model individual packets, it does not capture the variations in performance of different packet sizes. Another consequence of this decision is that our simulation cannot capture inter-flow dynamics or buffer behavior. As a result, it is likely that TCP Reno/New Reno would perform somewhat *worse* than predicted by our simulator. In addition, we model TCP flows as unidirectional although real TCP flows involve ACKs in the reverse direction; however, for 1500 byte Ethernet frames and delayed ACKs, the bandwidth consumed by ACKs is about 2%. We feel these trade-offs are necessary to study networks of the scale described in this paper.

We ran each simulation for the equivalent of 60 seconds and measured the average bisection bandwidth during the middle 40 seconds. Since the simulator does not capture inter-flow dynamics and traffic burstiness our results are optimistic (simulator bandwidth exceeds testbed measurements) for ECMP based flow placement because resulting hash collisions would sometimes cause an entire window of data to be lost, resulting in a coarse-grained timeout on the testbed (see Section 6). For the control network we observed that the performance in the simulator more closely matched the performance on the testbed. Similarly, for Global First Fit and Simulated Annealing, which try to optimize for minimum contention, we observed that the performance from the simulator and testbed matched very well. Across all the results, the simulator indicated better performance than the testbed when there is contention between flows.

6 Evaluation

This section describes our evaluation of Hedera using our testbed and simulator. The goal of these tests is to determine the aggregate achieved bisection bandwidth with various traffic patterns.

6.1 Benchmark Communication Suite

In the absence of commercial data center network traces, for both the testbed and the simulator evaluation, we first create a group of communication patterns similar to [3] according to the following styles:

(1) Stride(i): A host with index x sends to the host with index $(x + i) mod(num_hosts)$.

(2) Staggered Prob *(EdgeP, PodP)*: A host sends to another host in the same edge switch with probability *EdgeP*, and to its same pod with probability *PodP*, and to the rest of the network with probability *1-EdgeP - PodP*.

(3) Random: A host sends to any other host in the network with uniform probability. We include bijective mappings and ones where hotspots are present.

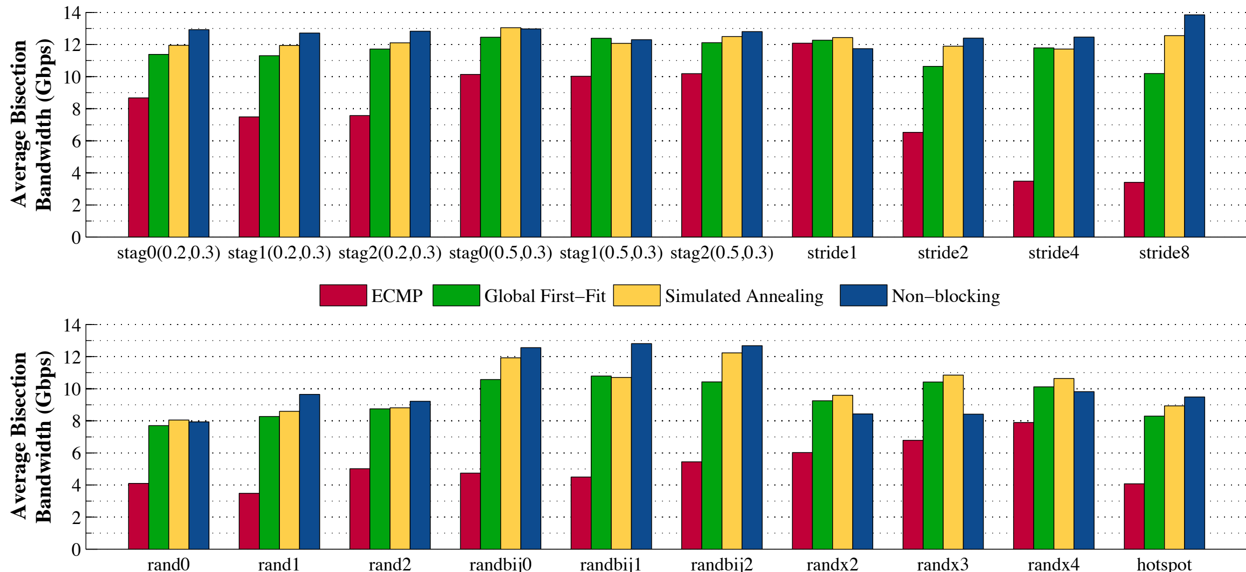


Figure 9: Physical testbed benchmark suite results for the three routing methods vs. a non-blocking switch. Figures indicate network bisection bandwidth achieved for staggered, stride, and randomized communication patterns.

We consider these mappings for networks of different sizes: 16 hosts, 1,024 hosts, and 8,192 hosts, corresponding to $k = \{4, 16, 32\}$.

6.2 Testbed Benchmark Results

We ran benchmark tests as follows: 16 hosts open socket sinks for incoming traffic and measure the incoming bandwidth constantly. The hosts in succession then start their flows according to the sizes and destinations as described above. Each experiment lasts for 60 seconds and uses TCP flows; we observed the average bisection bandwidth for the middle 40 seconds.

We compare the performance of the scheduler on the fat-tree network to that of the same experiments on the control network. The control network connects all 16 hosts using a non-blocking 48-port gigabit Ethernet switch and represents an ideal network. In addition, we include a static hash-based ECMP scheme, where the forwarding path is determined by a hash of the destination host IP address.

Figure 9 shows the bisection bandwidth for a variety of randomized, staggered, stride and hotspot communication patterns; our experiments saturate the links using TCP. In virtually all the communication patterns explored, Global First Fit and Simulated Annealing significantly outperform static hashing (ECMP), and achieve near the optimal bisection bandwidth of the network (15.4Gb/s goodput). Naturally, the performance of these schemes improves as the level of communication locality increases, as demonstrated by the staggered probability figures. Note that for stride patterns (common to HPC computation applications), the heuristics consistently compute the correct flow-to-core mappings to efficiently utilize the fat-tree network, whereas the performance of static hash quickly deteriorates as the stride length increases. Furthermore, for certain patterns, these heuristics also marginally outperform the commercial 48-port switch used for our control network. We suspect this is due to different buffers/algorithms of the NetFPGAs vs. the Quanta switch.

Upon closer examination of the performance using packet captures from the testbed, we found that when there was contention between flows, an entire TCP window of packets was often lost. So the TCP connection was idle until the retransmission timer fired (RTO_{min} = 200ms). ECMP hash based flow placement experienced over 5 times the number of retransmission timeouts as the other schemes. This explains the overoptimistic performance of ECMP in the simulator as explained in Section 5 since our simulator does not model retransmission timeouts and individual packet losses.

6.3 Data Shuffle

We also performed an all-to-all in-memory data shuffle in our testbed. A data shuffle is an expensive but necessary operation for many MapReduce/Hadoop operations in which every host transfers a large amount of data to every other host participating in the shuffle. In this experiment, each host sequentially transfers 500MB to every other host using TCP (a 120GB shuffle).

	ECMP	GFF	SA	Control
Shuffle time (s)	438.44	335.50	335.96	306.37
Host completion (s)	358.14	258.70	261.96	226.56
Bisec. BW (Gbps)	2.811	3.891	3.843	4.443
Goodput (MB/s)	20.94	28.99	28.63	33.10

Table 2: A 120GB shuffle for the placement heuristics in our testbed. Shown is total shuffle time, average host-completion time, average bisection bandwidth and average host goodput.

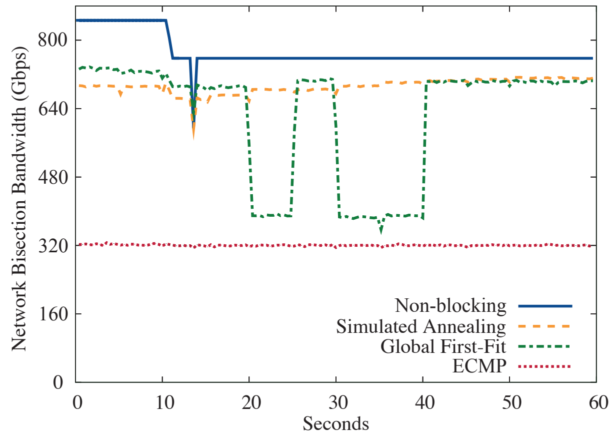


Figure 11: Network bisection bandwidth vs. time for a 1,024 host fat-tree and a random bijective traffic pattern.

The shuffle results in Table 2 show that centralized flow scheduling performs considerably better (39% better bisection bandwidth) than static ECMP hash-based routing. Comparing this to the data shuffle performed in VL2 [16], which involved all hosts making *simultaneous* transfers to all other hosts (versus the sequential transfers in our work), we see that static hashing performs better when the number of flows is significantly larger than the number of paths; intuitively a hash collision is less likely to introduce significant degradation when any imbalance is averaged over a large number of flows. For this reason, in addition to the delay of the Hedera observation/route-computation control loop, we believe that traffic workloads characterized by many small, short RPC-like flows would have limited benefit from dynamic scheduling, and Hedera's default ECMP forwarding performs load-balancing efficiently in this case. Hence, by thresholding our scheduler to only operate on larger flows, Hedera performs well for both types of communication patterns.

6.4 Simulation Results

6.4.1 Communication Patterns

In Figure 10 we show the aggregate bisection bandwidth achieved when running the benchmark suite for a simulated fat-tree network with 8,192 hosts (when k=32).

	Number of Hosts		
SA Iterations	16	1,024	8,192
1000	78.73	74.69	72.83
50000	78.93	75.79	74.27
100000	78.62	75.77	75.00
500000	79.35	75.87	74.94
1000000	79.04	75.78	75.03
1500000	78.71	75.82	75.13
2000000	78.17	75.87	75.05
Non-blocking	81.24	78.34	77.63

Table 3: Percentage of final bisection bandwidth by varying the Simulated Annealing iterations, for a case of random destinations, normalized to the full network bisection. Also shown is the same load running on a non-blocking topology.

We compare our algorithms against a hypothetical non-blocking switch for the entire data center and against static ECMP hashing. The performance of ECMP worsens as the probability of local communication decreases. This is because even for a completely fair and perfectly uniform hash function, collisions in path assignments *do* happen, either within the same switch or with flows at a downstream switch, wasting a portion of the available bandwidth. A global scheduler makes discrete flow placements that are chosen by design to reduce overlap. In most of these different communication patterns, our dynamic placement algorithms significantly outperform static ECMP hashing. Figure 11 shows the variation over time of the bisection bandwidth for the 1,024 host fat-tree network. Global First Fit and Simulated Annealing perform fairly close to optimal for most of the experiment.

6.4.2 Quality of Simulated Annealing

To explore the parameter space of Simulated Annealing, we show in Table 3 the effect of varying the number of iterations at each scheduling period for a randomized, non-bijective communication pattern. This table confirms our initial intuition regarding the assignment quality vs. the number of iterations, as most of the improvement takes place in the first few iterations. We observed that the performance of Simulated Annealing asymptotically approaches the best result found by Simulated Annealing after the first few iterations.

The table also shows the percentage of final bisection bandwidth for a random communication pattern as number of hosts and flows increases. This supports our belief that Simulated Annealing can be run with relatively few iterations in each scheduling period and still achieve comparable performance over time. This is aided by remembering core assignments across periods, and by the arrival of only a few new large flows each interval.

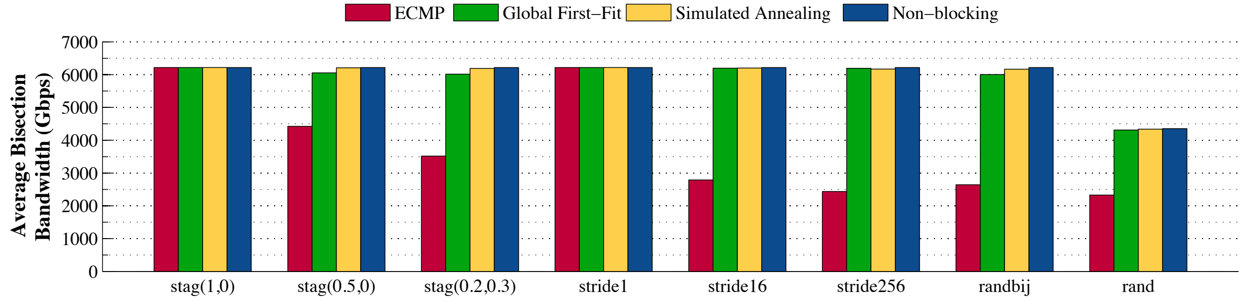


Figure 10: Comparison of scheduling algorithms for different traffic patterns on a fat-tree topology of 8,192-hosts.

k	Hosts	Large flows	Runtime (ms)
16	1024	1024	1.45
16	1024	5000	4.14
32	8192	8192	2.71
32	8192	25000	9.23
32	8192	50000	26.31
48	27648	27648	6.91
48	27648	100000	51.30
48	27648	250000	199.43

Table 4: Demand estimation runtime.

	1,024 hosts		8,192 hosts	
Iterations	f =3,215	f =6,250	f =25k	f =50k
1000	2.997	5.042	6.898	11.573
5000	12.209	20.848	19.091	32.079
10000	23.447	40.255	32.912	55.741

Table 5: Runtime (ms) vs. number of Simulated Annealing iterations for different number of flows f.

6.4.3 Complexity of Demand Estimation

Since the demand estimation is performed once per scheduling period, its runtime must be reasonably small so that the length of the control loop is as small as possible. We studied the runtime of demand estimation for different traffic matrices in data centers of varying sizes.

Table 4 shows the runtimes of the demand estimator for different input sizes. The reported runtimes are for runs of the demand estimator using 4 parallel threads of execution on a modest quad-core 2.13 GHz machine. Even for a large data center with 27,648 hosts and 250,000 large flows (average of nearly 10 large flows per host), the runtime of the demand estimation algorithm is only 200 ms. For more common scenarios, the runtime is approximately 50-100ms in our setup. We expect the scheduler machine to be a fairly high performance machine with more cores, thereby still keeping the runtime well under 100ms even for extreme scenarios.

The memory requirement for the demand estimator in our implementation using a sparse matrix representation is less than 20 MB even for the extreme scenario with nearly 250,000 large flows in a data center with 27k hosts. In more common scenarios, with a reasonable number of large flows in the data center, the entire data structure would fit in the L2 cache of a modern CPU.

Considering the simplicity and number of operations involved, an FPGA implementation can store the sparse matrix in an off-chip SRAM. An FPGA such as the Xilinx Virtex-5 can implement up to 200 parallel processing cores to process this matrix. We estimate that such a configuration would have a computational latency of approximately 5 ms to perform demand estimation even for the case of 250,000 large flows.

6.4.4 Complexity of Simulated Annealing

In Table 5 we show the runtime of Simulated Annealing for different experimental scenarios. The runtime of Simulated Annealing is asymptotically independent of the number of hosts and only dependent on the number of flows. The main takeaway here is the scalability of our Simulated Annealing implementation and its potential for practical application; for networks of thousands of hosts and a reasonable number of flows per host, the Simulated Annealing runtime is on the order of tens of milliseconds, even for 10,000 iterations.

6.4.5 Control Overhead

To evaluate the total control overhead of the centralized scheduling design, we analyzed the overall communication and computation requirements for scheduling. The control loop includes 3 components—all switches in the network send the details of large flows to the scheduler, the scheduler estimates demands of the flows and computes their routes, and the scheduler transmits the new placement of flows to the switches.

We made some assumptions to analyze the length of the control loop. (1) The control plane is made up of 48-port GigE switches with an average 10 μs latency per

	Flows per host		
Hosts	1	5	10
1,024	100.2	100.9	101.7
8,192	101.4	106.8	113.5
27,648	104.6	122.8	145.5

Table 6: Length of control loop (ms).

switch. (2) The format of messages between the switches and the controller are based on the OpenFlow protocol (72B per flow entry) [27]. (3) The total computation time for demand estimation and scheduling of the flows is conservatively assumed to be 100 ms. (4) The last hop link to the scheduler is assumed to be a 10 GigE link. This higher speed last hop link allows a large number of switches to communicate with the scheduler simultaneously. We assumed that the 10 GigE link to the controller can be fully utilized for transfer of scheduling updates.

Table 6 shows the length of the control loop for varying number of large flows per host. The values indicate that the length of the control loop is dominated by the computation time, estimated at 100 ms. These results show the scalability of the centralized scheduling approach for large data centers.

7 Related Work

There has been a recent flood of new research proposals for data center networks; however, none satisfyingly addresses the issue of the network's bisection bandwidth. VL2 [16] and Monsoon [17] propose using per-flow Valiant Load Balancing, which can cause bandwidth losses due to long-term collisions as demonstrated in this work. SEATTLE [25] proposes a single Layer 2 domain with a one-hop switch DHT for MAC address resolution, but does not address multipathing. DCell [19] and BCube [18] suggest using recursively-defined topologies for data center networks, which involves multi-NIC servers and can lead to oversubscribed links with deeper levels. Once again, multipathing is not explicitly addressed.

Researchers have also explored scheduling flows in a multi-path environment from a wide-area context. TeXCP [23] and MATE [10] perform dynamic traffic engineering across multiple paths in the wide-area by using explicit congestion notification packets, which require as yet unavailable switch support. They employ *distributed* traffic engineering, whereas we leverage the data center environment using a tightly-coupled central scheduler. FLARE [31] proposes multipath forwarding in the wide-area on the granularity of *flowlets* (TCP packet bursts); however, it is unclear whether the low intra-data center latencies meet the timing requirements of flowlet bursts to prevent packet reordering and still achieve good performance. Miura *et al.* exploit fat-tree networks by multipathing using tagged-VLANs and commodity PCs [28]. Centralized router control to enforce routing or access control policy has been proposed before by the 4D architecture [15], and projects like Tesseract [35], Ethane [6], and RCP [5], similar in spirit to Hedera's approach to centralized flow scheduling.

Much work has focused on virtual switching fabrics and on individual Clos networks in the abstract, but do not address building an operational multi-level switch architecture using existing commodity components. Turner proposed an optimal non-blocking virtual circuit switch [33], and Smiljanic improved Turner's load balancer and focused on the guarantees the algorithm could provide in a generalized Clos packet-switched network [32]. Oki *et al.* design improved algorithms for scheduling in individual 3-stage Clos switches [30], and Holmburg provides models for simultaneous and incremental scheduling of multi-stage Clos networks [20]. Geoffray and Hoefler describe a number of strategies to increase bisection bandwidth in multistage interconnection networks, specifically focusing on source-routed, per-packet dispersive approaches that break the ordering requirement of TCP/IP over Ethernet [13].

8 Conclusions

The most important finding of our work is that in the pursuit of efficient use of available network resources, a central scheduler with global knowledge of active flows can significantly outperform the state-of-the-art hash-based ECMP load-balancing. We limit the overhead of our approach by focusing our scheduling decisions on the large flows likely responsible for much of the bytes sent across the network. We find that Hedera's performance gains are dependent on the rates and durations of the flows in the network; the benefits are more evident when the network is stressed with many large data transfers both within pods and across the diameter of the network.

In this paper, we have demonstrated the feasibility of building a working prototype of our scheduling system for multi-rooted trees, and have shown that Simulated Annealing almost always outperforms Global First Fit and is capable of delivering near-optimal bisection bandwidth for a wide range of communication patterns both in our physical testbed and in simulated data center networks consisting of thousands of nodes. Given the low computational and latency overheads of our flow placement algorithms, the large investment in network infrastructure associated with data centers (many millions of dollars), and the incremental cost of Hedera's deployment (e.g., one or two servers), we show that dynamic flow scheduling has the potential to deliver substantial bandwidth gains with moderate additional cost.

Acknowledgments

We are indebted to Nathan Farrington and the DCSwitch team at UCSD for their invaluable help and support of this project. We also thank our shepherd Ant Rowstron and the anonymous reviewers for their helpful feedback on earlier drafts of this paper.

References

[1] Apache Hadoop Project. http://hadoop.apache.org/.

[2] Cisco Data Center Infrastructure 2.5 Design Guide. http://www.cisco.com/univercd/cc/td/doc/solution/dcidg21.pdf.

[3] AL-FARES, M., LOUKISSAS, A., AND VAHDAT, A. A Scalable, Commodity Data Center Network Architecture. In *Proceedings of ACM SIGCOMM, 2008*.

[4] BENSON, T., ANAND, A., AKELLA, A., AND ZHANG, M. Understanding Data Center Traffic Characteristics. In *Proceedings of ACM WREN, 2009*.

[5] CAESAR, M., CALDWELL, D., FEAMSTER, N., REXFORD, J., SHAIKH, A., AND VAN DER MERWE, J. Design and Implementation of a Routing Control Platform. In *Proceedings of NSDI, 2005*.

[6] CASADO, M., FREEDMAN, M. J., PETTIT, J., LUO, J., MCKEOWN, N., AND SHENKER, S. Ethane: Taking Control of the Enterprise. In *Proceedings of ACM SIGCOMM, 2007*.

[7] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of OSDI, 2006*.

[8] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of OSDI, 2004*.

[9] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of SOSP, 2007*.

[10] ELWALID, A., JIN, C., LOW, S., AND WIDJAJA, I. MATE: MPLS Adaptive Traffic Engineering. *Proceedings of INFOCOM, 2001*.

[11] EVEN, S., ITAI, A., AND SHAMIR, A. On the Complexity of Timetable and Multicommodity Flow Problems. *SIAM Journal on Computing 5*, 4 (1976), 691–703.

[12] FLEISCHER, M. Simulated Annealing: Past, Present, and Future. In *Proceedings of IEEE WSC, 1995*.

[13] GEOFFRAY, P., AND HOEFLER, T. Adaptive Routing Strategies for Modern High Performance Networks. In *Proceedings of IEEE HOTI, 2008*.

[14] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google File System. In *Proceedings of SOSP, 2003*.

[15] GREENBERG, A., HJALMTYSSON, G., MALTZ, D. A., MYERS, A., REXFORD, J., XIE, G., YAN, H., ZHAN, J., AND ZHANG, H. A Clean Slate 4D Approach to Network Control and Management. *ACM SIGCOMM CCR, 2005*.

[16] GREENBERG, A., JAIN, N., KANDULA, S., KIM, C., LAHIRI, P., MALTZ, D., PATEL, P., AND SENGUPTA, S. VL2: A Scalable and Flexible Data Center Network. In *Proceedings of ACM SIGCOMM, 2009*.

[17] GREENBERG, A., LAHIRI, P., MALTZ, D. A., PATEL, P., AND SENGUPTA, S. Towards a Next Generation Data Center Architecture: Scalability and Commoditization. In *Proceedings of ACM PRESTO, 2008*.

[18] GUO, C., LU, G., LI, D., WU, H., ZHANG, X., SHI, Y., TIAN, C., ZHANG, Y., AND LU, S. BCube: A High Performance, Server-Centric Network Architecture for Modular Data Centers. In *Proceedings of ACM SIGCOMM, 2009*.

[19] GUO, C., WU, H., TAN, K., SHI, L., ZHANG, Y., AND LU, S. DCell: A Scalable and Fault-Tolerant Network Structure for Data Centers. In *Proceedings of ACM SIGCOMM, 2008*.

[20] HOLMBERG, K. Optimization Models for Routing in Switching Networks of Clos Type with Many Stages. *Advanced Modeling and Optimization 10*, 1 (2008).

[21] HOPPS, C. Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992, IETF, 2000.

[22] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proceedings of ACM EuroSys, 2007*.

[23] KANDULA, S., KATABI, D., DAVIE, B., AND CHARNY, A. Walking the Tightrope: Responsive yet Stable Traffic Engineering. In *Proceedings of ACM SIGCOMM, 2005*.

[24] KANDULA, S., SENGUPTA, S., GREENBERG, A., PATEL, P., AND CHAIKEN, R. The Nature of Data Center Traffic: Measurements & Analysis. In *Proceedings ACM IMC 2009*.

[25] KIM, C., CAESAR, M., AND REXFORD, J. Floodless in SEATTLE: A Scalable Ethernet Architecture for Large Enterprises. In *Proceedings of ACM SIGCOMM, 2008*.

[26] LOCKWOOD, J. W., MCKEOWN, N., WATSON, G., GIBB, G., HARTKE, P., NAOUS, J., RAGHURAMAN, R., AND LUO, J. NetFPGA–An Open Platform for Gigabit-rate Network Switching and Routing. In *Proceedings of IEEE MSE, 2007*.

[27] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM CCR, 2008*.

[28] MIURA, S., BOKU, T., OKAMOTO, T., AND HANAWA, T. A Dynamic Routing Control System for High-Performance PC Cluster with Multi-path Ethernet Connection. In *Proceedings of IEEE IPDPS, 2008*.

[29] MYSORE, R. N., PAMPORIS, A., FARRINGTON, N., HUANG, N., MIRI, P., RADHAKRISHNAN, S., SUBRAMANYA, V., AND VAHDAT, A. PortLand: A Scalable, Fault-Tolerant Layer 2 Data Center Network Fabric. In *Proceedings of ACM SIGCOMM, 2009*.

[30] OKI, E., JING, Z., ROJAS-CESSA, R., AND CHAO, H. J. Concurrent Round-robin-based Dispatching Schemes for Clos-network Switches. *IEEE/ACM TON, 2002*.

[31] SINHA, S., KANDULA, S., AND KATABI, D. Harnessing TCPs Burstiness using Flowlet Switching. In *Proceedings of ACM HotNets, 2004*.

[32] SMILJANIC, A. Rate and Delay Guarantees Provided by Clos Packet Switches with Load Balancing. *Proceedings of IEEE/ACM TON, 2008*.

[33] TURNER, J. S. An Optimal Nonblocking Multicast Virtual Circuit Switch. In *Proceedings of IEEE INFOCOM, 1994*.

[34] VALIANT, L. G., AND BREBNER, G. J. Universal Schemes for Parallel Communication. In *Proceedings of ACM STOC, 1981*.

[35] YAN, H., MALTZ, D. A., NG, T. S. E., GOGINENI, H., ZHANG, H., AND CAI, Z. Tesseract: A 4D Network Control Plane. In *Proceedings of NSDI, 2007*.

Airavat: Security and Privacy for MapReduce

Indrajit Roy Srinath T.V. Setty Ann Kilzer Vitaly Shmatikov Emmett Witchel
The University of Texas at Austin
{*indrajit, srinath, akilzer, shmat, witchel*}*@cs.utexas.edu*

Abstract

We present Airavat, a MapReduce-based system which provides strong security and privacy guarantees for distributed computations on sensitive data. Airavat is a novel integration of mandatory access control and differential privacy. Data providers control the security policy for their sensitive data, including a mathematical bound on potential privacy violations. Users without security expertise can perform computations on the data, but Airavat confines these computations, preventing information leakage beyond the data provider's policy.

Our prototype implementation demonstrates the flexibility of Airavat on several case studies. The prototype is efficient, with run times on Amazon's cloud computing infrastructure within 32% of a MapReduce system with no security.

1 Introduction

Cloud computing involves large-scale, distributed computations on data from multiple sources. The promise of cloud computing is based in part on its envisioned ubiquity: Internet users will contribute their individual data and obtain useful services from the cloud. For example, targeted advertisements can be created by mining a user's clickstream, while health-care applications of the future may use an individual's DNA sequence to tailor drugs and personalized medical treatments. Cloud computing will fulfill this vision only if it supports flexible computations while guaranteeing security and privacy for the input data. To balance the competing goals of a permissive programming model and the need to prevent information leaks, the untrusted code should be confined [30].

Contributors of data to cloud-based computations face several threats to their privacy. For example, consider a medical patient who is deciding whether to participate in a large health-care study. First, she may be concerned that a careless or malicious application operating on her data as part of the study may expose it—for instance, by writing it into a world-readable file which will then be indexed by a search engine. Second, she may be concerned that even if all computations are done correctly and securely, the result itself, *e.g.*, aggregate health-care statistics computed as part of the study, may leak sensitive information about her personal medical record.

Traditional approaches to data privacy are based on syntactic anonymization, *i.e.*, removal of "personally identifiable information" such as names, addresses, and Social Security numbers. Unfortunately, anonymization does not provide meaningful privacy guarantees. High-visibility privacy fiascoes recently resulted from public releases of anonymized individual data, including AOL search logs [22] and the movie-rating records of Netflix subscribers [41]. The datasets in question were released to support legitimate data-mining and collaborative-filtering research, but naïve anonymization was easy to reverse in many cases. These events motivate a new approach to protecting data privacy.

One of the challenges of bringing security to cloud computing is that users and developers want to spend as little mental effort and system resources on security as possible. Completely novel APIs, even if secure, are unlikely to gain wide acceptance. Therefore, a key research question is how to design a practical system that (1) enables efficient distributed computations, (2) supports a familiar programming model, and (3) provides precise, rigorous privacy and security guarantees to data owners, even when the code performing the computation is untrusted. In this paper, we aim to answer this question.

Mandatory access control (MAC) is a useful building block for securing distributed computations. MAC-based operating systems, both traditional [26, 33, 37] and recent variants based on decentralized information flow control [45, 49, 51], enforce a single access control policy for the entire system. This policy, which cannot be overridden by users, prevents information leakage via storage channels such as files, sockets, and program names.

Access control alone does not achieve end-to-end privacy in cloud computing environments, where the input data may originate from multiple sources. The output of the computation may leak sensitive information about the inputs. Since the output generally depends on all input sources, mandatory access control requires that only someone who has access rights to all inputs should have access rights to the output; enforcing this requirement would render the output unusable for most purposes. To be useful, the output of an aggregate computation must be "declassified," but only when it is safe to do so, *i.e.*, when it does not reveal too much information about any single input. Existing access control mechanisms simply delegate this declassification decision to the implementor. In the case of untrusted code, there is no guarantee that the output of the computation does not reveal sensitive

information about the inputs.

In this paper, we present Airavat,[1] a system for distributed computations which provides end-to-end confidentiality, integrity, and privacy guarantees using a combination of mandatory access control and differential privacy. Airavat is based on the popular MapReduce framework, thus its interface and programming model are already familiar to developers. *Differential privacy* is a new methodology for ensuring that the output of aggregate computations does not violate the privacy of individual inputs [11]. It provides a mathematically rigorous basis for declassifying data in a mandatory access control system. Differential privacy mechanisms add some random noise to the output of a computation, usually with only a minor impact on the computation's accuracy.

Our contributions. We describe the design and implementation of Airavat. Airavat enables the execution of trusted and untrusted MapReduce computations on sensitive data, while assuring comprehensive enforcement of data providers' privacy policies. To prevent information leaks through system resources, Airavat runs on SELinux [37] and adds SELinux-like mandatory access control to the MapReduce distributed file system. To prevent leaks through the output of the computation, Airavat enforces differential privacy using modifications to the Java Virtual Machine and the MapReduce framework. Access control and differential privacy are synergistic: if a MapReduce computation is differentially private, the security level of its result can be safely reduced.

To show the practicality of Airavat, we carry out several substantial case studies. These focus on privacy-preserving data-mining and data-analysis algorithms, such as clustering and classification. The Airavat prototype for these experiments is based on the Hadoop framework [2], executing in Amazon's EC2 compute cloud environment. In our experiments, Airavat produced accurate, yet privacy-preserving answers with runtimes within 32% of conventional MapReduce.

Airavat provides a practical basis for secure, privacy-preserving, large-scale, distributed computations. Potential applications include a wide variety of cloud-based computing services with provable privacy guarantees, including genomic analysis, outsourced data mining, and clickstream-based advertising.

2 System overview

Airavat enables the execution of potentially untrusted data-mining and data-analysis code on sensitive data. Its objective is to accurately compute general or aggregate features of the input dataset without leaking information about specific data items.

[1]The all-powerful king elephant in Indian mythology, known as the elephant of the clouds.

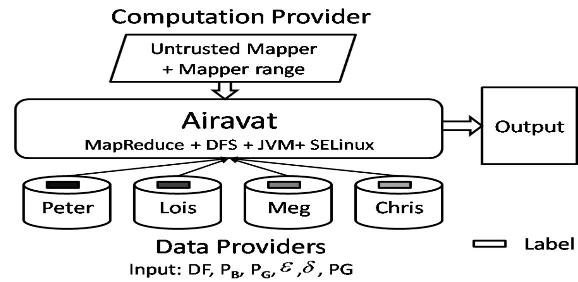


Figure 1: High-level architecture of Airavat.

As a motivating scenario, consider an online retailer, BigShop, which holds a large database of customer transactions. For now, assume that all records in the database have the form $\langle$customer, order, date$\rangle$, with only one record per customer. A machine learning expert, Bob, pays BigShop to mine the data for certain transaction patterns. BigShop loads the data into the Hadoop framework and Bob writes the MapReduce code to analyze it.

Such computations are commonly used for targeted advertising and customer relationship management, but we will keep the example simple for clarity and assume that Bob wants to find the total number of orders placed on a particular date D. He writes a mapper that looks at each record and emits the key/value pair $\langle K$, order$\rangle$ if the date on the record is D. Here, K is a string constant. The reducer simply sums up the values associated with each key K and outputs the result.

The main risk for BigShop in this scenario is the fact that Bob's code is untrusted and can therefore be unintentionally buggy or even actively malicious. Because Bob's mapper has direct access to BigShop's proprietary transaction records, it can store parts of these data in a file which will be later accessed by Bob, or it can send them over the network. Such a leak would put BigShop at a commercial disadvantage and may also present a serious reputational risk if individual BigShop transactions were made public without the consent of the customer.

The output of the computation may also leak information. For example, Bob's mapper may signal the presence (or absence) of a certain customer in the input dataset by manipulating the order count for a particular day: if the record of this customer is in the dataset, the mapper outputs an order count of 1 million; otherwise, it outputs zero. Clearly, the result of the computation in this case violates the privacy of the customer in question.

2.1 Architecture of Airavat

The three main entities in our model are (1) the data provider (BigShop, in our motivating example), (2) the computation provider (Bob, sometimes referred to as a user making a query), and (3) the computation frame-

work (Airavat). We aim to prevent malicious computation providers from violating the privacy policy of the data provider(s) by leaking information about individual data items.

Computation providers write their code in the familiar MapReduce paradigm, while data providers specify the parameters of their privacy policies. We relieve data providers of the need to audit computation providers' code for privacy compliance.

Figure 1 gives an overview of the Airavat architecture. Airavat consists of modifications to the MapReduce framework, the distributed file system, and the Java virtual machine with SELinux as the underlying operating system. Airavat uses SELinux's mandatory access control to ensure that untrusted code does not leak information via system resources, including network connections, pipes, or other storage channels such as names of running processes. To prevent information leakage through the output of the computation, Airavat relies on a differential privacy mechanism [11].

Data providers put access control labels on their data and upload them to Airavat. Airavat ensures that the result of a computation is labeled with the union of all input labels. A data provider, D, can set the *declassify flag* (**DF** in Table 1) to true if he wants Airavat to remove his label from the output when it is safe to do so. If the flag is set, Airavat removes D's label if and only if the computation is differentially private. Data providers must also create a privacy policy by setting the value of several privacy parameters (explained in Section 4).

The computation provider must write his code in the Airavat programming model, which is close to standard MapReduce. The *sensitivity* of the function being computed determines the amount of perturbation that will be applied to the output to ensure differential privacy (§ 4). Therefore, in Airavat the computation provider must supply an upper bound on the sensitivity of his computation by specifying the range of possible values that his mapper code may output. Airavat then ensures that the code never outputs values outside the declared range and perturbs those within the range so as to ensure privacy (§ 5.1). If malicious or incorrect code tries to output a value outside its declared range, the enforcement mechanism guarantees that no privacy breach will occur, but the results of the computation may no longer be accurate.

Apart from specifying the parameters mentioned above, neither the data provider, nor the computation provider needs to understand the intricacies of differential privacy and its enforcement.

2.2 Trusted computing base of Airavat

Airavat trusts the cloud provider and the cloud-computing infrastructure. It assumes that SELinux correctly implements MAC and relies on the MAC features added to the MapReduce distributed file system, as well as on the (modified) Java virtual machine to enforce certain properties of the untrusted mapper code (see Section 5.3). Airavat includes trusted implementations of several reducer functions.

Participant	Input
Data provider	Labeled data (**DB**) Declassify flag (**DF**) Privacy parameters ($\boldsymbol{\epsilon}, \boldsymbol{\delta}$) Privacy budget ($\mathbf{P_B}$) Code to determine privacy groups* (**PG**)
Computation provider (user making a query)	Mapper range ($\mathbf{M_{min}}, \mathbf{M_{max}}$) Independent mapper code (**Map**) Number of outputs (**N**) Code to determine partitions* (**PC**) Map of partition to output* (**PM**) Max keys output for any privacy group* (**n**)
Airavat	Trusted reducer code (**Red**) Modified MapReduce (**MMR**) Modified distributed file system (**MDFS**) SELinux policy (**SE**)

Table 1: Parameters and components provided by different participants. Optional parameters are starred.

We assume that the adversary is a malicious computation provider who has full control over the mapper code supplied to Airavat. The adversary may attempt to access the input, intermediate, and output files created by this code, or to reconstruct the values of individual inputs from the result of the computation.

2.3 Limitations of Airavat

Airavat cannot confine every computation performed by untrusted code. For example, a MapReduce computation may output key/value pairs. Keys are text strings that provide a storage channel for malicious mappers. In general, Airavat cannot guarantee privacy for computations which output keys produced by untrusted mappers. In many cases, privacy *can* be achieved by requiring the computation provider to declare the key in advance and then using Airavat to compute the corresponding value in a differentially private way.

MapReduce computations that necessarily output keys require trusted mappers. For example, printing the top K items sold in a store involves printing item names. Because a malicious mapper can use a name to encode information about individual inputs, this computation requires trusted mappers. By contrast, the number of iPods sold can be calculated using an untrusted mapper because the key ("iPod" in this case) is known prior to the answer being released. (See Section 5 for details.)

3 MapReduce and MAC

Table 1 lists the components of the system contributed by the data provider(s), computation provider(s), and Airavat. The following discussion explains entries in the table

in the context of MapReduce computations, mandatory access control, or differential privacy. We place a bold label in the text to indicate that the discussion is about a particular row in the table.

3.1 MapReduce

MapReduce [9] is a framework for performing data-intensive computations in parallel on commodity computers. A MapReduce computation reads input files from a distributed file system which splits the file into multiple chunks. Each chunk is assigned to a *mapper* which reads the data, performs some computation, and outputs a list of key/value pairs. In the next phase, *reducers* combine the values belonging to each distinct key according to some function and write the result into an output file. The framework ensures fault-tolerant execution of mappers and reducers while scheduling them in parallel on any machine (node) in the system. In MapReduce, *combiners* are an optional processing stage before the reduce phase. They are a performance optimization, so for simplicity, we defer them to future work. Airavat secures the execution of untrusted mappers (**Map**) using a MAC OS (**SE**), as well as modifications to the MapReduce framework (**MMR**) and distributed file system (**MDFS**).

3.2 Mandatory access control

Mandatory access control (MAC) assigns security attributes to system resources and uses these attributes to constrain the interaction of subjects (*e.g.*, processes) with objects (*e.g.*, files). In contrast to discretionary access control (*e.g.*, UNIX permissions), MAC systems (1) check permissions on every operation and transitively enforce access restrictions (*e.g.*, processes that access secret data cannot write non-secret files) and (2) enforce access rules specified by the system administrator at all times, without user override. MAC systems include mainstream implementations such as SELinux [37] and AppArmor [1] which appear in Linux distributions, as well as research prototypes [45, 49, 51] which implement a MAC security model called decentralized information flow control (DIFC). Our current implementation uses SELinux because it is a mature system that provides sufficient functionality to enforce Airavat's security policies.

SELinux divides subjects and objects into groups called *domains* or *types*. The domain is part of the security attribute of system resources. A domain can be thought of as a sandbox which constrains the permissions of the process. For example, the system administrator may specify that a given domain can only access files belonging to certain domains. In SELinux, one can specify rules that govern transition from one domain to another. Generally, a transition occurs by executing a program declared as the entry point for a domain.

In SELinux, users are assigned roles. A role governs the permissions granted to the user by determining which domains he can access. For example, the system administrator role (`sysadm_r`) has permissions to access the `ifconfig_t` domain and can perform operations on the network interface. In SELinux, access decisions are declared in a policy file which is customized and configured by the system administrator. The Airavat-specific SELinux policy to enforce mandatory access control and declassification (**SE, DF**) is described in Section 6.

4 Differential privacy

The objective of Airavat is to enable large-scale computation on data items that originate from different sources and belong to different owners. The fundamental question of what it means for a computation to preserve the privacy of its inputs has been the subject of much research (see Section 9).

Airavat uses the recently developed framework of *differential privacy* [11, 12, 13, 14] to answer this question. Intuitively, a computation on a set of inputs is differentially private if, for any possible input item, the probability that the computation produces a given output does not depend much on whether this item is included in the input dataset or not. Formally, a computation $\mathcal{F}$ satisfies (ϵ, δ)-differential privacy [15] (where ϵ and δ are privacy parameters) if, for all datasets D and D' whose only difference is a single item which is present in D but not D', and for all outputs $S \subseteq Range(\mathcal{F})$,

$$\Pr[\mathcal{F}(D) \in S] \leq \exp(\epsilon) \times \Pr[\mathcal{F}(D') \in S] + \delta$$

Another intuitive way to understand this definition is as follows. Given the output of the computation, one cannot tell if any specific data item was used as part of the input because the probability of producing this output would have been the same even without that item. Not being able to tell whether the item was used at all in the computation precludes learning any useful information about it from the computation's output alone.

The computation $\mathcal{F}$ must be randomized to achieve privacy (probability in the above definition is taken over the randomness of $\mathcal{F}$). Deterministic computations are made privacy-preserving by adding random noise to their outputs. The privacy parameter ϵ controls the tradeoff between the accuracy of the output and the probability that it leaks information about any individual input.

The purpose of the δ parameter is to relax the multiplicative definition of privacy for certain kinds of computation. Consider `TOPWORDS`, which calculates the frequency of words in a corpus and outputs the top 10 words. Let D and D' be two large corpora; the only difference is that D contains a single instance of the word "sesquipedalophobia," while D' does not. The probability that `TOPWORDS` outputs "sesquipedalophobia" is very small on input D and zero on input D'. The mul-

tiplicative bound on the ratio between these probabilities required by differential privacy cannot be achieved (since one of the probabilities is zero), but the absolute difference is very small. The purpose of δ in the definition is to allow a small absolute difference in probabilities. In many of the computations considered in this paper, this situation does not arise and δ can be safely set to 0.

In Section 9, we discuss why differential privacy is the "right" concept of privacy for cloud computing. The most important feature of differential privacy is that it does not make any assumptions about the adversary. When satisfied, it holds regardless of the auxiliary or prior knowledge that the adversary may possess. Furthermore, differential privacy is composable: a composition of two differentially private computations is also differentially private (of course, ϵ and δ may increase).

There are many mechanisms for achieving differential privacy [5, 14, 17, 40]. In this paper, we will use the mechanism that adds Laplacian noise to the output of a computation $f : D \to R^k$:

$$f(x) + (\text{Lap}(\Delta f/\epsilon))^k$$

where $\text{Lap}(\Delta f/\epsilon)$ is a symmetric exponential distribution with standard deviation $\sqrt{2}\Delta f/\epsilon$.

Privacy groups. To provide privacy guarantees which are meaningful to users, it is sometimes important to consider input datasets that differ not just on a single record, but on a group of records (**PG**). For example, when searching for a string within a set of documents, each input might be a line from a document, but the privacy guarantee should apply to whole documents. Differential privacy extends to privacy groups via composability: the effect of n input items on the output is at most n times the effect of a single item.

4.1 Function sensitivity

A function's *sensitivity* measures the maximum change in the function's output when any single item is removed from or added to its input dataset. Intuitively, the more sensitive a function, the more information it leaks about the presence or absence of a particular input. Therefore, more sensitive functions require the addition of more random noise to their output to achieve differential privacy.

Formally, the sensitivity of a function $f : D \to R^k$ is

$$\Delta(f) = \max_{D,D'} \|f(D) - f(D')\|_1$$

for any D, D' that are identical except for a single element, which is present in D, but not in D'. In this paper, we will be primarily interested in functions that produce a single output, *i.e.*, $k = 1$.

Many common functions have low sensitivity. For example, a function that counts the number of elements satisfying a certain predicate has sensitivity 1 (because the count can change by at most 1 when any single element is removed from the dataset). The sensitivity of a function that sums up integers from a bounded range is the maximum value in that range. Malicious functions that aim to leak information about an individual input or signal its presence in the input dataset are likely to be sensitive because their output must necessarily differentiate between the datasets in which this input is present and those in which it is not present.

In general, estimating the sensitivity of arbitrary untrusted code is difficult. Therefore, we require the computation provider to furnish the range of possible outputs for his mappers and use this range to derive *estimated sensitivity*. Estimated sensitivity is then used to add sufficient random noise to the output and guarantee privacy regardless of what the untrusted code does. If the code is malicious and attempts to output values outside its declared range, the enforcement mechanism will chose a value within the range. The computation still guarantees privacy, but the results may no longer be accurate (§ 5.1).

Sensitivity of SUM. Consider a use of `SUM` that takes as input 100 integers and returns their sum. If we know in advance that the inputs are all 0 or 1, then the sensitivity of `SUM` is low because the result varies at most by 1 depending on the presence of any given input. Only a little noise needs to be added to the sum to achieve privacy.

In general, the sensitivity of `SUM` is determined by the largest possible input. In this example, if one input *could* be as big as 1,000 and the rest are all 0 or 1, the probability of outputting any given sum should be almost the same with or without 1,000. Even if all actual inputs are 0 or 1, a lot of noise must be added to the output of `SUM` in order to hide whether 1,000 was among the inputs.

Differential privacy works best for low-sensitivity computations, where the maximum influence any given input can have on the output of the computation is low.

4.2 Privacy budget

Data providers may want an absolute privacy guarantee that holds regardless of the number and nature of computations carried out on the data. Unfortunately, an absolute privacy guarantee cannot be achieved for meaningful definitions of privacy. A fundamental result by Dinur and Nissim [10] shows that the entire dataset can be decoded with a linear (in the size of the dataset) number of queries. This is a serious, but inevitable, limitation. Existing privacy mechanisms which are not based on differential privacy either severely limit the utility of the data, or are only secure against very restricted adversaries (see [13] and Section 9).

The composability of differential privacy and the need to restrict the number of queries naturally give rise to the concept of a "*privacy budget*" ($\mathbf{P_B}$) [17, 38]. Each differentially private computation with a privacy parameter of ϵ

results in subtracting ϵ from this budget. Once the privacy budget is exhausted, results can no longer be automatically declassified. The need to pre-specify a limit on how much computation can be done over a given dataset constrains some usage scenarios. We emphasize that there are no definitions of privacy that are robust, composable, and achievable in practice without such a limit.

After the privacy budget has been exhausted, Airavat still provides useful functionality. While the output can no longer be automatically declassified without risking a privacy violation, Airavat still enforces access control restrictions on the untrusted code and associates proper access control labels with the output. In this case, outputs are no longer public and privacy protection is based solely on mandatory access control.

5 Enforcing differential privacy

Airavat supports both trusted and untrusted mappers. Because reducers are responsible for enforcing privacy, they must be trusted. The computation provider selects a reducer from a small set included in the system.

The outputs of mappers and reducers are lists of key/-value pairs. An untrusted, potentially malicious mapper may try to leak information about an individual input by encoding it in (1) the values it outputs, (2) the keys it outputs, (3) the order in which it outputs key/value pairs, or (4) relationships between output values of different keys.

MapReduce keys are arbitrary strings. Airavat cannot determine whether a key encodes sensitive information. The mere presence of a particular key in the output may signal information about an individual input. Therefore, Airavat never outputs any keys produced by untrusted mappers. Instead, the computation provider submits a key or list of keys as part of the query and Airavat returns (noisy) values associated with these keys. As explained below, Airavat always returns a value for every key in the query, even if none of the mappers produced this key. This prevents untrusted mappers from signaling information by adding or removing keys from their output.

For example, Airavat can be used to compute the noisy answer to the query "*What is the total number of iPods and pens sold today?*" (see the example in Section 5.4) because the two keys `iPod` and `pen` are declared as part of the computation. The query "*List all items and their sales*" is not allowed in Airavat, unless the mapper trusted. The reason is that a malicious mapper can leak information by encoding it in item names.

Trusted Airavat reducers always sort keys prior to outputting them. Therefore, a malicious mapper cannot use key order as a channel to leak information about a particular input record.

A malicious mapper may attempt to encode information by emitting a certain combination of values associated with different keys. As explained below, trusted reducers use the declared output range of mappers to add sufficient noise to ensure differential privacy for the outputs. In particular, a combination C of output values across multiple keys does not leak information about any given input record r because the probability of Airavat producing C is approximately the same with or without r in the input dataset.

In the rest of this section, we explain how Airavat enforces differential privacy for computations involving untrusted mappers. We use BigShop from Section 2 as our running example. We also briefly describe a broader class of differentially private computations which can be implemented using trusted mappers.

5.1 Range declarations and estimated sensitivity

Airavat reducers enforce differential privacy by adding exponentially distributed noise to the output of the computation. The sensitivity of the computation determines the amount of noise: the noise must be sufficient to mask the maximum influence that any single input record can have on the output (§ 4.1).

In the case of untrusted mappers, the function(s) they compute and their sensitivity are unknown. To help Airavat estimate sensitivity, we require the computation provider to declare the range of output values (M_{min}, M_{max}) that his mapper can produce. Airavat combines this range with the sensitivity of the function implemented by the trusted reducer (**Red**) into *estimated sensitivity*. For example, estimated sensitivity of the `SUM` reducer is $max(|M_{max}|, |M_{min}|)$, because any single input can change the output by at most this amount.

The declared mapper range can be greater or smaller than the true global sensitivity of the function computed by the mapper. While global sensitivity measures the output difference between any two inputs that differ in at most one element (§ 4.1), the mapper range captures the difference between *any* two inputs. That said, the computation provider may assume that all inputs for the current computation lie in a certain subset of the function's domain, so the declared range may be lower than the global sensitivity. In our clustering case study (§ 8.4), such an assumption allows us to obtain accurate results even though global sensitivity of clustering is very high (on "bad" input datasets, a single point can significantly change the output of the clustering algorithms).

The random noise added by Airavat to the output of MapReduce computations is a function of the data provider's privacy parameter ϵ and the estimated sensitivity. For example, Airavat's `SUM` reducer adds noise from the Laplace distribution, Lap($\frac{b}{\epsilon}$), where $b = max(|M_{max}|, |M_{min}|)$.

Example. In the BigShop example, Bob writes his own mapper and uses the `SUM` reducer to compute the total number of orders placed on date D. Assuming that a cus-

tomer can order at most 25 items on any single day, Bob declares his mapper range as $(0, 25)$. The estimated sensitivity is 25 because the presence or absence of a record can affect the order total by at most 25.

Privacy groups. In the BigShop example, we may want to provide privacy guarantees at the level of customers rather than records (a single customer may have multiple records). Airavat supports privacy groups (§ 4), which are collections of records that are jointly present or absent in the dataset. The data provider supplies a program (**PG**) that takes a record as input and emits the corresponding group identifier, `gid`. Airavat attaches these identifiers to key/value pairs to track the dispersal of information from each input privacy group through intermediate keys to the output. The mapper range declared by the computation provider is interpreted at the group level. For example, suppose that each BigShop record represents a purchase, a customer can make at most 10 purchases a day, and each purchase contains at most 25 orders. If all orders of a single customer are viewed as a privacy group, then the mapper range is $(0, 250)$.

5.2 Range enforcement

To prevent malicious mappers from leaking information about inputs through their output values, Airavat associates a *range enforcer* with each mapper. The range enforcer checks that the value in each key/value pair output by the mapper lies within its declared range. This check guarantees that the actual sensitivity of the computation performed by the mapper does not exceed the estimated sensitivity, which is based on the declared range. If a malicious mapper outputs a value outside the range, the enforcer replaces it with a value inside the range. In the latter case, differential privacy holds, but the computation may no longer produce accurate or meaningful results.

Range enforcement in Airavat prioritizes privacy over accuracy. If the computation provider declares the range incorrectly, the computation remains differentially private. However, the results are not meaningful and the provider gets no feedback about the problem, because any such feedback would be an information leak. The lack of feedback may seem unsatisfying, but other systems that tightly regulate information flow make similar tradeoffs. For example, MAC systems Flume and Asbestos make pipes (used for interprocess communication) unreliable and do not give the user any feedback about their failure because such feedback would leak information [29, 49].

Providing a mapper range is simple for some computations. For example, Netflix movie ratings (§8.3) are always between 1 and 5. When computing the word count of a set of documents, however, estimating the mapper range is more difficult. If each document is at most N words, and the document is a privacy group, then the $0 - N$ range will guarantee privacy of individual documents. Depending on the number of documents, such a large range may result in adding excessive noise. For some domains, it might not be possible to obtain a reasonable estimate of the mapper's range. Airavat gives accurate results only when the computation provider understands the sensitivity of his computation.

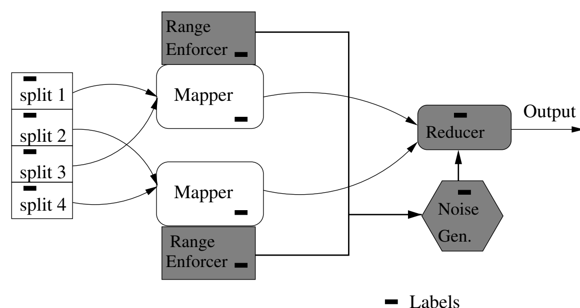


Figure 2: Simplified overview of range enforcers and noise generation. Trusted components are shaded.

In the BigShop example, the range enforcer ensures that in every $\langle$K, V$\rangle$ pair output by the mapper, $0 \leq V \leq 25$. Suppose a malicious mapper attempts to leak information by outputting $1,000$ when Alice's record is in the input dataset and 0 otherwise. Because $1,000$ is outside the declared range, the range enforcer will replace it with, say, 12.5. The difference between 0 and 12.5 is less than the estimated sensitivity. Therefore, enough noise will be added so that one cannot tell, by looking at the output, whether this output was obtained by adding noise to 12.5 or 0. The noisy output thus does not reveal whether Alice's record was present in the input dataset or not.

Distributed range enforcement. A single MapReduce operation may execute mappers on many different machines. These mappers may process input elements with the same key or privacy group. Airavat associates a range enforcer with each mapper and merges their states at the end of the map phase. After merging, Airavat ensures that the values corresponding to each key or privacy group are within the declared range (see Figure 2).

Example: "noisy sum." Figure 3 illustrates differential privacy enforcement with an untrusted mapper and the `SUM` reducer. This "noisy sum" primitive was shown by Blum *et al.* [6] to be sufficient for privacy-preserving computation of all algorithms in the statistical query model [27], including k-Means, Naive Bayes, principal component analysis, and linear classifiers such as perceptrons (for a slightly different definition of privacy).

Each input record is its own privacy group. The computation provider supplies the implementation of the actual mapper function **Map**, which converts every input record into a list of key/value pairs.

```
// Inputs and definitions
Data owner: DB, ε, δ = 0,  P_B,
Computation provider: Map, M_min, M_max, N
Airavat: SUM (trusted reducer, Red)
b = max(|M_max|, |M_min|)
μ = (M_max − M_min)/2
----------------------------------------
// Map phase
if(P_B − ε×N < 0){
   print ``Privacy limit exceeded'';
   TERMINATE
}
P_B = P_B − ε×N
For(Record r in DB){
   (k_0, v_0),...,(k_n, v_n) = Map(r)
   For(i: 1 to n){
      if(v_i < M_min or v_i > M_max) {
        v_i = μ }
   }
   emit⟨k_0, v_0⟩...⟨k_n, v_n⟩
}
----------------------------------------
// Reduce phase
count = N
Reduce(Key k, List val){
   if(--count ≤ 0) { Skip }
   V = SUM(val)
   print V + Lap(b/ε)
}
for(i: count to 0) {
   print Lap(b/ε) }
```

Figure 3: Simplified pseudo-code demonstrating differential privacy enforcement.

5.3 Mapper independence

Airavat forces all invocations of a mapper in a given MapReduce computation to be *independent*. Only a single input record is allowed to affect the key/value pairs output by the mapper. The mapper may not store the key/value pair(s) produced from an input record and use them later, when computing the key/value pair for another record. Without this restriction, estimated sensitivity used in privacy enforcement may be lower than the actual sensitivity of the mapper, resulting in a potential privacy violation. Mappers can only create additional keys for the same input record, they cannot merge information contained in different input records. We ensure mapper independence by modifying the JVM (§ 7.3).

Each mapper is permitted by the Airavat JVM to initialize itself once by overriding the `configure` function, called when the mapper is instantiated. To ensure independence, during initialization the mapper may not read any files written in this MapReduce computation.

5.4 Managing multiple outputs

A MapReduce computation may output more than one key/value pair (*e.g.*, Figure 3). The computation provider must specify the number of output keys (**N**) beforehand; otherwise, the number of outputs can become a channel through which a malicious mapper can leak information about the inputs. If a computation produces more (fewer) than the declared number of outputs, then Airavat removes (creates) outputs to match the declared value.

Range restrictions are enforced separately for each (privacy group, key) pair. Therefore, random noise is independently added to all values associated with the final output keys. Recall that Airavat never outputs a key produced by an untrusted mapper. Instead, the computation provider must specify a key or list of keys as part of the query, and Airavat will return the noisy values associated with each key in the query. For such queries, **N** can be calculated automatically.

In general, each output represents a separate release of information about the same input. Therefore, Airavat must subtract more than one ϵ from the privacy budget (see Figure 3). If different outputs are based on disjoint parts of the input, then smaller deductions from the privacy budget are needed (see below).

Example. Consider the BigShop example, where each record includes the customer's name, a list of products, and the number of items bought for each product (*e.g.*, [Joe, iPod, 1, pen, 10]). The privacy group is the customer, and each customer may have multiple records. Bob wants to compute the total number of iPods and pens sold. Bob must specify that he expects two outputs. If he specifies the keys for these outputs as part of the query (*e.g.*, "iPod" and "pen"), then the keys will be printed. Otherwise, only the values will be printed. Airavat subtracts 2ϵ from the privacy budget for this query.

Bob's mapper, after reading a record, outputs the product name and the number of sold items (*e.g.*, ⟨iPod, 1⟩, ⟨pen, 10⟩—note that more than one key/value pair is output for each record). Bob also declares the mapper range for each key, *e.g.*, $(0, 5)$ for the number of iPods bought and $(0, 25)$ for the number of pens. Airavat range enforcers automatically group the values by customer name and enforce the declared range for each item count. The final reducer adds random noise to the total item counts.

Computing on disjoint partitions of the input. When different outputs depend on disjoint parts of the input, the MapReduce computation can be decomposed into independent, parallel computations on independent datasets, and smaller deductions from the privacy budget are sufficient to ensure differential privacy. To help Airavat track and enforce the partitioning of the input, the computation provider must (1) supply the code (**PC**) that assigns input records to disjoint partitions, and (2) specify which of the

declared outputs will be based on which partition (**PM**).

The **PC** code is executed as part of the initial mapper. For each key/value pair generated by a mapper, Airavat constructs records of the form $\langle key, value, gid, pid \rangle$, where `gid` is the privacy group identifier and `pid` is the partition identifier.

The computation provider declares which partition produces which of the **N** final outputs (**PM**). Airavat uses **PM** to calculate **p**, the maximum number of final outputs that depend on any single partition. If **PC** and **PM** are not provided, Airavat sets **p** to equal **N**. Airavat charges $\epsilon \times \min(\mathbf{N}, \mathbf{p})$ from the privacy budget. For example, a computation provider may partition the BigShop data into two cities `Austin` and `Seattle` which act as the partition identifiers. He then specifies that the MapReduce computation will have 8 outputs, the first five of which are calculated from the `Austin` partition and the next three from the `Seattle` partition. In this example, $N = 8$, $p = 5$, and to run the computation, Airavat will subtract 5ϵ from the privacy budget. In Figure 3, $\epsilon \times N$ is charged to the privacy budget because the N outputs depend on the entire input, *not* on disjoint partitions.

Airavat enforces the partitioning declared by the computation provider. Trusted Airavat reducers use partition identifiers to ensure that only key/value pairs that have the correct `pid` are combined to generate the output. Airavat uses **PM** for computations on disjoint partitions in the same way as it uses **N** for unpartitioned data. If the number of outputs for a partition is less (more) than what is specified by **PM**, Airavat adds (deletes) outputs.

5.5 Trusted reducers and reducer composition

Trusted reducers such as `SUM` and `COUNT` are executed directly on the output of the mappers. The computation provider can combine these reducers with any untrusted mapper, and Airavat will ensure differential privacy for the reducer's output. For example, to calculate the total number of products sold by BigShop, the mapper will be responsible for the parsing logic and manipulation of the data. `COUNT` is a special case of `SUM` where the output range is $\{0, 1\}$. `MEAN` is computed by calculating the `SUM` and dividing it by the `COUNT`.

Reducers can be composed sequentially. `THRESHOLD`, `K-COUNT`, and `K-SUM` reducers are most useful when applied to the output of another reducer. `THRESHOLD` prints the outputs whose value is more than C, where C is a parameter. `K-COUNT` counts the number of records, and `K-SUM` sums the values associated with each record. For example, to count the number of distinct words occurring in a document, one can first write a MapReduce computation to group the words and then apply `K-COUNT` to calculate the number of groups. The sensitivity of `K-COUNT` is equal to the maximum number of distinct keys that a mapper can output after processing any input record.

5.6 Enforcing δ

Privacy guarantees associated with the `THRESHOLD` reducer may have non-zero δ. Intuitively, δ bounds the probability that the values generated from a given record will exceed the threshold and appear in the final output. Assuming that the mapper outputs at most **n** keys after processing a single record and the threshold value is C,

$$\delta \leq \frac{\mathbf{n}}{2} \exp\left(\epsilon \cdot (1 - \frac{C}{\Delta f})\right)$$

The proof is omitted because of space constraints and appears in a technical report [46]. When the computation provider uses the `THRESHOLD` reducer, Airavat first calculates the value of δ. If it is less than the bound specified by the data provider, then computation can proceed; otherwise it is aborted.

5.7 Mapper composition

Multiple mappers $\{M_1, \ldots, M_j\}$ can be chained one after another, followed by a final reducer R_j. Each mapper after the initial mapper propagates the partition identifier (`pid`) and privacy group (`gid`) values from the input record to output key/value pairs. Airavat enforces the declared range for the output of the final mapper M_j. Noise is added only once by the final reducer R_j.

To reduce the charge to the privacy budget, the computation provider can specify the maximum number of keys **n** that any mapper can output after reading records from a single privacy group. If provided, Airavat will enforce that maximum. If **n** is not provided, Airavat sets **n** equal to **N**. If a mapper generates more than **n** key/value pairs, Airavat will only pass **n** randomly selected pairs to the next mapper.

When charging the total cost of a composed computation to the privacy budget, Airavat uses $\epsilon \times \min(\mathbf{N}, \mathbf{p}, \mathbf{n}^j)$ where j is the number of composed mappers, **p** is the maximum number of outputs from any partition (§5.4), and **N** is the total number of output keys. If the computation provider supplies the optional arguments, then $\mathbf{N} > \mathbf{p} > \mathbf{n}^j$ results in a more economical use of the privacy budget.

MapReduce composition not supported. Airavat supports composition of mappers and composition of reducers, but not general composition of MapReduce computations (*i.e.*, reducer followed by another mapper). For many reducers, the output of a MapReduce depends on the inputs in a complex way that Airavat cannot easily represent, making sensitivity calculations difficult.

In the future, we plan to investigate MapReduce composition for reducers that do not combine information associated with different keys (*e.g.*, those corresponding to a "select" statement).

5.8 Choosing privacy parameters

Providers of sensitive data must supply privacy parameters ϵ and δ, as well as the privacy budget P_B, in order for their data to be used in an Airavat computation. These parameters are part of the differential privacy model. They control the tradeoff between accuracy and privacy. It is not possible to give a generic recommendation for setting their values because they are highly dependent on the type of the data, the purpose of the computation, privacy threats that the data provider is concerned about, *etc.*

As ϵ increases, the amount of noise added to the output decreases. Therefore, the output becomes more accurate, but there is a higher chance that it reveals the presence of a record in the input dataset. In many cases, the accuracy required determines the minimum ϵ-privacy that can be guaranteed. For example, in Section 8.5 we classify documents in a privacy-preserving fashion. Our experiments show that to achieve 95% accuracy in classification, we need to set ϵ greater than 0.6.

Intuitively, δ bounds the probability of producing an output which can occur only as a result of a particular input (see Section 4). Clearly, such an output immediately reveals the presence of the input in question. In many computations—for example, statistical computations where each input datapoint is a single number—δ should be set to 0. In our AOL experiment (§8.2), which outputs the search queries that occur more than a threshold number of times, δ is set to a value close to the number of unique users. This value of δ bounds the probability that a single user's privacy is breached due to the release of his search query.

The privacy budget (P_B) is finite. If data providers specify a single privacy budget for all computation providers, then one provider can exhaust more than its fair share. Data providers could specify privacy budgets for each computation provider to ensure fairness. Managing privacy budgets is an administrative issue inherent to all differential privacy mechanisms and orthogonal to the design of Airavat.

5.9 Computing with trusted mappers

While basic differential privacy only applies to computations that produce numeric outputs, it can be generalized to discrete domains (*e.g.*, discrete categories or strings) using the exponential mechanism of McSherry and Talwar [40]. In general, this requires both mappers and reducers to be trusted, because keys are an essential part of the system's output. Our prototype includes an implementation of this mechanism for simple cases, but we omit the definition and discussion for brevity.

As a case study, one of the authors of this paper ported CloudBurst, a genome mapping algorithm written for MapReduce [47], to Airavat. The CloudBurst code contains two mappers and two reducers (3,500 lines total, including library routines). The mappers are not independent and the reducers perform non-trivial computation.

The entire system was ported in a week. If a reducer was non-trivial, it was replaced by an identity reducer and its functionality was executed as the second mapper stage. This transformation was largely syntactic. Some work was required to make the mappers independent, and about 50 lines of code had to be added for differential privacy enforcement.

6 Enforcing mandatory access control

This section describes how Airavat confines MapReduce computations, preventing information leaks via system resources by using mandatory access control mechanisms. Airavat uses SELinux to execute untrusted code in a sandbox-like environment and to ensure that local and HDFS files are safeguarded from malicious users. While decentralized information flow control (DIFC) [45, 49, 51] would provide far greater flexibility for access control policies within Airavat, only prototype DIFC operating systems exist. By contrast, SELinux is a broadly deployed, mature system.

6.1 SELinux policy

Airavat's SELinux policy creates two domains, one trusted and the other untrusted. The trusted components of Airavat, such as the MapReduce framework and DFS, execute inside the trusted domain. These processes can read and write trusted files and connect to the network. Untrusted components, such as the user-provided mapper, execute in the untrusted domain and have very limited permissions.

Table 2 shows the different domains and how they are used. The `airavatT_t` type is a trusted domain used by the MapReduce framework and the distributed file system. Airavat labels executables that launch the framework and file system with the `airavatT_exec_t` type so the process executes in the trusted domain. This trusted domain reads and writes only trusted files (labeled with `airavatT_rw_t`). No other domain is allowed to read or write these files. For example, the distributed file system stores blocks of data in the underlying file system and labels files containing those blocks with `airavatT_rw_t`.

In certain cases Airavat requires the trusted domain to create configuration files that can later be read by untrusted processes for initialization. Airavat uses the `airavatT_notsec_t` domain to label configuration files which do not contain any secrets but whose integrity is guaranteed. Since MapReduce requires network communication for transferring data, our policy allows network access by the trusted domain.

Only privileged users may enter the trusted domain. To implement this restriction, Airavat creates a trusted SELinux user called `airavat_user`.

Domain	Object labeled	Remark
airavatT_t	Process	Trusted domain. Can access airavatT_*_t and common domains like sockets, networking, etc.
airavatT_exec_t	Executable	Used to transition to the airavatT_t domain.
airavatT_rw_t	File	Used to protect trusted files.
airavatT_notsec_t	File	Used to protect configuration files that contain no secrets. Can be read by untrusted code for initialization.
airavatU_t	Process	Untrusted domain. Can access only airavatT_notsec_t and can read and write to pipes of the airavatT_t domain.
airavatU_exec_t	Executable	Used to transition to the airavatU_t domain.
airavavat_user	User type	Trusted user who can transition to the airavatT_t domain.

Table 2: SELinux domains defined in Airavat and their usage.

Only `airavat_user` can execute files labeled with `airavatT_exec_t` and transition to the trusted domain. The system administrator maps a Linux user to `airavat_user`.

The untrusted domain, `airavatU_t`, has very few privileges. A process in the untrusted domain cannot connect to the network, nor read or write files. There are two exceptions to this rule. First, the untrusted domain can read configuration files of the type `airavatT_notsec_t`. Second, it can communicate with the trusted domain using pipes. All communication with the mapper happens via these pipes which are established by the trusted framework. A process can enter the untrusted domain by executing a file of the type `airavatU_exec_t`. In our implementation, the framework transitions to the untrusted domain by executing the JVM that runs the mapper code.

Each data provider labels its input files (**DB**) with a domain specific to that provider. Only the trusted `airavatT_t` domain can read files from all providers. The output of a computation is stored in a file labeled with the trusted domain `airavatT_rw_t`. Data providers may set their declassify flag if they agree to declassify the result when Airavat guarantees differential privacy. If all data providers agree to declassify, then the trusted domain label is removed from the result when differential privacy holds. If only a subset of the data providers agree to declassify, then the result is labeled by a new domain, restricted to entities that have permission from all providers who chose to retain their label. Since creating domains in SELinux is a cumbersome process, our current prototype only supports full declassification. DIFC makes this ad hoc sharing among domains easy.

6.2 Timing channels

A malicious mapper may leak data using timing channels. MapReduce is a batch-oriented programming style where most programs do not rely on time. The bandwidth of covert timing channels is reduced by making clocks noisy and low-resolution [25]. Airavat currently denies untrusted mappers access to the high-resolution processor cycle counter (TSC), which is accessed via Java APIs. A recent timing attack requires the high-definition processor counter to create a channel with 0.2 bits per second capacity [44]. Without the TSC, the data rate drops three orders of magnitude.

We are working to eliminate all obvious time-based APIs from the Airavat JVM for untrusted mappers, including `System.currentTimeMillis`. We assume an environment like Amazon's elastic MapReduce, where the only interface to the system is the MapReduce programming interface and untrusted mappers are the only untrusted code on the system. Untrusted mappers cannot create files, so they cannot use file metadata to measure time. Airavat eliminates the API through which programs are notified about garbage collection (GC), so untrusted code has only indirect evidence about GC through the execution of finalizers, weak, soft, and phantom references (no Java native interface calls are allowed). Channels related to GC are inherently noisy and are controlled by trusted software whose implementation can be changed if it is found to leak too much timing information.

Airavat does not block timing channels caused by infinite loops (non-termination). Such channels have low bandwidth, leaking one bit per execution. Cloud providers send their users billing information (including execution time) which may be exploited as a timing channel. Quantizing billing units (*e.g.*, billing in multiples of $10) and aggregating billing over long time periods (*e.g.*, monthly) greatly reduce the data rate of this channel. A computer system cannot completely close all time-based channels, but a batch-oriented system like MapReduce where mappers may not access the network can decrease the utility of timing channels for the attacker to a point where another attack vector would appear preferable.

7 Implementation

The Airavat implementation includes modifications to the Hadoop MapReduce framework and Hadoop file system (HDFS), a custom JVM for running user-supplied mappers, trusted reducers, and an SELinux policy file. In our prototype, we modified $2,000$ lines of code in the MapReduce framework, $3,000$ lines in HDFS, and 500 lines of code in the JVM. The SELinux policy is approximately 450 lines that include the type enforcement

rules and interface declarations. This section describes the changes to the HDFS, implementation details of the range enforcers, and JVM modifications.

7.1 HDFS modifications

An HDFS cluster consists of a single *NameNode* server that manages the file system namespace and a number of *DataNode* servers that store file contents. HDFS currently supports file and directory permissions that are similar to the discretionary access control of the POSIX model. Airavat modifies HDFS to support MAC labels, by placing them in the file `inode` structure. Inodes are stored in the NameNode server. Any request for a file operation by a client is validated against the inode label. In the DataNodes, Airavat adds the HDFS label of the file to the block information structure.

7.2 Enforcing sensitivity

As described in Section 5.1, each mapper has an associated range enforcer. The range enforcer determines the group for each input record and tags the output produced by the mapper with the `gid`. In the degenerate case when each input belongs to a group of its own, each output by the mapper is given a unique identifier as its `gid`. The range enforcer also determines and tags the outputs with the partition identifier, `pid`. The default is to tag each record as belonging to the same partition.

During the reduce phase, each reducer fetches the sorted key/value pairs produced by the mappers. The reducer then uses the `gid` tag to group together the output values. Any value that falls outside the range declared by the computation provider ($M_{min} \dots M_{max}$) is replaced by a value inside the range. Such a substitution (if it happens) prioritizes privacy over accuracy (§ 5.1). The reducer also enforces that only key/value pairs with the correct `pid` are combined to generate the final output.

7.3 Ensuring mapper independence

To add the proper amount of noise to ensure differential privacy, the result of the mapper on each input record must not depend on any other input record (§ 5.3). A mapper is stateful if it writes a value to storage during an invocation and then uses this value in a later invocation. Airavat ensures that mapper invocations are not stateful by executing them in an untrusted domain that cannot write to files or the network. The MAC OS enforces the limitation that mappers cannot write to system resources.

For memory objects, Airavat adds access checks to two types of data: *objects*, which reside on the heap, and *statics*, which reside in the global pool. Airavat modifies the Java virtual machine to enforce these checks. Our prototype uses Jikes RVM 3.0.0,[2] a Java-in-Java research virtual machine.

[2] www.jikesrvm.org

Airavat prevents mappers from writing static variables. This restriction is enforced dynamically by using write barriers that are inserted whenever a static is accessed. Airavat modifies the object allocator to add a word to each object header. This word points to a 64-bit number called the `invocation number (ivn)`. The Airavat JVM inserts read and write barriers for all objects. Before each write, the `ivn` of the object is updated to the current invocation number (which is maintained by the trusted framework). Before a read, the JVM checks if the object's `ivn` is less than the current invocation number. If so, then the mapper is assumed to be stateful and the JVM throws an exception. After this exception, the current map invocation is re-executed and the final output of the MapReduce operation is not differentially private and must be protected using MAC (without declassification).

Jikes RVM is not mature enough to run code as large and complex as the Hadoop framework. We therefore use Hadoop's streaming feature to ensure that mappers run on Jikes and that most of the framework executes on Sun's JVM. The streaming utility forks a trusted Jikes process that loads the mapper using reflection. The Jikes process then executes the map function for each input provided by the streaming utility. The streaming utility communicates with the Jikes process using pipes. This communication is secured by SELinux.

8 Evaluation

This section empirically makes the case that Airavat can be used to efficiently compute a wide variety of algorithms in a privacy-preserving manner with acceptable accuracy loss. Table 3 provides an overview of the case studies. Our experiments show that computations in Airavat incur approximately 32% overhead compared to those running on unmodified Hadoop and Linux. In all experiments except the one with the AOL queries, the mappers are untrusted. The AOL experiment outputs keys, so we trust the mapper not to encode information in the key.

8.1 Airavat overheads

We ran all experiments on Amazon's EC2 service on a cluster of 100 machines. We use the large EC2 instances, each with two cores of 1.0–1.2 GHz Opteron or Xeon, 7.5 GB memory, 850 GB hard disk, and running SELinux-enabled Fedora 8. The numbers reported are the average of 5 runs, and the variance is less than 8%. K-Means and Naive Bayes use the public implementations from Apache Mahout.[3]

Figure 4 breaks down the execution time for each benchmark. The values are normalized to the execution time of the applications running on unmodified Hadoop and unmodified Linux. The graph depicts the percentage

[3] http://lucene.apache.org/mahout/

Benchmark	Privacy grouping	Reducer primitive	#MapReduce computations	Accuracy metric
AOL queries	Users	`THRESHOLD, SUM`	Multiple	% Queries released
kNN recommender	Individual rating	`COUNT, SUM`	Multiple	RMSE
k-Means	Individual points	`COUNT, SUM`	Multiple, till convergence	Intra-cluster variance
Naive Bayes	Individual articles	`SUM`	Multiple	Misclassification rate

Table 3: Details of the benchmarks, including the grouping of data, type of reducer used, number of MapReduce phases, and the accuracy metric.

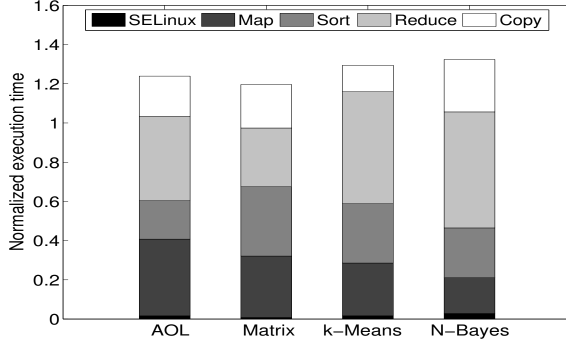


Figure 4: Normalized execution time of benchmarks when running on Airavat, compared to execution on Hadoop. Lower is better.

Benchmark	JVM overhead	Total overhead	Time (sec)
AOL	36.3%	23.9%	228 ±3
Cov. Matrix	43.2%	19.6%	1080 ±6
k-Means	28.5%	29.4%	154 ±7
Naive Bayes	37.4%	32.3%	94 ±2

Table 4: Performance details.

of the total time spent in different phases, such as map, sort, and reduce. The category *Copy* represents the phase where the output data from the mappers is copied by the reducer. Note that the copy phase generally overlaps with the map phase. The benchmarks show that Airavat slows down the computation by less than 33%.

Table 4 measures the performance overhead of enforcing differential privacy. The JVM instrumentation, to ensure mapper independence, adds up to 44% overhead in the map phase.

8.2 Queries on AOL dataset

Recently, Korolova *et al.* showed how to release search queries while preserving privacy [28]. They first find the frequency of each query and then output the noisy count of those that exceed a certain threshold. Intuitively, the threshold suppresses uncommon, low-frequency queries, since such queries are likely to breach privacy.

We demonstrate how Airavat can perform similar computations on the AOL dataset, while ensuring differential privacy. Airavat does not output non-numeric values if the mapper is untrusted because non-numeric values can leak information (§5). The outputs of this experiment are search queries (which are non-numeric) and their frequencies, so we assume that the mapper is trusted. We use `SUM` and `THRESHOLD` as reducers to generate the frequency of distinct queries and then output those that exceed the threshold. The privacy group is the *user*, and M is the maximum number of search queries made by any single user. The mapper range is $(0, M)$. We vary M in our experiments.

Our experiments use the AOL data for the first week of April 2006 (253K queries). Since we use the threshold function, Airavat needs a non-zero δ as input. We chose $\delta = 10^{-5}$ based on the number of unique users for this week, 24,861. Fixing the value of ϵ and δ also determines the minimum threshold to ensure privacy. The exact threshold value can be calculated from the formula in section 5.5: $C = M(1 - \frac{ln(\frac{2\delta}{M})}{\epsilon})$.

It is possible that a single user may perform an uncommon search multiple times (*e.g.*, if he searches for his name or address). Releasing such search queries can compromise the user's privacy. The probability of such a release can be reduced by increasing M and/or setting a low value of δ. A large value of M implies that the release threshold C is also large, thus reducing the chance that an uncommon query will be released.

In our experiments, we show the effect of different parameters on the number of queries that get published. First, we vary M, the maximum number of search queries that belong to any one user. Figure 5(a) shows that as we increase the value of M, the threshold value also increases, resulting in a smaller number of distinct queries being released. Second, we vary the privacy parameter ϵ. As we increase ϵ, *i.e.*, decrease the privacy restrictions, more queries can be released. Note that fewer than 1% of total unique queries (109K) are released. The reason is that most queries are issued very few times and hence cannot be released without jeopardizing the privacy of users who issued them.

8.3 Covariance matrices

Covariance matrices find use in many machine-learning computations. For example, McSherry and Mironov recently showed how to build a recommender system that preserves individual privacy [39]. The main idea is to construct a covariance matrix in a privacy-preserving fashion and then use a recommender algorithm such as

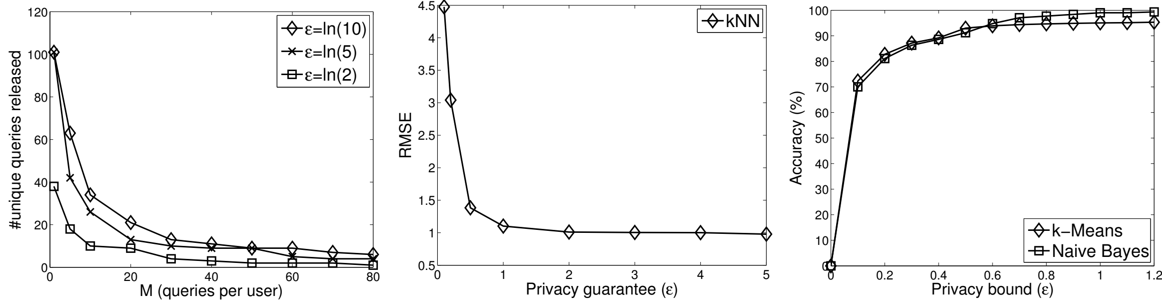


Figure 5: Effect of privacy parameter on the (a) number of released AOL search queries, (b) accuracy in RMSE in kNN recommender system (lower is better) and (c) accuracy in k-Means and Naive Bayes (higher is better).

k-nearest neighbor (kNN) on the matrix.

We picked $1,000$ movies from the Netflix prize dataset and generated a covariance matrix using Airavat. The computation protects the privacy of any individual Netflix user. We cannot calculate the complete matrix in one computation using the Airavat primitives. Instead, we fill the matrix cell by individual cell. The disadvantage of this approach is that the privacy budget is expended very quickly. For example, if the matrix has M^2 cells, then we subtract ϵM^2 from the privacy budget (equivalently, we achieve ϵM^2-differential privacy).

Because each movie rating is between 1 and 5 and an entry of the covariance matrix is a product of two such ratings, the mapper range is $(0, 25)$. Figure 5(b) plots the root mean squared error (RMSE) of the kNN algorithm when executed on the covariance matrix generated by Airavat. The x-axis corresponds to the privacy guarantee for the complete covariance matrix. Our results show that with the guarantee of 5-differential privacy, the RMSE of kNN is approximately 0.97. For comparison, Netflix's own algorithm, called Cinematch, has a RMSE of 0.95 when applied on the complete Netflix dataset.

8.4 Clustering Algorithm: k-Means

The k-Means algorithm clusters input vectors into k partitions. The partitioning aims to minimize intra-cluster variances. We use Lloyd's iterative heuristic to compute k-Means. The algorithm proceeds in two steps [6]. In the first step, the cardinality of each cluster is calculated. In the second step, all points in the new cluster are added up and then divided by the cardinality derived in the previous step, producing new cluster centers. The input dataset consists of 600 examples of control charts.[4] Control charts are used to assess whether a process is functioning properly. Machine learning techniques are often applied to such charts to detect anomaly patterns.

Figure 5(c) plots the accuracy of the k-Means algorithm as we change the privacy parameter ϵ. We assume that each point belongs to a different user whose privacy must be guaranteed. The mapper range of the computation that calculates the cluster size is $(0, 1)$. The mapper range for calculating the actual cluster centers is bounded by the maximum value of any coordinate over all points, which is 36 for the current dataset. We measure the accuracy of the algorithm by computing the intra-cluster variance. With $\epsilon > 0.5$, the accuracy of the clustering algorithm exceeds 90%.

8.5 Classification algorithm: Naive Bayes

Naive Bayes is a simple probabilistic classifier that applies the Bayes Theorem with assumptions of strong independence. During the training phase, the algorithm is given a set of feature vectors and the class labels to which they belong. The algorithm creates a model, which is then used in the classification phase to classify previously unseen vectors.

Figure 5(c) plots the accuracy against the privacy parameter ϵ. We used the 20newsgroup dataset,[5] which consists of different articles represented by words that appear in them. We train the classifier on one partition of the dataset and test it on another. The value of ϵ affects the noise which is added to the model in the training phase. We measure the accuracy of the classifier by looking at the number of misclassified articles. An article contributes at most $1,000$ to a category of words, so the range for mapper outputs is $(0, 1000)$. Our results show that, for this particular dataset, we require $\epsilon > 0.6$ to achieve 95% accuracy.

9 Related work

Differential privacy guarantees are somewhat similar to robust or secure statistical estimation, which provides statistical computations with low sensitivity to any single input (*e.g.*, see [21, 23, 24]). While robust estimators do not by themselves guarantee privacy, they can serve as the basis for differentially private estimators [16].

[4] http://archive.ics.uci.edu/ml/databases/synthetic_control

[5] http://people.csail.mit.edu/jrennie/20Newsgroups/

In its current version, Airavat requires computation providers to provide an upper bound on the sensitivity of their code by declaring the range of its possible outputs in advance. An alternative is to have the enforcement system estimate local, input-specific sensitivity of the function computed by the code—either by re-running it on perturbed inputs, or by sampling from the input space [43]. Local sensitivity measures how much the output of the function varies on neighboring inputs from a subset of the function's domain. It often requires less noise to be added to the output in order to achieve the same differential privacy guarantee. We plan to investigate this approach in future work.

PINQ. Privacy Integrated Queries (PINQ) is a declarative system for computing on sensitive data [38] which ensures differential privacy for the outputs of the computation. Airavat mappers are Java bytecode, with restrictions on the programming model enforced at runtime. Mapper independence is an example of a restriction enforced by the language runtime which is absent from PINQ. PINQ provides a restricted programming language with a small number of trusted, primitive data operations in the LINQ framework. PINQ employs a request/reply model, which avoids adding noise to the intermediate results of the computation by keeping them on a trusted data server or an abstraction of a trusted data server provided by a distributed system.

Airavat's privacy enforcement mechanisms provide end-to-end guarantees, while PINQ provides language-level guarantees. Airavat's enforcement mechanisms include all software in the MapReduce framework, including language runtimes, the distributed file system, and the operating system. Enforcing privacy throughout the software stack allows Airavat computations to be securely distributed across multiple nodes, achieving the scalability that is the hallmark of the MapReduce framework. While the PINQ API can be supported in a similar setting (*e.g.*, DryadLINQ), PINQ's security would then depend on the security of Microsoft's common language runtime (CLR), the Cosmos distributed file system, the Dryad framework, and the operating system. Securing the levels below the language layer would require the same security guarantees as provided by Airavat.

Alternative definitions of privacy. Differential privacy is a relative notion: it assures the owner of any individual data item that the same privacy violations, if any, will occur whether this item is included in the aggregate computation or not. Therefore, no additional privacy risk arises from participating in the computation. While this may seem like a relatively weak guarantee, stronger properties *cannot* be achieved without making unjustified assumptions about the adversary [11, 12]. Superficially plausible but unachievable definitions include "the adversary does not learn anything about the data that he did not know before" [8] and "the adversary's posterior distribution of possible data values after observing the result of the computation is close to his prior distribution."

Secure multi-party computation [20] ensures that a distributed protocol leaks no more information about the inputs than is revealed by the output of the computation. The goal is to keep the intermediate steps of the computation secret. This technique is not appropriate in our setting, where the goal is to ensure that the output itself does not leak too much information about the inputs.

While differential privacy mechanisms often employ output perturbation (adding random noise to the result of a computation), several approaches to privacy-preserving data mining add random noise to inputs instead. Privacy guarantees are usually average-case and do not imply anything about the privacy of individual inputs. For example, the algorithm of Agrawal and Srikant [4] fails to hide individual inputs [3]. In turn, Evfimievski *et al.* show that the definitions of [3] are too weak to provide individual privacy [18].

k-anonymity focuses on non-interactive releases of relational data and requires that every record in the released dataset be syntactically indistinguishable from at least $k-1$ other records on the so-called quasi-identifying attributes, such as ZIP code and date of birth [7, 48]. k-anonymity is achieved by syntactic generalization and suppression of these attributes (*e.g.*, [31]). k-anonymity does not provide meaningful privacy guarantees. It fundamentally assumes that the adversary's knowledge is limited to the quasi-identifying attributes and thus fails to provide any protection against adversaries who have additional information [34, 35]. It does not hide whether a particular individual is in the dataset [42], nor the sensitive attributes associated with any individual [32, 34]. Multiple releases of the same dataset or mere knowledge of the k-anonymization algorithm may completely break the protection [19, 52]. Variants, such as l-diversity [34] and m-invariance [50], suffer from many of the same flaws.

Program analysis techniques can be used to estimate how much information is leaked by a program [36]. Privacy in MapReduce computations, however, is difficult if not impossible to express as a quantitative information flow problem. The flow bound cannot be set at 0 bits because the output depends on every single input. But even a 1-bit leakage may be sufficient to reveal, for example, whether a given person's record was present in the input dataset or not, violating privacy. By contrast, differential privacy guarantees that the information revealed by the computation cannot be specific to any given input.

10 Conclusion

Airavat is the first system that integrates mandatory access control with differential privacy, enabling many privacy-preserving MapReduce computations without the need to audit untrusted code. We demonstrate the practicality of Airavat by evaluating it on a variety of case studies.

Acknowledgements

We are grateful to Frank McSherry for several insightful discussions and for helping us understand PINQ. We thank the anonymous referees and our shepherd Michael J. Freedman for the constructive feedback. We also thank Hany Ramadan for his help with the initial implementation. This research is supported by NSF grants CNS-0746888 and CNS-0905602, "Collaborative Policies and Assured Information Sharing" MURI, and a Google research award.

References

[1] *AppArmor*. `https://help.ubuntu.com/8.04/serverguide/C/apparmor.html`.

[2] *Hadoop*. `http://hadoop.apache.org/core/`.

[3] D. Agrawal and C. Aggarwal. On the design and quantification of privacy-preserving data mining algorithms. In *PODS*, 2001.

[4] R. Agrawal and R. Srikant. Privacy-preserving data mining. In *SIGMOD*, 2000.

[5] B. Barak, K. Chaudhuri, C. Dwork, S. Kale, F. McSherry, and K. Talwar. Privacy, accuracy, and consistency too: a holistic solution to contingency table release. In *PODS*, 2007.

[6] A. Blum, C. Dwork, F. McSherry, and K. Nissim. Practical privacy: the SuLQ framework. In *PODS*, 2005.

[7] V. Ciriani, S. De Capitani di Vimercati, S. Foresti, and P. Samarati. k-anonymity. *Secure Data Management in Decentralized Systems*, 2007.

[8] T. Dalenius. Towards a methodology for statistical disclosure control. *Statistik Tidskrift*, 15, 1977.

[9] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1), 2008.

[10] I. Dinur and K. Nissim. Revealing information while preserving privacy. In *PODS*, 2003.

[11] C. Dwork. Differential privacy. In *ICALP*, 2006.

[12] C. Dwork. An ad omnia approach to defining and achieving private data analysis. In *PinKDD*, 2007.

[13] C. Dwork. Ask a better question, get a better answer: A new approach to private data analysis. In *ICDT*, 2007.

[14] C. Dwork. Differential privacy: A survey of results. In *TAMC*, 2008.

[15] C. Dwork, K. Kenthapadi, F. McSherry, I. Mironov, and M. Naor. Our data, ourselves: Privacy via distributed noise generation. In *EUROCRYPT*, 2006.

[16] C. Dwork and J. Lei. Differential privacy and robust statistics. In *STOC*, 2009.

[17] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *TCC*, 2006.

[18] A. Evfimievski, J. Gehrke, and R. Srikant. Limiting privacy breaches in privacy-preserving data mining. In *PODS*, 2003.

[19] S. Ganta, S. Kasiviswanathan, and A. Smith. Composition attacks and auxiliary information in data privacy. In *KDD*, 2008.

[20] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *STOC*, 1987.

[21] F. Hampel, E. Ronchetti, P. Rousseeuw, and W. Stahel. *Robust Statistics - The Approach Based on Influence Functions*. Wiley, 1986.

[22] S. Hansell. AOL removes search data on vast group of web users. New York Times, Aug 8 2006.

[23] J. Heitzig. The "jackknife" method: confidentiality protection for complex statistical analyses. Joint UNECE/Eurostat work session on statistical data confidentiality, 2005.

[24] J. Hellerstein. Quantitative data cleaning for large databases. `http://db.cs.berkeley.edu/jmh/papers/cleaning-unece.pdf`, February 2008.

[25] W-M. Hu. Reducing timing channels with fuzzy time. In *S&P*, 1987.

[26] P. Karger, M.E. Zurko, D. Bonin, A. Mason, and C. Kahn. A retrospective on the VAX VMM security kernel. *IEEE Trans. Softw. Eng.*, 17(11), 1991.

[27] M. Kearns. Efficient noise-tolerant learning from statistical queries. *J. ACM*, 45(6), 1998.

[28] A. Korolova, K. Kenthapadi, N. Mishra, and A. Ntoulas. Releasing search queries and clicks privately. In *WWW*, 2009.

[29] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *SOSP*, 2007.

[30] B. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10), 1973.

[31] K. LeFevre, D. DeWitt, and R. Ramakrishnan. Incognito: Efficient full-domain k-anonymity. In *SIGMOD*, 2005.

[32] N. Li, T. Li, and S. Venkatasubramanian. t-closeness: Privacy beyond k-anonymity and ℓ-diversity. In *ICDE*, 2007.

[33] S. Lipner. A comment on the confinement problem. In *SOSP*, 1975.

[34] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam. ℓ-diversity: Privacy beyond k-anonymity. In *ICDE*, 2006.

[35] D. Martin, D. Kifer, A. Machanavajjhala, J. Gehrke, and J. Halpern. Worst-case background knowledge for privacy-preserving data publishing. In *ICDE*, 2007.

[36] S. McCamant and M. Ernst. Quantitative information flow as network flow capacity. In *PLDI*, 2008.

[37] B. McCarty. *SELinux: NSA's Open Source Security Enhanced Linux*. O'Reilly Media, 2004.

[38] F. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *SIGMOD*, 2009.

[39] F. McSherry and I. Mironov. Differentially private recommender systems: Building privacy into the Netflix Prize contenders. In *KDD*, 2009.

[40] F. McSherry and K. Talwar. Mechanism design via differential privacy. In *FOCS*, 2007.

[41] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. In *S&P*, 2008.

[42] M. Nergiz, M. Atzori, and C. Clifton. Hiding the presence of individuals from shared database. In *SIGMOD*, 2007.

[43] K. Nissim, S. Raskhodnikova, and A. Smith. Smooth sensitivity and sampling in private data analysis. In *STOC*, 2007.

[44] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud! Exploring information leakage in third-party compute clouds. In *CCS*, 2009.

[45] I. Roy, D. Porter, M. Bond, K. Mckinley, and E. Witchel. Laminar: Practical fine-grained decentralized information flow control. In *PLDI*, 2009.

[46] I. Roy, S. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and privacy for MapReduce. Technical Report TR-10-09, UT-Austin, 2010.

[47] M. Schatz. CloudBurst: highly sensitive read mapping with MapReduce. *Bioinformatics*, 2009.

[48] L. Sweeney. k-anonymity: A model for protecting privacy. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 10(5), 2002.

[49] S. Vandebogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazières. Labels and event processes in the Asbestos operating system. *TOCS*, 2007.

[50] X. Xiao and T. Tao. m-invariance: Towards privacy preserving re-publication of dynamic datasets. In *SIGMOD*, 2007.

[51] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI*, 2006.

[52] L. Zhang, S. Jajodia, and A. Brodsky. Information disclosure under realistic assumptions: Privacy versus optimality. In *CCS*, 2007.

MapReduce Online

Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein
UC Berkeley

Khaled Elmeleegy, Russell Sears
Yahoo! Research

Abstract

MapReduce is a popular framework for data-intensive distributed computing of batch jobs. To simplify fault tolerance, many implementations of MapReduce *materialize* the entire output of each map and reduce task before it can be consumed. In this paper, we propose a modified MapReduce architecture that allows data to be *pipelined* between operators. This extends the MapReduce programming model beyond batch processing, and can reduce completion times and improve system utilization for batch jobs as well. We present a modified version of the Hadoop MapReduce framework that supports *online aggregation*, which allows users to see "early returns" from a job as it is being computed. Our Hadoop Online Prototype (*HOP*) also supports *continuous queries*, which enable MapReduce programs to be written for applications such as event monitoring and stream processing. HOP retains the fault tolerance properties of Hadoop and can run unmodified user-defined MapReduce programs.

1 Introduction

MapReduce has emerged as a popular way to harness the power of large clusters of computers. MapReduce allows programmers to think in a *data-centric* fashion: they focus on applying transformations to sets of data records, and allow the details of distributed execution, network communication and fault tolerance to be handled by the MapReduce framework.

MapReduce is typically applied to large batch-oriented computations that are concerned primarily with time to job completion. The Google MapReduce framework [6] and open-source Hadoop system reinforce this usage model through a batch-processing implementation strategy: the entire output of each map and reduce task is *materialized* to a local file before it can be consumed by the next stage. Materialization allows for a simple and elegant checkpoint/restart fault tolerance mechanism that is critical in large deployments, which have a high probability of slowdowns or failures at worker nodes.

We propose a modified MapReduce architecture in which intermediate data is *pipelined* between operators, while preserving the programming interfaces and fault tolerance models of previous MapReduce frameworks. To validate this design, we developed the Hadoop Online Prototype (HOP), a pipelining version of Hadoop.[1]

Pipelining provides several important advantages to a MapReduce framework, but also raises new design challenges. We highlight the potential benefits first:

- Since reducers begin processing data as soon as it is produced by mappers, they can generate and refine an approximation of their final answer during the course of execution. This technique, known as *online aggregation* [12], can provide initial estimates of results several orders of magnitude faster than the final results. We describe how we adapted online aggregation to our pipelined MapReduce architecture in Section 4.

- Pipelining widens the domain of problems to which MapReduce can be applied. In Section 5, we show how HOP can be used to support *continuous queries*: MapReduce jobs that run continuously, accepting new data as it arrives and analyzing it immediately. This allows MapReduce to be used for applications such as event monitoring and stream processing.

- Pipelining delivers data to downstream operators more promptly, which can increase opportunities for parallelism, improve utilization, and reduce response time. A thorough performance study is a topic for future work; however, in Section 6 we present some initial performance results which demonstrate that pipelining can reduce job completion times by up to 25% in some scenarios.

[1]The source code for HOP can be downloaded from `http://code.google.com/p/hop/`

Pipelining raises several design challenges. First, Google's attractively simple MapReduce fault tolerance mechanism is predicated on the materialization of intermediate state. In Section 3.3, we show that this can coexist with pipelining, by allowing producers to periodically ship data to consumers in parallel with their materialization. A second challenge arises from the greedy communication implicit in pipelines, which is at odds with batch-oriented optimizations supported by "combiners": map-side code that reduces network utilization by performing pre-aggregation before communication. We discuss how the HOP design addresses this issue in Section 3.1. Finally, pipelining requires that producers and consumers are co-scheduled intelligently; we discuss our initial work on this issue in Section 3.4.

1.1 Structure of the Paper

In order to ground our discussion, we present an overview of the Hadoop MapReduce architecture in Section 2. We then develop the design of HOP's pipelining scheme in Section 3, keeping the focus on traditional batch processing tasks. In Section 4 we show how HOP can support online aggregation for long-running jobs and illustrate the potential benefits of that interface for MapReduce tasks. In Section 5 we describe our support for continuous MapReduce jobs over data streams and demonstrate an example of near-real-time cluster monitoring. We present initial performance results in Section 6. Related and future work are covered in Sections 7 and 8.

2 Background

In this section, we review the MapReduce programming model and describe the salient features of Hadoop, a popular open-source implementation of MapReduce.

2.1 Programming Model

To use MapReduce, the programmer expresses their desired computation as a series of *jobs*. The input to a job is an input specification that will yield key-value pairs. Each job consists of two stages: first, a user-defined *map* function is applied to each input record to produce a list of intermediate key-value pairs. Second, a user-defined *reduce* function is called once for each distinct key in the map output and passed the list of intermediate values associated with that key. The MapReduce framework automatically parallelizes the execution of these functions and ensures fault tolerance.

Optionally, the user can supply a *combiner* function [6]. Combiners are similar to reduce functions, except that they are not passed *all* the values for a given key: instead, a combiner emits an output value that summarizes the input values it was passed. Combiners are typically used to perform map-side "pre-aggregation," which reduces the amount of network traffic required between the map and reduce steps.

```
public interface Mapper<K1, V1, K2, V2> {
  void map(K1 key, V1 value,
           OutputCollector<K2, V2> output);

  void close();
}
```

Figure 1: Map function interface.

2.2 Hadoop Architecture

Hadoop is composed of *Hadoop MapReduce*, an implementation of MapReduce designed for large clusters, and the *Hadoop Distributed File System* (HDFS), a file system optimized for batch-oriented workloads such as MapReduce. In most Hadoop jobs, HDFS is used to store both the input to the map step and the output of the reduce step. Note that HDFS is *not* used to store intermediate results (e.g., the output of the map step): these are kept on each node's local file system.

A Hadoop installation consists of a single master node and many worker nodes. The master, called the *JobTracker*, is responsible for accepting jobs from clients, dividing those jobs into *tasks*, and assigning those tasks to be executed by worker nodes. Each worker runs a *TaskTracker* process that manages the execution of the tasks currently assigned to that node. Each TaskTracker has a fixed number of slots for executing tasks (two maps and two reduces by default).

2.3 Map Task Execution

Each map task is assigned a portion of the input file called a *split*. By default, a split contains a single HDFS block (64MB by default), so the total number of file blocks determines the number of map tasks.

The execution of a map task is divided into two phases.

1. The *map* phase reads the task's split from HDFS, parses it into records (key/value pairs), and applies the map function to each record.

2. After the map function has been applied to each input record, the *commit* phase registers the final output with the TaskTracker, which then informs the JobTracker that the task has finished executing.

Figure 1 contains the interface that must be implemented by user-defined map functions. After the *map* function has been applied to each record in the split, the *close* method is invoked.

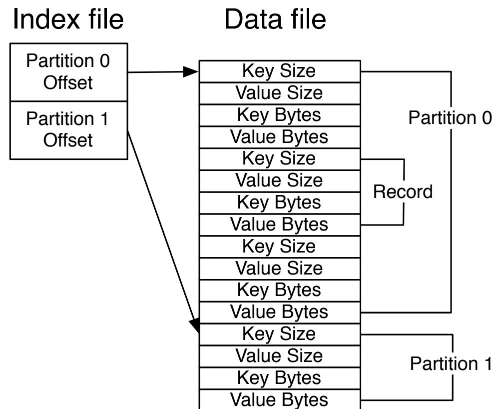


Figure 2: Map task index and data file format (2 partition/reduce case).

The third argument to the *map* method specifies an *OutputCollector* instance, which accumulates the output records produced by the map function. The output of the map step is consumed by the reduce step, so the OutputCollector stores map output in a format that is easy for reduce tasks to consume. Intermediate keys are assigned to reducers by applying a partitioning function, so the OutputCollector applies that function to each key produced by the map function, and stores each record and partition number in an in-memory buffer. The OutputCollector spills this buffer to disk when it reaches capacity.

A spill of the in-memory buffer involves first sorting the records in the buffer by partition number and then by key. The buffer content is written to the local file system as an index file and a data file (Figure 2). The index file points to the offset of each partition in the data file. The data file contains only the records, which are sorted by the key within each partition segment.

During the *commit* phase, the final output of the map task is generated by merging all the spill files produced by this task into a single pair of data and index files. These files are registered with the TaskTracker before the task completes. The TaskTracker will read these files when servicing requests from reduce tasks.

2.4 Reduce Task Execution

The execution of a reduce task is divided into three phases.

1. The *shuffle* phase fetches the reduce task's input data. Each reduce task is assigned a partition of the key range produced by the map step, so the reduce task must fetch the content of this partition from every map task's output.

2. The *sort* phase groups records with the same key together.

```
public interface Reducer<K2, V2, K3, V3> {
  void reduce(K2 key, Iterator<V2> values,
              OutputCollector<K3, V3> output);

  void close();
}
```

Figure 3: Reduce function interface.

3. The *reduce* phase applies the user-defined reduce function to each key and corresponding list of values.

In the *shuffle* phase, a reduce task fetches data from each map task by issuing HTTP requests to a configurable number of TaskTrackers at once (5 by default). The JobTracker relays the location of every TaskTracker that hosts map output to every TaskTracker that is executing a reduce task. Note that a reduce task cannot fetch the output of a map task until the map has finished executing and committed its final output to disk.

After receiving its partition from all map outputs, the reduce task enters the *sort* phase. The map output for each partition is already sorted by the reduce key. The reduce task merges these runs together to produce a single run that is sorted by key. The task then enters the *reduce* phase, in which it invokes the user-defined reduce function for each distinct key in sorted order, passing it the associated list of values. The output of the reduce function is written to a temporary location on HDFS. After the reduce function has been applied to each key in the reduce task's partition, the task's HDFS output file is atomically renamed from its temporary location to its final location.

In this design, the output of both map and reduce tasks is written to disk before it can be consumed. This is particularly expensive for reduce tasks, because their output is written to HDFS. Output materialization simplifies fault tolerance, because it reduces the amount of state that must be restored to consistency after a node failure. If any task (either map or reduce) fails, the JobTracker simply schedules a new task to perform the same work as the failed task. Since a task never exports any data other than its final answer, no further recovery steps are needed.

3 Pipelined MapReduce

In this section we discuss our extensions to Hadoop to support pipelining between tasks (Section 3.1) and between jobs (Section 3.2). We describe how our design supports fault tolerance (Section 3.3), and discuss the interaction between pipelining and task scheduling (Section 3.4). Our focus here is on batch-processing workloads; we discuss online aggregation and continuous queries in Section 4 and Section 5. We defer performance results to Section 6.

3.1 Pipelining Within A Job

As described in Section 2.4, reduce tasks traditionally issue HTTP requests to *pull* their output from each TaskTracker. This means that map task execution is completely decoupled from reduce task execution. To support pipelining, we modified the map task to instead *push* data to reducers as it is produced. To give an intuition for how this works, we begin by describing a straightforward pipelining design, and then discuss the changes we had to make to achieve good performance.

3.1.1 Naïve Pipelining

In our naïve implementation, we modified Hadoop to send data directly from map to reduce tasks. When a client submits a new job to Hadoop, the JobTracker assigns the map and reduce tasks associated with the job to the available TaskTracker slots. For purposes of discussion, we assume that there are enough free slots to assign all the tasks for each job. We modified Hadoop so that each reduce task contacts every map task upon initiation of the job, and opens a TCP socket which will be used to pipeline the output of the map function. As each map output record is produced, the mapper determines which partition (reduce task) the record should be sent to, and immediately sends it via the appropriate socket.

A reduce task accepts the pipelined data it receives from each map task and stores it in an in-memory buffer, spilling sorted runs of the buffer to disk as needed. Once the reduce task learns that every map task has completed, it performs a final merge of all the sorted runs and applies the user-defined reduce function as normal.

3.1.2 Refinements

While the algorithm described above is straightforward, it suffers from several practical problems. First, it is possible that there will not be enough slots available to schedule every task in a new job. Opening a socket between every map and reduce task also requires a large number of TCP connections. A simple tweak to the naïve design solves both problems: if a reduce task has not yet been scheduled, any map tasks that produce records for that partition simply write them to disk. Once the reduce task is assigned a slot, it can then pull the records from the map task, as in regular Hadoop. To reduce the number of concurrent TCP connections, each reducer can be configured to pipeline data from a bounded number of mappers at once; the reducer will pull data from the remaining map tasks in the traditional Hadoop manner.

Our initial pipelining implementation suffered from a second problem: the map function was invoked by the same thread that wrote output records to the pipeline sockets. This meant that if a network I/O operation blocked (e.g., because the reducer was over-utilized), the mapper was prevented from doing useful work. Pipeline stalls should not prevent a map task from making progress—especially since, once a task has completed, it frees a TaskTracker slot to be used for other purposes. We solved this problem by running the map function in a separate thread that stores its output in an in-memory buffer, and then having another thread periodically send the contents of the buffer to the connected reducers.

3.1.3 Granularity of Map Output

Another problem with the naïve design is that it eagerly sends each record as soon as it is produced, which prevents the use of map-side combiners. Imagine a job where the reduce key has few distinct values (e.g., gender), and the reduce applies an aggregate function (e.g., count). As discussed in Section 2.1, combiners allow map-side "pre-aggregation": by applying a reduce-like function to each distinct key at the mapper, network traffic can often be substantially reduced. Eagerly pipelining each record as it is produced prevents the use of map-side combiners.

A related problem is that eager pipelining moves some of the sorting work from the mapper to the reducer. Recall that in the blocking architecture, map tasks generate sorted spill files: all the reduce task must do is merge together the pre-sorted map output for each partition. In the naïve pipelining design, map tasks send output records in the order in which they are generated, so the reducer must perform a full external sort. Because the number of map tasks typically far exceeds the number of reduces [6], moving more work to the reducer increased response time in our experiments.

We addressed these issues by modifying the in-memory buffer design described in Section 3.1.2. Instead of sending the buffer contents to reducers directly, we wait for the buffer to grow to a threshold size. The mapper then applies the combiner function, sorts the output by partition and reduce key, and writes the buffer to disk using the spill file format described in Section 2.3.

Next, we arranged for the TaskTracker at each node to handle pipelining data to reduce tasks. Map tasks register spill files with the TaskTracker via RPCs. If the reducers are able to keep up with the production of map outputs and the network is not a bottleneck, a spill file will be sent to a reducer soon after it has been produced (in which case, the spill file is likely still resident in the map machine's kernel buffer cache). However, if a reducer begins to fall behind, the number of unsent spill files will grow.

When a map task generates a new spill file, it first queries the TaskTracker for the number of unsent spill files. If this number grows beyond a certain threshold (two unsent spill files in our experiments), the map task does not immediately register the new spill file with the

TaskTracker. Instead, the mapper will accumulate multiple spill files. Once the queue of unsent spill files falls below the threshold, the map task merges and combines the accumulated spill files into a single file, and then resumes registering its output with the TaskTracker. This simple flow control mechanism has the effect of *adaptively* moving load from the reducer to the mapper or vice versa, depending on which node is the current bottleneck.

A similar mechanism is also used to control how aggressively the combiner function is applied. The map task records the ratio between the input and output data sizes whenever it invokes the combiner function. If the combiner is effective at reducing data volumes, the map task accumulates more spill files (and applies the combiner function to all of them) before registering that output with the TaskTracker for pipelining.[2]

The connection between pipelining and adaptive query processing techniques has been observed elsewhere (e.g., [2]). The adaptive scheme outlined above is relatively simple, but we believe that adapting to feedback along pipelines has the potential to significantly improve the utilization of MapReduce clusters.

3.2 Pipelining Between Jobs

Many practical computations cannot be expressed as a single MapReduce job, and the outputs of higher-level languages like Pig [20] typically involve multiple jobs. In the traditional Hadoop architecture, the output of each job is written to HDFS in the reduce step and then immediately read back from HDFS by the map step of the next job. Furthermore, the JobTracker cannot schedule a consumer job until the producer job has completed, because scheduling a map task requires knowing the HDFS block locations of the map's input split.

In our modified version of Hadoop, the reduce tasks of one job can optionally pipeline their output directly to the map tasks of the next job, sidestepping the need for expensive fault-tolerant storage in HDFS for what amounts to a temporary file. Unfortunately, the computation of the reduce function from the previous job and the map function of the next job cannot be overlapped: the final result of the reduce step cannot be produced until all map tasks have completed, which prevents effective pipelining. However, in the next sections we describe how online aggregation and continuous query pipelines can publish "snapshot" outputs that can indeed pipeline between jobs.

[2]Our current prototype uses a simple heuristic: if the combiner reduces data volume by $\frac{1}{k}$ on average, we wait until k spill files have accumulated before registering them with the TaskTracker. A better heuristic would also account for the computational cost of applying the combiner function.

3.3 Fault Tolerance

Our pipelined Hadoop implementation is robust to the failure of both map and reduce tasks. To recover from map task failures, we added bookkeeping to the reduce task to record which map task produced each pipelined spill file. To simplify fault tolerance, the reducer treats the output of a pipelined map task as "tentative" until the JobTracker informs the reducer that the map task has committed successfully. The reducer can merge together spill files generated by the same uncommitted mapper, but will not combine those spill files with the output of other map tasks until it has been notified that the map task has committed. Thus, if a map task fails, each reduce task can ignore any tentative spill files produced by the failed map attempt. The JobTracker will take care of scheduling a new map task attempt, as in stock Hadoop.

If a reduce task fails and a new copy of the task is started, the new reduce instance must be sent all the input data that was sent to the failed reduce attempt. If map tasks operated in a purely pipelined fashion and discarded their output after sending it to a reducer, this would be difficult. Therefore, map tasks retain their output data on the local disk for the complete job duration. This allows the map's output to be reproduced if any reduce tasks fail. For batch jobs, the key advantage of our architecture is that reducers are not blocked waiting for the complete output of the task to be written to disk.

Our technique for recovering from map task failure is straightforward, but places a minor limit on the reducer's ability to merge spill files. To avoid this, we envision introducing a "checkpoint" concept: as a map task runs, it will periodically notify the JobTracker that it has reached offset x in its input split. The JobTracker will notify any connected reducers; map task output that was produced before offset x can then be merged by reducers with other map task output as normal. To avoid duplicate results, if the map task fails, the new map task attempt resumes reading its input at offset x. This technique would also reduce the amount of redundant work done after a map task failure or during speculative execution of "backup" tasks [6].

3.4 Task Scheduling

The Hadoop JobTracker had to be retrofitted to support pipelining between jobs. In regular Hadoop, job are submitted one at a time; a job that consumes the output of one or more other jobs cannot be submitted until the producer jobs have completed. To address this, we modified the Hadoop job submission interface to accept a list of jobs, where each job in the list depends on the job before it. The client interface traverses this list, annotating each job with the identifier of the job that it depends on. The

JobTracker looks for this annotation and co-schedules jobs with their dependencies, giving slot preference to "upstream" jobs over the "downstream" jobs they feed. As we note in Section 8, there are many interesting options for scheduling pipelines or even DAGs of such jobs that we plan to investigate in future.

4 Online Aggregation

Although MapReduce was originally designed as a batch-oriented system, it is often used for interactive data analysis: a user submits a job to extract information from a data set, and then waits to view the results before proceeding with the next step in the data analysis process. This trend has accelerated with the development of high-level query languages that are executed as MapReduce jobs, such as Hive [27], Pig [20], and Sawzall [23].

Traditional MapReduce implementations provide a poor interface for interactive data analysis, because they do not emit any output until the job has been executed to completion. In many cases, an interactive user would prefer a "quick and dirty" approximation over a correct answer that takes much longer to compute. In the database literature, online aggregation has been proposed to address this problem [12], but the batch-oriented nature of traditional MapReduce implementations makes these techniques difficult to apply. In this section, we show how we extended our pipelined Hadoop implementation to support online aggregation within a single job (Section 4.1) and between multiple jobs (Section 4.2). In Section 4.3, we evaluate online aggregation on two different data sets, and show that it can yield an accurate approximate answer long before the job has finished executing.

4.1 Single-Job Online Aggregation

In HOP, the data records produced by map tasks are sent to reduce tasks shortly after each record is generated. However, to produce the final output of the job, the reduce function cannot be invoked until the entire output of every map task has been produced. We can support online aggregation by simply applying the reduce function to the data that a reduce task has received so far. We call the output of such an intermediate reduce operation a *snapshot*.

Users would like to know how accurate a snapshot is: that is, how closely a snapshot resembles the final output of the job. Accuracy estimation is a hard problem even for simple SQL queries [15], and particularly hard for jobs where the map and reduce functions are opaque user-defined code. Hence, we report job *progress*, not accuracy: we leave it to the user (or their MapReduce code) to correlate progress to a formal notion of accuracy. We give a simple progress metric below.

Snapshots are computed periodically, as new data arrives at each reducer. The user specifies how often snapshots should be computed, using the progress metric as the unit of measure. For example, a user can request that a snapshot be computed when 25%, 50%, and 75% of the input has been seen. The user may also specify whether to include data from tentative (unfinished) map tasks. This option does not affect the fault tolerance design described in Section 3.3. In the current prototype, each snapshot is stored in a directory on HDFS. The name of the directory includes the progress value associated with the snapshot. Each reduce task runs independently, and at a different rate. Once a reduce task has made sufficient progress, it writes a snapshot to a temporary directory on HDFS, and then atomically renames it to the appropriate location.

Applications can consume snapshots by polling HDFS in a predictable location. An application knows that a given snapshot has been completed when every reduce task has written a file to the snapshot directory. Atomic rename is used to avoid applications mistakenly reading incomplete snapshot files.

Note that if there are not enough free slots to allow all the reduce tasks in a job to be scheduled, snapshots will not be available for reduce tasks that are still waiting to be executed. The user can detect this situation (e.g., by checking for the expected number of files in the HDFS snapshot directory), so there is no risk of incorrect data, but the usefulness of online aggregation will be reduced. In the current prototype, we manually configured the cluster to avoid this scenario. The system could also be enhanced to avoid this pitfall entirely by optionally waiting to execute an online aggregation job until there are enough reduce slots available.

4.1.1 Progress Metric

Hadoop provides support for monitoring the progress of task executions. As each map task executes, it is assigned a *progress score* in the range [0,1], based on how much of its input the map task has consumed. We reused this feature to determine how much progress is represented by the current input to a reduce task, and hence to decide when a new snapshot should be taken.

First, we modified the spill file format depicted in Figure 2 to include the map's current progress score. When a partition in a spill file is sent to a reducer, the spill file's progress score is also included. To compute the progress score for a snapshot, we take the average of the progress scores associated with each spill file used to produce the snapshot.

Note that it is possible that a map task might not have pipelined *any* output to a reduce task, either because the map task has not been scheduled yet (there are no free TaskTracker slots), the map tasks does not produce any

output for the given reduce task, or because the reduce task has been configured to only pipeline data from at most k map tasks concurrently. To account for this, we need to scale the progress metric to reflect the portion of the map tasks that a reduce task has pipelined data from: if a reducer is connected to $\frac{1}{n}$ of the total number of map tasks in the job, we divide the average progress score by n.

This progress metric could easily be made more sophisticated: for example, an improved metric might include the selectivity ($|output|/|input|$) of each map task, the statistical distribution of the map task's output, and the effectiveness of each map task's combine function, if any. Although we have found our simple progress metric to be sufficient for most experiments we describe below, this clearly represents an opportunity for future work.

4.2 Multi-Job Online Aggregation

Online aggregation is particularly useful when applied to a long-running analysis task composed of multiple MapReduce jobs. As described in Section 3.2, our version of Hadoop allows the output of a reduce task to be sent directly to map tasks. This feature can be used to support online aggregation for a sequence of jobs.

Suppose that j_1 and j_2 are two MapReduce jobs, and j_2 consumes the output of j_1. When j_1's reducers compute a snapshot to perform online aggregation, that snapshot is written to HDFS, and also sent directly to the map tasks of j_2. The map and reduce steps for j_2 are then computed as normal, to produce a snapshot of j_2's output. This process can then be continued to support online aggregation for an arbitrarily long sequence of jobs.

Unfortunately, inter-job online aggregation has some drawbacks. First, the output of a reduce function is not "monotonic": the output of a reduce function on the first 50% of the input data may not be obviously related to the output of the reduce function on the first 25%. Thus, as new snapshots are produced by j_1, j_2 must be recomputed from scratch using the new snapshot. As with inter-job pipelining (Section 3.2), this could be optimized for reduce functions that are declared to be distributive or algebraic aggregates [9].

To support fault tolerance for multi-job online aggregation, we consider three cases. Tasks that fail in j_1 recover as described in Section 3.3. If a task in j_2 fails, the system simply restarts the failed task. Since subsequent snapshots produced by j_1 are taken from a superset of the mapper output in j_1, the next snapshot received by the restarted reduce task in j_2 will have a higher progress score. To handle failures in j_1, tasks in j_2 cache the most recent snapshot received by j_1, and replace it when they receive a new snapshot with a higher progress metric. If tasks from both jobs fail, a new task in j_2 recovers the most recent snapshot from j_1 that was stored in HDFS and then wait for snapshots with a higher progress score.

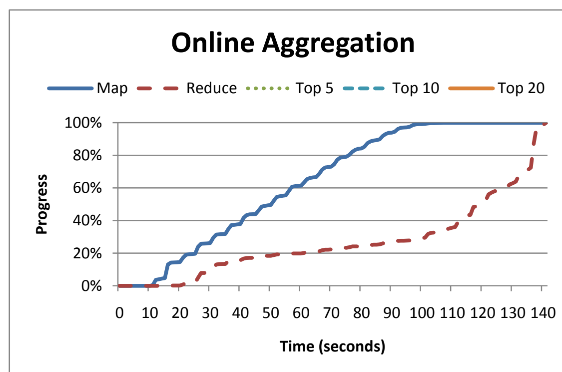


Figure 4: Top-100 query over 5.5GB of Wikipedia article text. The vertical lines describe the increasing accuracy of the approximate answers produced by online aggregation.

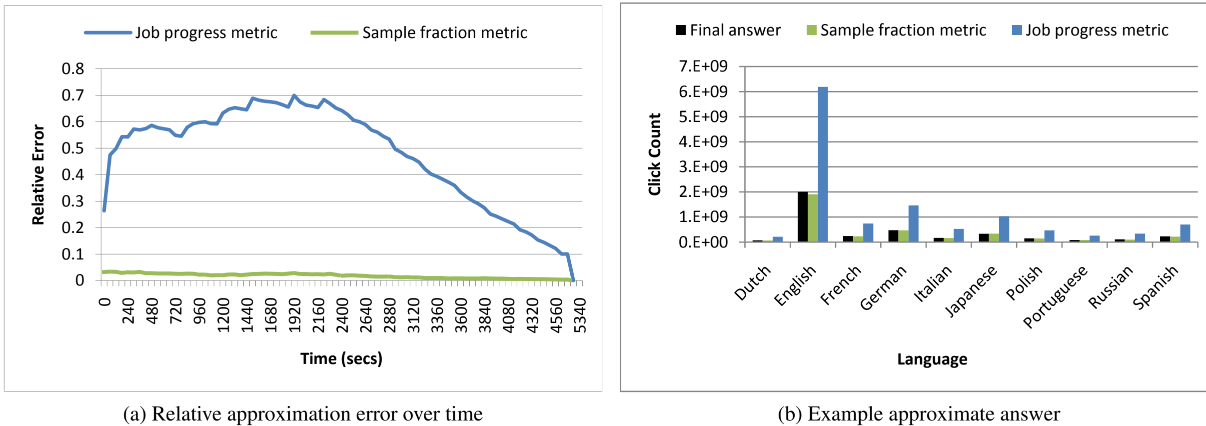
4.3 Evaluation

To evaluate the effectiveness of online aggregation, we performed two experiments on Amazon EC2 using different data sets and query workloads. In our first experiment, we wrote a "Top-K" query using two MapReduce jobs: the first job counts the frequency of each word and the second job selects the K most frequent words. We ran this workload on 5.5GB of Wikipedia article text stored in HDFS, using a 128MB block size. We used a 60-node EC2 cluster; each node was a "high-CPU medium" EC2 instance with 1.7GB of RAM and 2 virtual cores. A virtual core is the equivalent of a 2007-era 2.5Ghz Intel Xeon processor. A single EC2 node executed the Hadoop JobTracker and the HDFS NameNode, while the remaining nodes served as slaves for running the TaskTrackers and HDFS DataNodes.

Figure 4 shows the results of inter-job online aggregation for a Top-100 query. Our accuracy metric for this experiment is post-hoc — we note the time at which the Top-K words in the snapshot are the Top-K words in the final result. Although the final result for this job did not appear until nearly the end, we did observe the Top-5, 10, and 20 values at the times indicated in the graph. The Wikipedia data set was biased toward these Top-K words (e.g., "the", "is", etc.), which remained in their correct position throughout the lifetime of the job.

4.3.1 Approximation Metrics

In our second experiment, we considered the effectiveness of the job progress metric described in Section 4.1.1. Unsurprisingly, this metric can be inaccurate when it is used to estimate the accuracy of the approximate answers produced by online aggregation. In this experiment, we com-



(a) Relative approximation error over time

(b) Example approximate answer

Figure 5: Comparison of two approximation metrics. Figure (a) shows the relative error for each approximation metric over the runtime of the job, averaged over all groups. Figure (b) compares an example approximate answer produced by each metric with the final answer, for each language and for a single hour.

pared the job progress metric with a simple user-defined metric that leverages knowledge of the query and data set. HOP allows such metrics, although developing such a custom metric imposes more burden on the programmer than using the generic progress-based metric.

We used a data set containing seven months of hourly page view statistics for more than 2.5 million Wikipedia articles [26]. This constituted 320GB of compressed data (1TB uncompressed), divided into 5066 compressed files. We stored the data set on HDFS and assigned a single map task to each file, which was decompressed before the map function was applied.

We wrote a MapReduce job to count the total number of page views for each language and each hour of the day. In other words, our query grouped by language and hour of day, and summed the number of page views that occurred in each group. To enable more accurate approximate answers, we modified the map function to include the fraction of a given hour that each record represents. The reduce function summed these fractions for a given hour, which equated to one for all records from a single map task. Since the total number of hours was known ahead of time, we could use the result of this sum over all map outputs to determine the total fraction of each hour that had been sampled. We call this user-defined metric the "sample fraction."

To compute approximate answers, each intermediate result was scaled up using two different metrics: the generic metric based on job progress and the sample fraction described above. Figure 5a reports the relative error of the two metrics, averaged over all groups. Figure 5b shows an example approximate answer for a single hour using both metrics (computed two minutes into the job runtime). This figure also contains the final answer for comparison. Both results indicate that the sample fraction metric provides a much more accurate approximate answer for this query than the progress-based metric.

Job progress is clearly the wrong metric to use for approximating the final answer of this query. The primary reason is that it is too coarse of a metric. Each intermediate result was computed from some fraction of each hour. However, the job progress assumes that this fraction is uniform across all hours, when in fact we could have received much more of one hour and much less of another. This assumption of uniformity in the job progress resulted in a significant approximation error. By contrast, the sample fraction scales the approximate answer for each group according to the actual fraction of data seen for that group, yielding much more accurate approximations.

5 Continuous Queries

MapReduce is often used to analyze streams of constantly-arriving data, such as URL access logs [6] and system console logs [30]. Because of traditional constraints on MapReduce, this is done in large batches that can only provide periodic views of activity. This introduces significant latency into a data analysis process that ideally should run in near-real time. It is also potentially inefficient: each new MapReduce job does not have access to the computational state of the last analysis run, so this state must be recomputed from scratch. The programmer can manually save the state of each job and then reload it for the next analysis operation, but this is labor-intensive.

Our pipelined version of Hadoop allows an alternative architecture: MapReduce jobs that run *continuously*, accepting new data as it becomes available and analyzing it immediately. This allows for near-real-time analysis of data streams, and thus allows the MapReduce programming model to be applied to domains such as environment

monitoring and real-time fraud detection.

In this section, we describe how HOP supports continuous MapReduce jobs, and how we used this feature to implement a rudimentary cluster monitoring tool.

5.1 Continuous MapReduce Jobs

A bare-bones implementation of continuous MapReduce jobs is easy to implement using pipelining. No changes are needed to implement continuous map tasks: map output is already delivered to the appropriate reduce task shortly after it is generated. We added an optional "flush" API that allows map functions to force their current output to reduce tasks. When a reduce task is unable to accept such data, the mapper framework stores it locally and sends it at a later time. With proper scheduling of reducers, this API allows a map task to ensure that an output record is promptly sent to the appropriate reducer.

To support continuous reduce tasks, the user-defined reduce function must be periodically invoked on the map output available at that reducer. Applications will have different requirements for how frequently the reduce function should be invoked; possible choices include periods based on wall-clock time, logical time (e.g., the value of a field in the map task output), and the number of input rows delivered to the reducer. The output of the reduce function can be written to HDFS, as in our implementation of online aggregation. However, other choices are possible; our prototype system monitoring application (described below) sends an alert via email if an anomalous situation is detected.

In our current implementation, the number of map and reduce tasks is fixed, and must be configured by the user. This is clearly problematic: manual configuration is error-prone, and many stream processing applications exhibit "bursty" traffic patterns, in which peak load far exceeds average load. In the future, we plan to add support for elastic scaleup/scaledown of map and reduce tasks in response to variations in load.

5.1.1 Fault Tolerance

In the checkpoint/restart fault-tolerance model used by Hadoop, mappers retain their output until the end of the job to facilitate fast recovery from reducer failures. In a continuous query context, this is infeasible, since mapper history is in principle unbounded. However, many continuous reduce functions (e.g., 30-second moving average) only require a suffix of the map output stream. This common case can be supported easily, by extending the JobTracker interface to capture a rolling notion of reducer consumption. Map-side spill files are maintained in a ring buffer with unique IDs for spill files over time. When a reducer commits an output to HDFS, it informs the JobTracker about the *run* of map output records it no longer needs, identifying the run by spill file IDs and offsets within those files. The JobTracker can then tell mappers to garbage collect the appropriate data.

In principle, complex reducers may depend on very long (or infinite) histories of map records to accurately reconstruct their internal state. In that case, deleting spill files from the map-side ring buffer will result in potentially inaccurate recovery after faults. Such scenarios can be handled by having reducers checkpoint internal state to HDFS, along with markers for the mapper offsets at which the internal state was checkpointed. The MapReduce framework can be extended with APIs to help with state serialization and offset management, but it still presents a programming burden on the user to correctly identify the sensitive internal state. That burden can be avoided by more heavyweight process-pair techniques for fault tolerance, but those are quite complex and use significant resources [24]. In our work to date we have focused on cases where reducers can be recovered from a reasonable-sized history at the mappers, favoring minor extensions to the simple fault-tolerance approach used in Hadoop.

5.2 Prototype Monitoring System

Our monitoring system is composed of *agents* that run on each monitored machine and record statistics of interest (e.g., load average, I/O operations per second, etc.). Each agent is implemented as a continuous map task: rather than reading from HDFS, the map task instead reads from various system-local data streams (e.g., `/proc`).

Each agent forwards statistics to an *aggregator* that is implemented as a continuous reduce task. The aggregator records how agent-local statistics evolve over time (e.g., by computing windowed-averages), and compares statistics between agents to detect anomalous behavior. Each aggregator monitors the agents that report to it, but might also report statistical summaries to another "upstream" aggregator. For example, the system might be configured to have an aggregator for each rack and then a second level of aggregators that compare statistics between racks to analyze datacenter-wide behavior.

5.3 Evaluation

To validate our prototype system monitoring tool, we constructed a scenario in which one member of a MapReduce cluster begins thrashing during the execution of a job. Our goal was to test how quickly our monitoring system would detect this behavior. The basic mechanism is similar to an alert system one of the authors implemented at an Internet search company.

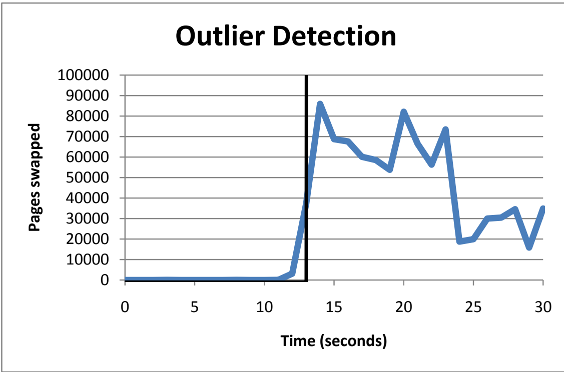


Figure 6: Number of pages swapped over time on the thrashing host, as reported by `vmstat`. The vertical line indicates the time at which the alert was sent by the monitoring system.

We used a simple load metric (a linear combination of CPU utilization, paging, and swap activity). The continuous reduce function maintains windows over samples of this metric: at regular intervals, it compares the 20 second moving average of the load metric for each host to the 120 second moving average of all the hosts in the cluster *except* that host. If the given host's load metric is more than two standard deviations above the global average, it is considered an outlier and a tentative alert is issued. To dampen false positives in "bursty" load scenarios, we do not issue an alert until we have received 10 tentative alerts within a time window.

We deployed this system on an EC2 cluster consisting of 7 "large" nodes (large nodes were chosen because EC2 allocates an entire physical host machine to them). We ran a wordcount job on the 5.5GB Wikipedia data set, using 5 map tasks and 2 reduce tasks (1 task per host). After the job had been running for about 10 seconds, we selected a node running a task and launched a program that induced thrashing.

We report detection latency in Figure 6. The vertical bar indicates the time at which the monitoring tool fired a (non-tentative) alert. The thrashing host was detected very rapidly—notably faster than the 5-second TaskTracker-JobTracker heartbeat cycle that is used to detect straggler tasks in stock Hadoop. We envision using these alerts to do early detection of stragglers within a MapReduce job: HOP could make scheduling decisions for a job by running a secondary continuous monitoring query. Compared to out-of-band monitoring tools, this economy of mechanism—reusing the MapReduce infrastructure for reflective monitoring—has benefits in software maintenance and system management.

6 Performance Evaluation

A thorough performance comparison between pipelining and blocking is beyond the scope of this paper. In this section, we instead demonstrate that pipelining can reduce job completion times in some configurations.

We report performance using both large (512MB) and small (32MB) HDFS block sizes using a single workload (a wordcount job over randomly-generated text). Since the words were generated using a uniform distribution, map-side combiners were ineffective for this workload. We performed all experiments using relatively small clusters of Amazon EC2 nodes. We also did not consider performance in an environment where multiple concurrent jobs are executing simultaneously.

6.1 Background and Configuration

Before diving into the performance experiments, it is important to further describe the division of labor in a HOP job, which is broken into task phases. A map task consists of two work phases: *map* and *sort*. The majority of work is performed in the *map* phase, where the map function is applied to each record in the input and subsequently sent to an output buffer. Once the entire input has been processed, the map task enters the *sort* phase, where a final merge sort of all intermediate spill files is performed before registering the final output with the TaskTracker. The progress reported by a map task corresponds to the *map* phase only.

A reduce task in HOP is divided into three work phases: *shuffle*, *reduce*, and *commit*. In the *shuffle* phase, reduce tasks receive their portion of the output from each map. In HOP, the *shuffle* phase consumes 75% of the overall reduce task progress while the remaining 25% is allocated to the *reduce* and *commit* phase.[3] In the *shuffle* phase, reduce tasks periodically perform a merge sort on the already received map output. These intermediate merge sorts decrease the amount of sorting work performed at the end of the *shuffle* phase. After receiving its portion of data from all map tasks, the reduce task performs a final merge sort and enters the *reduce* phase.

By pushing work from map tasks to reduce tasks more aggressively, pipelining can enable better overlapping of map and reduce computation, especially when the node on which a reduce task is scheduled would otherwise be underutilized. However, when reduce tasks are already the bottleneck, pipelining offers fewer performance benefits, and may even hurt performance by placing additional load on the reduce nodes.

[3] The stock version of Hadoop divides the reduce progress evenly among the three phases. We deviated from this approach because we wanted to focus more on the progress during the *shuffle* phase.

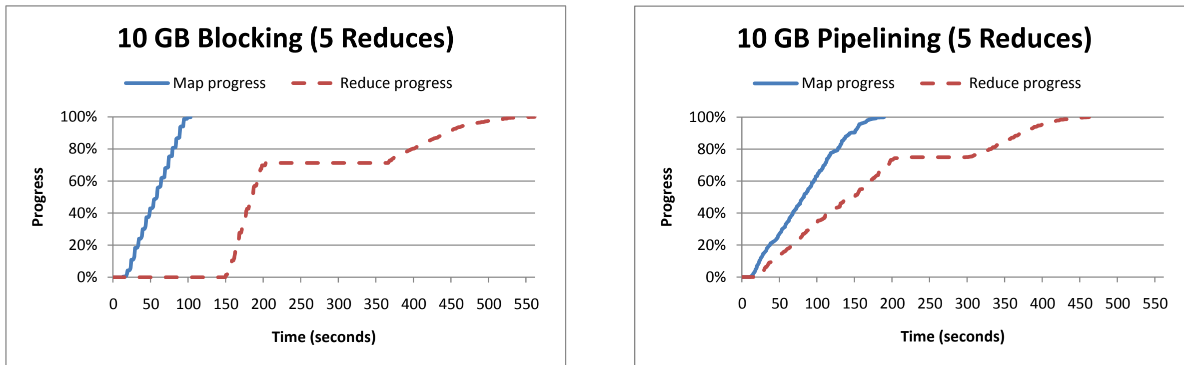


Figure 7: CDF of map and reduce task completion times for a 10GB wordcount job using 20 map tasks and 5 reduce tasks (512MB block size). The total job runtimes were 561 seconds for blocking and 462 seconds for pipelining.

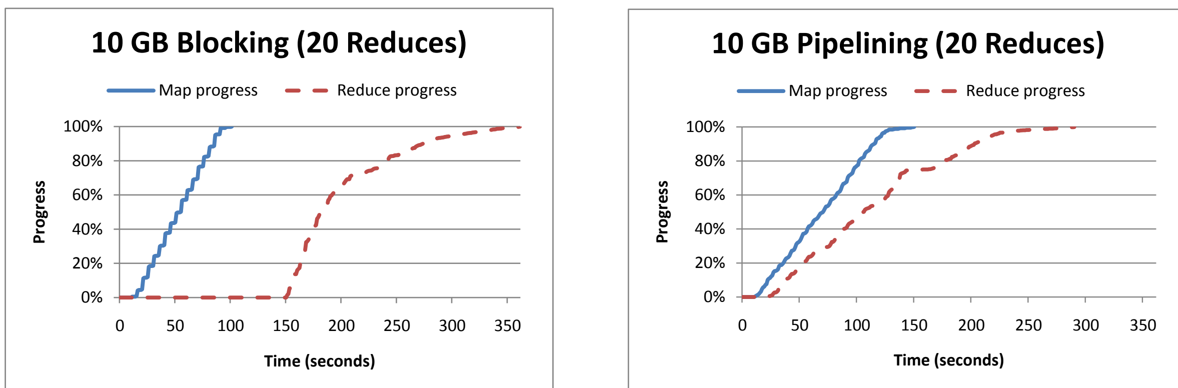


Figure 8: CDF of map and reduce task completion times for a 10GB wordcount job using 20 map tasks and 20 reduce tasks (512MB block size). The total job runtimes were 361 seconds for blocking and 290 seconds for pipelining.

The *sort* phase in the map task minimizes the merging work that reduce tasks must perform at the end of the *shuffle* phase. When pipelining is enabled, the *sort* phase is avoided since map tasks have already sent some fraction of the spill files to concurrently running reduce tasks. Therefore, pipelining increases the merging workload placed on the reducer. The adaptive pipelining scheme described in Section 3.1.3 attempts to ensure that reduce tasks are not overwhelmed with additional load.

We used two Amazon EC2 clusters depending on the size of the experiment: "small" jobs used 10 worker nodes, while "large" jobs used 20. Each node was an "extra large" EC2 instances with 15GB of memory and four virtual cores.

6.2 Small Job Results

Our first experiment focused on the performance of small jobs in an underutilized cluster. We ran a 10GB word-count with a 512MB block size, yielding 20 map tasks. We used 10 worker nodes and configured each worker to execute at most two map and two reduce tasks simultaneously. We ran several experiments to compare the performance of blocking and pipelining using different numbers of reduce tasks.

Figure 7 reports the results with five reduce tasks. A plateau can be seen at 75% progress for both blocking and pipelining. At this point in the job, all reduce tasks have completed the *shuffle* phase; the plateau is caused by the time taken to perform a final merge of all map output before entering the *reduce* phase. Notice that the plateau for the pipelining case is shorter. With pipelining, reduce tasks receive map outputs earlier and can begin sorting earlier, thereby reducing the time required for the final merge.

Figure 8 reports the results with twenty reduce tasks. Using more reduce tasks decreases the amount of merging that any one reduce task must perform, which reduces the duration of the plateau at 75% progress. In the blocking case, the plateau is practically gone.

Note that in both experiments, the map phase finishes faster with blocking than with pipelining. This is because pipelining allows reduce tasks to begin executing more quickly; hence, the reduce tasks compete for resources with the map tasks, causing the map phase to take slightly

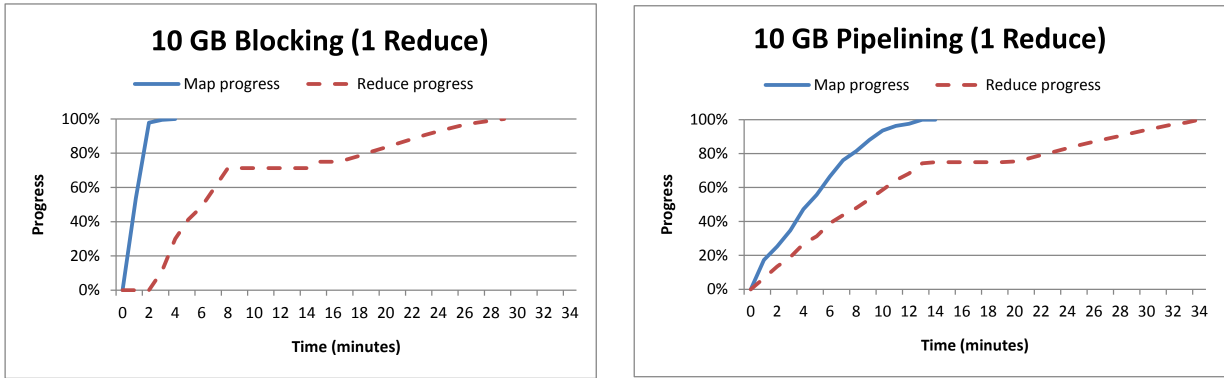


Figure 9: CDF of map and reduce task completion times for a 10GB wordcount job using 20 map tasks and 1 reduce task (512MB block size). The total job runtimes were 29 minutes for blocking and 34 minutes for pipelining.

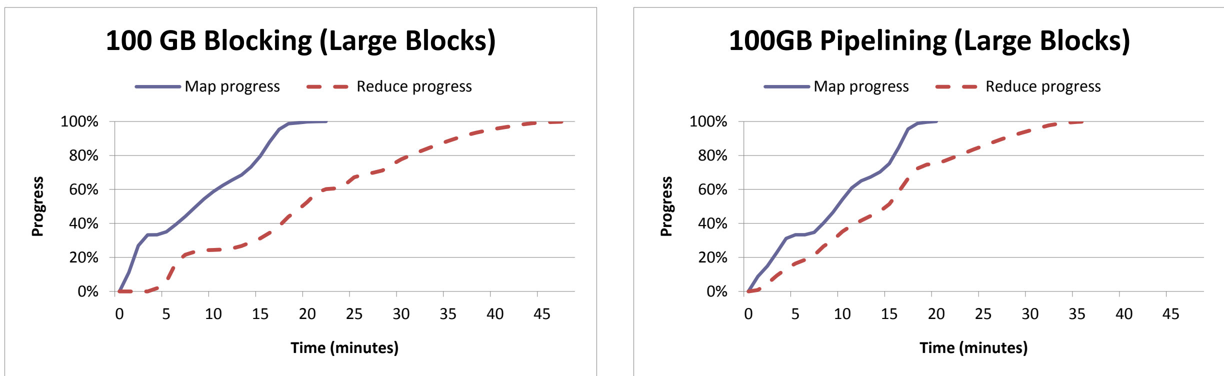


Figure 10: CDF of map and reduce task completion times for a 100GB wordcount job using 240 map tasks and 60 reduce tasks (512MB block size). The total job runtimes were 48 minutes for blocking and 36 minutes for pipelining.

longer. In this case, the increase in map duration is outweighed by the increase in cluster utilization, resulting in shorter job completion times: pipelining reduced completion time by 17.7% with 5 reducers and by 19.7% with 20 reducers.

Figure 9 describes an experiment in which we ran a 10GB wordcount job using a single reduce task. This caused job completion times to increase dramatically for both pipelining and blocking, because of the extreme load placed on the reduce node. Pipelining delayed job completion by ~17%, which suggests that our simple adaptive flow control scheme (Section 3.1.3) was unable to move load back to the map tasks aggressively enough.

6.3 Large Job Results

Our second set of experiments focused on the performance of somewhat larger jobs. We increased the input size to 100GB (from 10GB) and the number of worker nodes to 20 (from 10). Each worker was configured to execute at most four map and three reduce tasks, which meant that at most 80 map and 60 reduce tasks could execute at once. We conducted two sets of experimental runs, each run comparing blocking to pipelining using either large (512MB) or small (32MB) block sizes. We were interested in blocking performance with small block sizes because blocking can effectively emulate pipelining if the block size is small enough.

Figure 10 reports the performance of a 100GB wordcount job with 512MB blocks, which resulted in 240 map tasks, scheduled in three waves of 80 tasks each. The 60 reduce tasks were coscheduled with the first wave of map tasks. In the blocking case, the reduce tasks began working as soon as they received the output of the first wave, which is why the reduce progress begins to climb around four minutes (well before the completion of all maps). Pipelining was able to achieve significantly better cluster utilization, and hence reduced job completion time by ~25%.

Figure 11 reports the performance of blocking and pipelining using 32MB blocks. While the performance of pipelining remained similar, the performance of blocking improved considerably, but still trailed somewhat behind pipelining. Using block sizes smaller than 32MB did

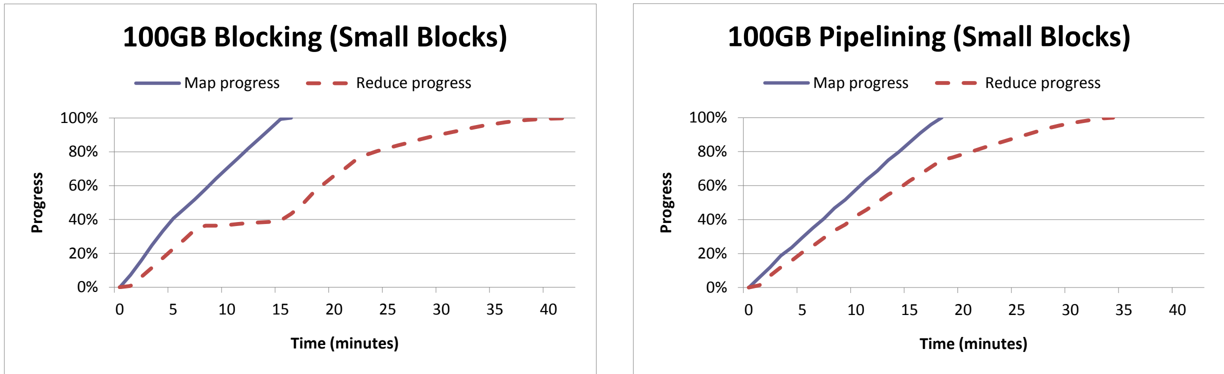


Figure 11: CDF of map and reduce task completion times for a 100GB wordcount job using 3120 map tasks and 60 reduce tasks (32MB block size). The total job runtimes were 42 minutes for blocking and 34 minutes for pipelining.

not yield a significant performance improvement in our experiments.

7 Related Work

The work in this paper relates to literature on parallel dataflow frameworks, online aggregation, and continuous query processing.

7.1 Parallel Dataflow

Dean and Ghemawat's paper on Google's MapReduce [6] has become a standard reference, and forms the basis of the open-source Hadoop implementation. As noted in Section 1, the Google MapReduce design targets very large clusters where the probability of worker failure or slowdown is high. This led to their elegant checkpoint/restart approach to fault tolerance, and their lack of pipelining. Our work extends the Google design to accommodate pipelining without significant modification to their core programming model or fault tolerance mechanisms.

Dryad [13] is a data-parallel programming model and runtime that is often compared to MapReduce, supporting a more general model of acyclic dataflow graphs. Like MapReduce, Dryad puts disk materialization steps between dataflow stages by default, breaking pipelines. The Dryad paper describes support for optionally "encapsulating" multiple asynchronous stages into a single process so they can pipeline, but this requires a more complicated programming interface. The Dryad paper explicitly mentions that the system is targeted at batch processing, and not at scenarios like continuous queries.

It has been noted that parallel database systems have long provided partitioned dataflow frameworks [21], and recent commercial databases have begun to offer MapReduce programming models on top of those frameworks [5, 10]. Most parallel database systems can provide pipelined execution akin to our work here, but they use a more tightly coupled iterator and *Exchange* model that keeps producers and consumers rate-matched via queues, spreading the work of each dataflow stage across all nodes in the cluster [8]. This provides less scheduling flexibility than MapReduce and typically offers no tolerance to mid-query worker faults. Yang et al. recently proposed a scheme to add support for mid-query fault tolerance to traditional parallel databases, using a middleware-based approach that shares some similarities with MapReduce [31].

Logothetis and Yocum describe a MapReduce interface over a continuous query system called *Mortar* that is similar in some ways to our work [16]. Like HOP, their mappers push data to reducers in a pipelined fashion. They focus on specific issues in efficient stream query processing, including minimization of work for aggregates in overlapping windows via special reducer APIs. They are not built on Hadoop, and explicitly sidestep issues in fault tolerance.

Hadoop Streaming is part of the Hadoop distribution, and allows map and reduce functions to be expressed as UNIX shell command lines. It does not stream data through map and reduce phases in a pipelined fashion.

7.2 Online Aggregation

Online aggregation was originally proposed in the context of simple single-table SQL queries involving "Group By" aggregations, a workload quite similar to MapReduce [12]. The focus of the initial work was on providing not only "early returns" to these SQL queries, but also statistically robust estimators and confidence interval metrics for the final result based on random sampling. These statistical matters do not generalize to arbitrary MapReduce jobs, though our framework can support those that have been developed. Subsequently, online aggregation was extended to handle join queries (via the *Ripple Join* method),

and the *CONTROL* project generalized the idea of online query processing to provide interactivity for data cleaning, data mining, and data visualization tasks [11]. That work was targeted at single-processor systems. Luo et al. developed a partitioned-parallel variant of Ripple Join, without statistical guarantees on approximate answers [17].

In recent years, this topic has seen renewed interest, starting with Jermaine et al.'s work on the *DBO* system [15]. That effort includes more disk-conscious online join algorithms, as well as techniques for maintaining randomly-shuffled files to remove any potential for statistical bias in scans [14]. Wu et al. describe a system for peer-to-peer online aggregation in a distributed hash table context [29]. The open programmability and fault-tolerance of MapReduce are not addressed significantly in prior work on online aggregation.

An alternative to online aggregation combines precomputation with sampling, storing fixed samples and summaries to provide small storage footprints and interactive performance [7]. An advantage of these techniques is that they are compatible with both pipelining and blocking models of MapReduce. The downside of these techniques is that they do not allow users to choose the query stopping points or time/accuracy trade-offs dynamically [11].

7.3 Continuous Queries

In the last decade there was a great deal of work in the database research community on the topic of continuous queries over data streams, including systems such as Borealis [1], STREAM [18], and Telegraph [4]. Of these, Borealis and Telegraph [24] studied fault tolerance and load balancing across machines. In the Borealis context this was done for pipelined dataflows, but without partitioned parallelism: each stage ("operator") of the pipeline runs serially on a different machine in the wide area, and fault tolerance deals with failures of entire operators [3]. SBON [22] is an overlay network that can be integrated with Borealis, which handles "operator placement" optimizations for these wide-area pipelined dataflows.

Telegraph's *FLuX* operator [24, 25] is the only work to our knowledge that addresses mid-stream fault-tolerance for dataflows that are both pipelined and partitioned in the style of HOP. FLuX ("Fault-tolerant, Load-balanced eXchange") is a dataflow operator that encapsulates the shuffling done between stages such as map and reduce. It provides load-balancing interfaces that can migrate operator state (e.g., reducer state) between nodes, while handling scheduling policy and changes to data-routing policies [25]. For fault tolerance, FLuX develops a solution based on process pairs [24], which work redundantly to ensure that operator state is always being maintained live on multiple nodes. This removes any burden on the continuous query programmer of the sort we describe in Section 5. On the other hand, the FLuX protocol is far more complex and resource-intensive than our pipelined adaptation of Google's checkpoint/restart tolerance model.

8 Conclusion and Future Work

MapReduce has proven to be a popular model for large-scale parallel programming. Our Hadoop Online Prototype extends the applicability of the model to pipelining behaviors, while preserving the simple programming model and fault tolerance of a full-featured MapReduce framework. This provides significant new functionality, including "early returns" on long-running jobs via online aggregation, and continuous queries over streaming data. We also demonstrate benefits for batch processing: by pipelining both within and across jobs, HOP can reduce the time to job completion.

In considering future work, scheduling is a topic that arises immediately. Stock Hadoop already has many degrees of freedom in scheduling batch tasks across machines and time, and the introduction of pipelining in HOP only increases this design space. First, pipeline parallelism is a new option for improving performance of MapReduce jobs, but needs to be integrated intelligently with both intra-task partition parallelism and speculative redundant execution for "straggler" handling. Second, the ability to schedule deep pipelines with direct communication between reduces and maps (bypassing the distributed file system) opens up new opportunities and challenges in carefully co-locating tasks from different jobs, to avoid communication when possible.

Olston and colleagues have noted that MapReduce systems—unlike traditional databases—employ "model-light" optimization approaches that gather and react to performance information during runtime [19]. The continuous query facilities of HOP enable powerful introspective programming interfaces for this: a full-featured MapReduce interface can be used to script performance monitoring tasks that gather system-wide information in near-real-time, enabling tight feedback loops for scheduling and dataflow optimization. This is a topic we plan to explore, including opportunistic methods to do monitoring work with minimal interference to outstanding jobs, as well as dynamic approaches to continuous optimization in the spirit of earlier work like Eddies [2] and FLuX [25].

As a more long-term agenda, we want to explore using MapReduce-style programming for even more interactive applications. As a first step, we hope to revisit interactive data processing in the spirit of the CONTROL work [11], with an eye toward improved scalability via parallelism. More aggressively, we are considering the idea of bridging the gap between MapReduce dataflow programming and lightweight event-flow programming models like SEDA [28]. Our HOP implementation's roots

in Hadoop make it unlikely to compete with something like SEDA in terms of raw performance. However, it would be interesting to translate ideas across these two traditionally separate programming models, perhaps with an eye toward building a new and more general-purpose framework for programming in architectures like cloud computing and many-core.

Acknowledgments

We would like to thank Daniel Abadi, Kuang Chen, Mosharaf Chowdhury, Akshay Krishnamurthy, Andrew Pavlo, Hong Tang , and our shepherd Jeff Dean for their helpful comments and suggestions. This material is based upon work supported by the National Science Foundation under Grant Nos. 0713661, 0722077 and 0803690, the Air Force Office of Scientific Research under Grant No. FA95500810352, the Natural Sciences and Engineering Research Council of Canada, and gifts from IBM, Microsoft, and Yahoo!.

References

[1] ABADI, D. J., AHMAD, Y., BALAZINSKA, M., CETINTEMEL, U., CHERNIACK, M., HWANG, J.-H., LINDNER, W., MASKEY, A. S., RASIN, A., RYVKINA, E., TATBUL, N., XING, Y., AND ZDONIK, S. The design of the Borealis stream processing engine. In *CIDR* (2005).

[2] AVNUR, R., AND HELLERSTEIN, J. M. Eddies: Continuously adaptive query processing. In *SIGMOD* (2000).

[3] BALAZINSKA, M., BALAKRISHNAN, H., MADDEN, S., AND STONEBRAKER, M. Fault-tolerance in the Borealis distributed stream processing system. In *SIGMOD* (2005).

[4] CHANDRASEKARAN, S., COOPER, O., DESHPANDE, A., FRANKLIN, M. J., HELLERSTEIN, J. M., HONG, W., KRISHNAMURTHY, S., MADDEN, S. R., RAMAN, V., REISS, F., AND SHAH, M. A. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR* (2003).

[5] CIESLEWICZ, J., FRIEDMAN, E., AND PAWLOWSKI, P. SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. In *VLDB* (2009).

[6] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *OSDI* (2004).

[7] GIBBONS, P. B., AND MATIAS, Y. New sampling-based summary statistics for improving approximate query answers. In *SIGMOD* (1998).

[8] GRAEFE, G. Encapsulation of parallelism in the Volcano query processing system. In *SIGMOD* (1990).

[9] GRAY, J., CHAUDHURI, S., BOSWORTH, A., LAYMAN, A., REICHART, D., VENKATRAO, M., PELLOW, F., AND PIRAHESH, H. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals. *Data Min. Knowl. Discov. 1*, 1 (1997), 29–53.

[10] Greenplum: A unified engine for RDBMS and MapReduce, Oct. 2008. Downloaded from `http://www.greenplum.com/download.php?alias=register-map-reduce&file=Greenplum-MapReduce-Whitepaper.pdf`.

[11] HELLERSTEIN, J. M., AVNUR, R., CHOU, A., HIDBER, C., OLSTON, C., RAMAN, V., ROTH, T., AND HAAS, P. J. Interactive data analysis with CONTROL. *IEEE Computer 32*, 8 (Aug. 1999).

[12] HELLERSTEIN, J. M., HAAS, P. J., AND WANG, H. J. Online aggregation. In *SIGMOD* (1997).

[13] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys* (2007).

[14] JERMAINE, C. Online random shuffling of large database tables. *IEEE Trans. Knowl. Data Eng. 19*, 1 (2007), 73–84.

[15] JERMAINE, C., ARUMUGAM, S., POL, A., AND DOBRA, A. Scalable approximate query processing with the DBO engine. In *SIGMOD* (2007).

[16] LOGOTHETIS, D., AND YOCUM, K. Ad-hoc data processing in the cloud (demonstration). *Proc. VLDB Endow. 1*, 2 (2008).

[17] LUO, G., ELLMANN, C. J., HAAS, P. J., AND NAUGHTON, J. F. A scalable hash ripple join algorithm. In *SIGMOD* (2002).

[18] MOTWANI, R., WIDOM, J., ARASU, A., BABCOCK, B., BABU, S., DATAR, M., MANKU, G., OLSTON, C., ROSENSTEIN, J., AND VARMA, R. Query processing, resource management, and approximation in a data stream management system. In *CIDR* (2003).

[19] OLSTON, C., REED, B., SILBERSTEIN, A., AND SRIVASTAVA, U. Automatic optimization of parallel dataflow programs. In *USENIX Technical Conference* (2008).

[20] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD* (2008).

[21] PAVLO, A., PAULSON, E., RASIN, A., ABADI, D. J., DEWITT, D. J., MADDEN, S., AND STONEBRAKER, M. A comparison of approaches to large-scale data analysis. In *SIGMOD* (2009).

[22] PIETZUCH, P., LEDLIE, J., SHNEIDMAN, J., ROUSSOPOULOS, M., WELSH, M., AND SELTZER, M. Network-aware operator placement for stream-processing systems. In *ICDE* (2006).

[23] PIKE, R., DORWARD, S., GRIESEMER, R., AND QUINLAN, S. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming 13*, 4 (2005), 277–298.

[24] SHAH, M. A., HELLERSTEIN, J. M., AND BREWER, E. A. Highly-available, fault-tolerant, parallel dataflows. In *SIGMOD* (2004).

[25] SHAH, M. A., HELLERSTEIN, J. M., CHANDRASEKARAN, S., AND FRANKLIN, M. J. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE* (2003).

[26] SKOMOROCH, P. N. Wikipedia page traffic statistics, 2009. Downloaded from `http://developer.amazonwebservices.com/connect/entry.jspa?externalID=2596`.

[27] THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ANTHONY, S., LIU, H., WYCKOFF, P., AND MURTHY, R. Hive — a warehousing solution over a Map-Reduce framework. In *VLDB* (2009).

[28] WELSH, M., CULLER, D., AND BREWER, E. SEDA: An architecture for well-conditioned, scalable internet services. In *SOSP* (2001).

[29] WU, S., JIANG, S., OOI, B. C., AND TAN, K.-L. Distributed online aggregation. In *VLDB* (2009).

[30] XU, W., HUANG, L., FOX, A., PATTERSON, D., AND JORDAN, M. I. Detecting large-scale system problems by mining console logs. In *SOSP* (2009).

[31] YANG, C., YEN, C., TAN, C., AND MADDEN, S. Osprey: Implementing MapReduce-style fault tolerance in a shared-nothing distributed database. In *ICDE* (2010).

The Architecture and Implementation of an Extensible Web Crawler

Jonathan M. Hsieh, Steven D. Gribble, and Henry M. Levy
Department of Computer Science & Engineering
University of Washington, Seattle, WA, USA 98195
{jmhsieh,gribble,levy}@cs.washington.edu

Abstract

Many Web services operate their own Web crawlers to discover data of interest, despite the fact that large-scale, timely crawling is complex, operationally intensive, and expensive. In this paper, we introduce the extensible crawler, a service that crawls the Web on behalf of its many client applications. Clients inject filters into the extensible crawler; the crawler evaluates all received filters against each Web page, notifying clients of matches. As a result, the act of crawling the Web is decoupled from determining whether a page is of interest, shielding client applications from the burden of crawling the Web themselves.

This paper describes the architecture, implementation, and evaluation of our prototype extensible crawler, and also relates early experience from several crawler applications we have built. We focus on the challenges and trade-offs in the system, such as the design of a filter language that is simultaneously expressive and efficient to execute, the use of filter indexing to cheaply match a page against millions of filters, and the use of document and filter partitioning to scale our prototype implementation to high document throughput and large numbers of filters. We argue that the low-latency, high selectivity, and scalable nature of our system makes it a promising platform for taking advantage of emerging real-time streams of data, such as Facebook or Twitter feeds.

1 Introduction

Over the past decade, an astronomical amount of information has been published on the Web. As well, Web services such as Twitter, Facebook, and Digg reflect a growing trend to provide people and applications with access to real-time streams of information updates. Together, these two characteristics imply that the Web has become an exceptionally potent repository of programmatically accessible data. Some of the most provocative recent Web applications are those that gather and process large-scale Web data, such as virtual tourism [33], knowledge extraction [15], Web site trust assessment [24], and emerging trend detection [6].

New Web services that want to take advantage of Web-scale data face a high barrier to entry. Finding and accessing data of interest requires crawling the Web, and if a service is sensitive to quick access to newly published data, its Web crawl must operate continuously and focus on the most relevant subset of the Web. Unfortunately, massive-scale, timely web crawling is complex, operationally intensive, and expensive. Worse, for services that are only interested in specific subsets of Web data, crawling is wasteful, as most pages retrieved will not match their criteria of interest.

In this paper, we introduce the *extensible crawler*, a utility service that crawls the Web on behalf of its many client applications. An extensible crawler lets clients specify filters that are evaluated over each crawled Web page; if a page matches one of the filters specified by a client, the client is notified of the match. As a result, the act of crawling the Web is decoupled from the application-specific logic of determining if a page is of interest, shielding Web-crawler applications from the burden of crawling the Web themselves.

We anticipate two deployment modes for an extensible crawler. First, it can run as a service accessible remotely across the wide-area Internet. In this scenario, filter sets must be very highly selective, since the bandwidth between the extensible crawler and a client application is scarce and expensive. Second, it can run as a utility service [17] within cloud computing infrastructure such as Amazon's EC2 or Google's AppEngine. Filters can be much less selective in this scenario, since bandwidth between the extensible crawler and its clients is abundant, and the clients can pay to scale up the computation processing selected documents.

This paper describes our experience with the design, implementation, and evaluation of an extensible crawler, focusing on the challenges and trade-offs inherent in this class of system. For example, an extensible crawler's filter language must be sufficiently expressive to support interesting applications, but simultaneously, filters must be efficient to execute. A naïve implementation of an extensible crawler would require computational resources

proportional to the number of filters it supports multiplied by its crawl rate; instead, our extensible crawler prototype uses standard indexing techniques to vastly reduce the cost of executing a large number of filters. To scale, an extensible crawler must be distributed across a cluster. Accordingly, the system must balance load (both filters and pages) appropriately across machines, otherwise an overloaded machine will limit the rate at which the entire system can process crawled pages. Finally, there must be appropriate mechanisms in place to allow web-crawler applications to update their filter sets frequently and efficiently.

We demonstrate that XCrawler, our early prototype system, is scalable across several dimensions: it can efficiently process tens of millions of concurrent filters while processing thousands of Web pages per second. XCrawler is also flexible. By construction, we show that its filter specification language facilitates a wide range of interesting web-crawler applications, including keyword-based notification, Web malware detection and defense, and copyright violation detection.

An extensible crawler bears similarities to several other systems, including streaming and parallel databases [1, 10, 11, 13, 14, 19], publish-subscribe systems [2, 9, 16, 27, 31], search engines and web crawlers [8, 12, 20, 21], and packet filters [23, 25, 30, 32, 34]. Our design borrows techniques from each, but we argue that the substantial differences in the workload, scale, and application requirements of extensible crawlers mandate many different design choices and optimizations. We compare XCrawler to related systems in the related work section (Section 5), and we provide an in-depth comparison to search engines in Section 2.1.

2 Overview

To better motivate the goals and requirements of extensible crawlers, we now describe a set of web-crawler applications that we have experimented with using our prototype system. Table 1 gives some order-of-magnitude estimates of the workload that we expect these applications would place on an extensible crawler if deployed at scale, including the total number of filters each application category would create and the selectivity of a client's filter set.

Keyword-based notification. Similar to Google Alerts, this application allows users to register keyword phrases of interest, and receive an event stream corresponding to Web pages containing those keywords. For example, users might upload a vanity filter ("Jonathan Hsieh"), or a filter to track a product or company ("palm pre"). This application must support a large number of users, each with a small and relatively slowly-changing filter set. Each filter should be highly selective, matching a very small fraction of Web pages.

	keyword notification	Web malware	copyright violation	Web research
# clients	~10^6	~10^2	~10^2	~10^3
# filters per client	~10^2	~10^6	~10^6	~10^2
fraction of pages that match for a client	~10^-5	~10^-4	~10^-6	~10^-3
total # filters	~10^8	~10^8	~10^8	~10^5

Table 1: **Web-crawler application workloads.** This table summarizes the approximate filter workloads we expect from four representative applications.

Web malware detection. This application uses a database of regular-expression-based signatures to identify malicious executables, JavaScript, or Web content. New malware signatures are injected daily, and clients require prompt notification when new malicious pages are discovered. This application must support a small number of clients (e.g., `McAfee`, `Google`, and `Symantec`), each with a large and moderately quickly changing filter set. Each filter should be highly selective; in aggregate, approximately roughly 1 in 1000 Web pages contain malicious content [26, 28].

Copyright violation detection. Similar to commercial offerings such as `attributor.com`, this application lets clients find Web pages containing content containing their intellectual property. A client, such as a news provider, maintains a large database of highly selective filters, such as key sentences from their news articles. New filters are injected into the system by a client whenever new content is published. This application must support a moderate number of clients, each with a large, selective, and potentially quickly changing filter set.

Web measurement research. This application permits scientists to perform large-scale measurements of the Web to characterize its content and dynamics. Individual research projects would inject filters to randomly sample Web pages (e.g., sample 1 in 1000 random pages as representative of the overall Web) or to select Web pages with particular features and tags relevant to the study (e.g., select Ajax-related JavaScript keywords in a study investigating the prevalence of Ajax on the Web). This application would support a modest number of clients with a moderately sized, slowly changing filter set.

2.1 Comparison to a search engine

At first glance, one might consider implementing an extensible crawler as a layer on top of a conventional search engine. This strawman would periodically execute filters against the search engine, looking for new document matches and transmitting those to applications. On closer inspection, however, several fundamental differences between search engines and extensible crawlers, their workloads, and their performance requirements are evident, as summarized in Table 2. Because of

	search engine	extensible crawler
clients	millions of people	thousands of applications
documents	~trillion stored in a crawl database and periodically refreshed by crawler	arrive in a stream from crawler, processed on-the-fly and not stored
filters / queries	arrive in a stream from users, processed on-the-fly and not stored	~billion stored in a filter database and periodically updated by applications
indexing	index documents	index queries
latency	query response time is crucial; document refresh latency is less important	document processing time is crucial; filter update latency is less important
selectivity and query result ranking	queries might not be selective; result ranking is crucial for usability	filters are assumed to be selective; all matches are sent to applications
caching	in-memory cache of popular query results and "important" index subset	entire filter index is stored in memory; result caching is not relevant

Table 2: **Search engines vs. extensible crawlers.** This table summarizes key distinctions between the workload, performance, and scalability requirements of search engines and extensible crawlers.

these differences, we argue that there is an opportunity to design an extensible crawler that will scale more efficiently and better suit the needs of its applications than a search-engine-based implementation.

In many regards, an extensible crawler is an inversion of a search engine. A search engine crawls the Web to periodically update its stored index of Web documents, and receives a stream of Web queries that it processes against the document index on-the-fly. In contrast, an extensible crawler periodically updates its stored index of filters, and receives a stream of Web documents that it processes against the filter index on-the-fly. For a search engine, though it is important to reduce the time in between document index updates, it is crucial to minimize query response time. For an extensible crawler, it is important to be responsive in receiving filter updates from clients, but for "real-time Web" applications, it is more important to process crawled documents with low latency.

There are also differences in scale between these two systems. A search engine must store and index hundreds of billions, if not trillions, of Web documents, containing kilobytes or megabytes of data. On the other hand, an extensible crawler must store and index hundreds of millions, or billions, of filters; our expectation is that filters are small, perhaps dozens or hundreds of bytes. As a result, an extensible crawler must store and index four or five orders of magnitude less data than a search engine, and it is more likely to be able to afford to keep its entire index resident in memory.

Finally, there are important differences in the performance and result accuracy requirements of the two systems. A given search engine query might match millions of Web pages. To be usable, the search engine must rely heavily on page ranking to present the top matches to users. Filters for an extensible crawler are assumed to be more selective than search engine queries, but even if they are not, filters are executed against documents as they are crawled rather than against the enormous Web corpus gathered by a search engine. All matching pages found by an extensible crawler are communicated to a web-crawler application; result ranking is not relevant.

Traditional search engines and extensible crawlers are in some ways complementary, and they can co-exist. Our work focuses on quickly matching freshly crawled documents against a set of filters, however, many applications can benefit from being able to issue queries against a full, existing Web index in addition to filtering newly discovered content.

2.2 Architectural goals

Our extensible crawler architecture has been guided by several principles and system goals:

High Selectivity. The primary role of an extensible crawler is to reduce the number of web pages a web-crawler application must process by a substantial amount, while preserving pages in which the application might have interest. An extensible crawler can be thought of as a highly selective, programmable matching filter executing as a pipeline stage between a stock Web crawler and a web-crawler application.

Indexability. When possible, an extensible crawler should trade off CPU for memory to reduce the computational cost of supporting a large number of filters. In practice, this implies constructing an index over filters to support the efficient matching of a document against all filters. One implication of this is that the index must be kept up-to-date as the set of filters defined by web-crawler applications is updated. If this update rate is low or the indexing technique used supports incremental updates, keeping the index up-to-date should be efficient.

Favor Efficiency over Precision. There is generally a tradeoff between the precision of a filter and its efficient execution, and in these cases, an extensible crawler should favor efficient execution. For example, a filter language that supports regular expressions can be more precise than a filter language that supports only conjuncts of substrings, but it is simpler to build an efficient index over the latter. As we will discuss in Section 3.2.2, our XCrawler prototype implementation exposes a rich filter language to web-crawler applications, but uses *relaxation* to convert precise filters into less-precise, indexable versions, increasing its scalability at the cost of exposing false positive matches to the applications.

Low Latency. To support crawler-applications that depend on real-time Web content, an extensible crawler should be capable of processing Web pages with low latency. This goal suggests the extensible crawler should

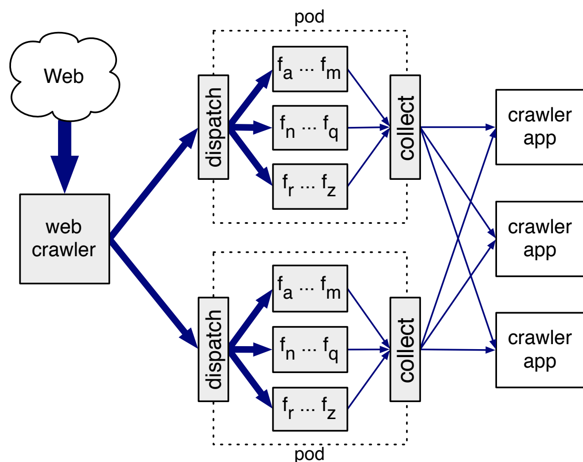


Figure 1: **Extensible crawler architecture.** This figure depicts the high-level architecture of an extensible crawler, including the flow of documents from the Web through the system.

be architected as a stage in a dataflow pipeline, rather than as a batch or map-reduce style computation.

Scalability. An extensible crawler should scale up to support high Web page processing rates and a very large number of filters. One of our specific goals is to handle a linear increase in document processing rate with a corresponding linear increase in machine resources.

2.3 System architecture

Figure 1 shows the high-level architecture of an extensible crawler. A conventional Web crawler is used to fetch a high rate stream of documents from the Web. Depending on the needs of the extensible crawler's applications, this crawl can be broad, focused, or both. For example, to provide applications with real-time information, the crawler might focus on real-time sources such as Twitter, Facebook, and popular news sites.

Web documents retrieved by the crawler are partitioned across **pods** for processing. A pod is a set of nodes that, in aggregate, contains all filters known to the system. Because documents are partitioned across pods, each document needs to be processed by a single pod; by increasing the number of pods within the system, the overall throughput of the system increases. **Document set partitioning** therefore facilitates the scaling up of the system's document processing rate.

Within each pod, the set of filters known to the extensible crawler is partitioned across the pod's nodes. **Filter set partitioning** is a form of sharding and it is used to address the memory or CPU limitations of an individual node. As more filters are added to the extensible crawler, additional nodes may need to be added to each pod, and the partitioning of filters across nodes might need adjustment. Because filters are partitioned across pod nodes, each document arriving at a pod needs to be distributed to each pod node for processing. Thus, the throughput of the pod is limited by the slowest node within the pod; this implies that load balancing of filters across pod nodes is crucially important to the overall system throughput.

Each node within the pod contains a subset of the system's filters. A naïve approach to processing a document on a node would involve looping over each filter on that node serially. Though this approach would work correctly, it would scale poorly as the number of filters grows. Instead, as we will discuss in Section 3.2, we trade memory for computation by using *filter indexing*, *relaxation*, and *staging* techniques; this allows us to evaluate a document against a node's full filter set with much faster than linear processing time.

If a document matches any filters on a node, the node notifies a match collector process running within the pod. The collector gathers all filters that match a given document and distributes match notifications to the appropriate web-crawler application clients.

Applications interact with the extensible crawler through two interfaces. They upload, delete, or modify filters in their filter sets with the filter management API. As well, they receive a stream of notification events corresponding to documents that match at least one of their filters through the notification API. We have considered but not yet experimented with other interfaces, such one for letting applications influence the pages that the web crawler visits.

3 XCrawler Design and Implementation

In this section, we describe the design and implementation of XCrawler, our prototype extensible crawler. XCrawler is implemented in Java and runs on a cluster of commodity multi-core x86 machines, connected by a gigabit switched network. Our primary optimization concern while building XCrawler was efficiently scaling to a large number of expressive filters.

In the rest of this section, we drill down into four aspects of XCrawler's design and implementation: the filter language it exposes to clients, how a node matches an incoming document against its filter set, how documents and filters are partitioned across pods and nodes, and how clients are notified about matches.

3.1 Filter language and document model

XCrawler's declarative filter language strikes a balance between expressiveness for the user and execution efficiency for the system. The filter language has four entities: attributes, operators, values, and expressions. There are two kinds of values: simple and composite. Simple values can be of several types, including byte sequences, strings, integers and boolean values. Composite values are tuples of values.

A document is tuple of attribute and values pairs. Attributes are named fields within a document; during crawling, each Web document is pre-processed to extract a static set of attributes and values. This set is passed to nodes and is referenced by filters during execution. Examples of a document's attribute-value pairs include its URL, the raw HTTP content retrieved by the crawler, certain HTTP headers like Content-Length or Content-Type, and if appropriate, structured text extracted from the raw content. To support sampling, we also provide a random number attribute whose per-document value is fixed at chosen when other attributes are extracted.

A user-provided filter is a predicate expression; if the expression evaluates to true against a document, then the filter matches the document. A predicate expression is either a boolean operator over a single document attribute, or a conjunct of predicate expressions. A boolean operator expression is an (attribute, operator, value) triple, and is represented in the form:

```
attribute.operator(value)
```

The filter language provides expensive operators such as substring and regular expression matching as well as simple operators like equalities and inequalities.

For example, a user could specify a search for the phrase "Barack Obama" in HTML files by specifying:

```
mimetype.equals("text/html") &
text.substring("Barack Obama")
```

Alternatively, the user could widen the set of acceptable documents by specifying a conjunction of multiple, less restrictive keyword substring filters.

```
mimetype.equals("text/html") &
text.substring("Barack") &
text.substring("Obama")
```

Though simple, this language is rich enough to support the applications outlined previously in Section 2. For example, our prototype Web malware detection application is implemented as a set of regular expression filters derived from the ClamAV virus and malware signature database.

3.2 Filter execution

When a newly crawled document is dispatched to a node, that node must match the document against its set of filters. As previously mentioned, a naïve approach to executing filters would be to iterate over them sequentially; unfortunately, the computational resources required for this approach would scale linearly with both the number of filters and the document crawl rate, which is severely limiting. Instead, we must find a way to optimize the execution of a set of filters.

To do this, we rely on three techniques. To maintain throughput while scaling up the number of filters on a node, we create memory-resident *indexes* for the attributes referenced by filters. Matching a document against an indexed filter set requires a small number of index lookups, rather than computation proportional to the number of filters. However, a high fidelity index might require too much memory, and constructing an efficient index over an attribute that supports a complex operator such as a regular expression might be intractable. In either case, we use *relaxation* to convert a filter into a form that is simpler or cheaper to index. For example, we can relax a regular expression filter into one that uses a conjunction of substring operators.

A relaxed filter is less precise than the full filter from which it was derived, potentially causing false positives. If the false positive rate is too high, we can feed the tentative matches from the index lookups into a second stage that executes filters precisely but at higher cost. By *staging* the execution of some filters, we regain higher precision while still controlling overall execution cost. However, if the false positive rate resulting from a relaxed filter is acceptably low, staging is not necessary, and all matches (including false positives) are sent to the client. Whether a false positive rate is acceptable depends on many factors, including the execution cost of staging in the extensible crawler, the bandwidth overhead of transmitting false positives to the client, and the cost to the client of handling false positives.

3.2.1 Indexing

Indexed filter execution requires the construction of an index for each attribute that a filter set references, and for each style of operator that is used on those attributes. For example, if a filter set uses a substring operator over the document body attribute, we build an Aho-Corasick multistring search trie [3] over the values specified by filters referencing that attribute. As another example, if a filter set uses numeric inequality operators over the document size attribute, we construct a binary search tree over the values specified by filters referencing that attribute.

Executing a document against a filter set requires looking up the document's attributes against all indexes to find potentially matching filters. For filters that contain a conjunction of predicate expressions, we could insert each expression into its appropriate index. Instead, we identify and index only the most selective predicate expression; if the filter survives this initial index lookup, we can either notify the client immediately and risk false positives or use staging (discussed in Section 3.2.3) to evaluate potential matches more precisely.

Creating indexes lets us execute a large number of filters efficiently. Figure 2 compares the number of nodes that would be required in our XCrawler prototype to sustain a crawl rate of 100,000 documents per second, using either naïve filter execution or filter execution with in-

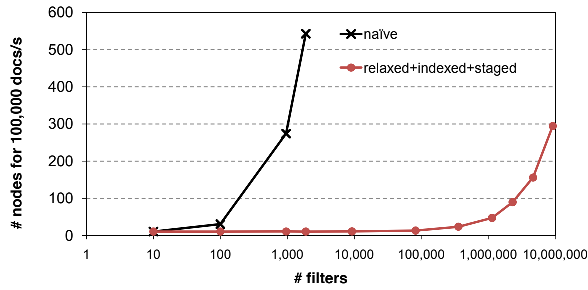


Figure 2: **Indexed filter execution.** This graph compares the number of nodes (machines) required for a crawl rate of 100,000 documents per second when using naïve filter execution and when using indexed filter execution, including relaxation and staging.

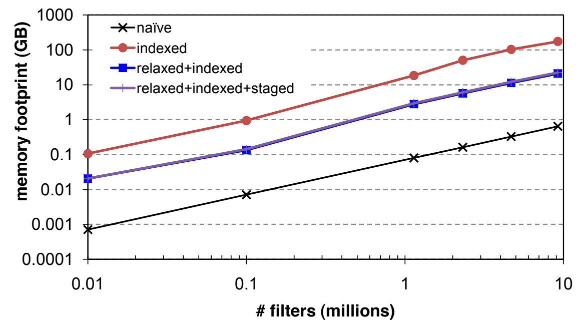


Figure 3: **Indexed and relaxed filter memory footprint.** This graph compares the memory footprint of substring filters when using four different execution strategies. The average filter length in the filter set was 130 bytes, and relaxation used 32 byte ngrams. Note that the *relaxed+indexed* and the *relaxed+indexed+staged* lines overlap on the graph.

dexing, relaxation, and staging enabled. The filter set used in this measurement are sentences extracted from Wikipedia articles; this emulates the workload of a copyright violation detection application. Our measurements were gathered on a small number of nodes, and projected upwards to larger numbers of nodes assuming linear scaleup in crawl rate with document set partitioning.

Our prototype runs on 8 core, 2 GHz Intel processors. When using indexing, relaxation, and staging, a node with 3GB of RAM is capable of storing approximately 400,000 filters of this workload, and can process documents at a rate of approximately 9,000 documents per second. To scale to 100,000 documents per second, we would need 12 pods, i.e., we must replicate the full filter set 12 times, and partition incoming documents across these replicas. To scale to 9,200,000 filters, we would need to partition the filter set across 24 machines with 3GB of RAM each. Thus, the final system configuration would have 12 pods, each with 24 nodes, for a total of 288 machines. If we installed more RAM on each machine, we would need commensurately fewer machines.

Even when including the additional cost of staging, indexed execution can provide several orders of magnitude better scaling characteristics than naïve execution as the number of filters grows. Note that the CPU is the bottleneck resource for execution in both cases, although with staged indexing, staging causes the CPUs to be primarily occupied with processing false positives from relaxed filters.

3.2.2 Relaxation

We potentially encounter two problems when using indexing: the memory footprint of indexes might be excessive, and it might be infeasible to index attributes or operators such as regular expressions or conjuncts. To cope with either problem, we use relaxation to convert a filter into a form that is less accurate but indexable.

As one example, consider copyright violation detection filters that contain sentences that should be searched for as substrings within documents. Instead of searching for the full sentence, filters can be relaxed to search for an ngram extracted from the sentence (e.g., a 16 byte character fragment). This would significantly reduce the size of the in-memory index.

There are many possible ngram relaxations for a specific string; the ideal relaxation would be just as selective as the full sentence, returning no false positives. Intuitively, shorter ngrams will tend to be less selective but more memory efficient. Less intuitively, different fragments extracted from the same string might have different selectivity. Consider the string `<a href="http://zyzzyva.com">`, and two possible 8-byte relaxations `<a href=` and `/zyzzyva`: the former would be much less selective than the latter. Given this, our prototype gathers run-time statistics on the hit rate of relaxed substring operations, identifies relaxations that have anomalously high hit rates, and selects alternative relaxations for them. If we cannot find a low hit rate relaxation, we ultimately reject the filter.

Relaxation also allows us to index operations that are not directly or efficiently indexable. Conjuncts are not directly indexable, but can be relaxed by picking a selective indexable subexpression. A match of this subexpression is not as precise as the full conjunction, but can eliminate a large portion of true negatives. Similarly, regular expressions could hypothetically be indexed by combining their automata, but combined automata tend to have exponentially large state requirements or high computational requirements [12, 23, 32]. Instead, if we can identify substrings that the regular expression implies must occur in an accepted document, we can relax the regular expression into a less selective but indexable substring. If a suitably selective substring cannot be identified from a given regular expression, that filter can be rejected when

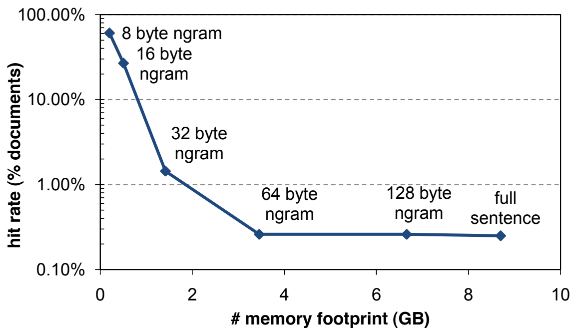


Figure 4: **Filter relaxation trade-off.** This graph illustrates the trade-off between memory footprint and false positive rates when using different degrees of relaxation.

the client application attempts to upload it.

Figure 3 compares the memory footprint of naïve, indexed, relaxed+indexed, and relaxed+indexed+staged filter execution. The filter set used in this measurement is the same as in Figure 2, namely sentences extracted from Wikipedia articles, averaging 130 characters in length. Relaxation consists of selecting a random 32 character substring from a sentence. The figure demonstrates that indexing imposes a large memory overhead relative to naïve execution, but that relaxation can substantially reduce this overhead.

Relaxation potentially introduces false positives. Figure 4 illustrates the trade-off between the memory footprint of filter execution and the hit rate, as the degree of relaxation used varies. With no relaxation, an indexed filter set of 400,000 Wikipedia sentences averaging 130 characters in length requires 8.7GB of memory and has a hit rate of 0.25% of Web documents. When relaxing these filters to 64 byte ngrams, the memory footprint is reduced to 3.5GB and the hit rate marginally climbs to 0.26% of documents. More aggressive relaxation causes a substantial increase in false positives. With 32 byte ngrams, the memory footprint is just 1.4GB, but the hit rate grows to 1.44% of documents: nearly four out of five hits are false positives.

3.2.3 Staging

If a relaxed filter causes too many false positives, we can use staging to eliminate them at the cost of additional computation. More specifically, if a filter is marked for staging, any document that matches the relaxed version of the filter (a *partial hit*) is subsequently executed against the full version of that filter. Thus, the first stage of filter execution consists of index lookups, while the second stage of execution iterates through the partial hits identified by the first stage.

The second stage of execution does not benefit from indexing or relaxation. Accordingly, if the partial hit rate in the first stage is too high, the second stage of execution has the potential to dominate computation time and limit the throughput of the system. As well, any filter that is staged requires the full version of the filter to be stored in memory. Staging eliminates false positives, but has both a computational and memory cost.

3.3 Partitioning

As with most cluster-based services, the extensible crawler achieves cost-efficient scaling by partitioning its work across inexpensive commodity machines. Our workload consists of two components: documents that continuously arrive from the crawler and filters that are periodically uploaded or updated by client applications. To scale, the extensible crawler must find an intelligent partitioning of both documents and filters across machines.

3.3.1 Document set partitioning

Our first strategy, which we call *document set partitioning*, is used to increase the overall throughput of the extensible crawler. As previously described, we define a **pod** as a set of nodes that, in aggregate, contains all filters known to the system. Thus, each pod contains all information necessary to process a document against a filter set. To increase the throughput of the system, we can add a pod, essentially replicating the configuration of existing pods onto a new set of machines.

Incoming documents are partitioned across pods, and consequently, each document must be routed to a single pod. Since each document is processed independently of others, no interaction between pods is necessary in the common case. Document set partitioning thus leads to an embarrassingly parallel workload, and linear scalability. Our implementation monitors the load of each pod, periodically adjusting the fraction of incoming documents directed to each pod to alleviate hot spots.

3.3.2 Filter set partitioning

Our second strategy, which we call *filter set partitioning*, is used to address the memory and CPU limitations of an individual node within a pod. Filter set partitioning is analogous to sharding, declustering, and horizontal partitioning. Since indexing operations are memory intensive, any given node can only index a bounded number of filters. Thus, as we scale up the number of filters in the system, we are forced to partition filters across the nodes within a pod.

Our system supports complex filters composed of a conjunction of predicate expressions. In principle, we could decompose filters into predicates, and partition predicates across nodes. In practice, our implementation uses the simpler approach of partitioning entire filters. As such, a document that arrives at a node can be fully

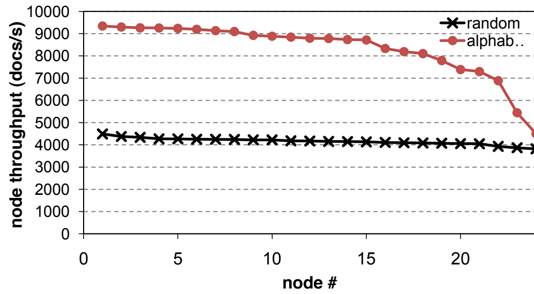


Figure 5: **Node throughput.** This (sorted) graph shows the maximum throughput each node within a pod is capable of sustaining under two different policies: random filter placement, and alphabetic filter placement.

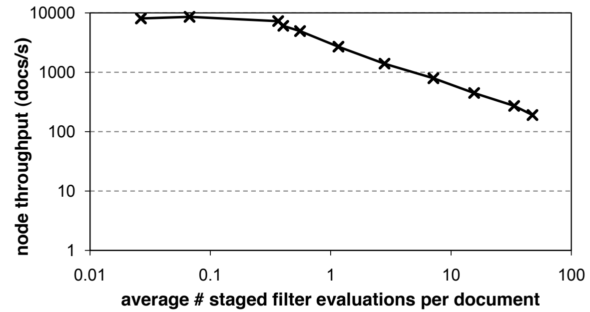


Figure 6: **Staged evaluations vs. throughput.** This graph shows the effect of increasing partial hit rate of a filter set within a node on the maximum document throughput that node is capable of processing.

evaluated against each filter on that node without requiring any cross-node interactions.

Since a document must be evaluated against all filters known by the system, each document arriving at a pod must be transmitted to and evaluated by each node within the pod. Because of this, the document throughput that a pod can sustain is limited by the throughput of the slowest node within the pod.

Two issues substantially affect node and pod throughput. First, a filter partitioning policy that is aware of the indexing algorithms used by nodes can tune the placement of filters to drive up the efficiency and throughput of all nodes. Second, some filters are more expensive to process than others. Particularly expensive filters can induce load imbalances across nodes, driving down overall pod throughput.

Figure 5 illustrates these effects. Using the same Wikipedia workload as before, this graph illustrates the maximum document throughput that each node within a pod of 24 machines is capable of sustaining, under two different filter set partitioning policies. The first policy, *random*, randomly places each filter on a node, while the second policy, *alphabetic*, sorts the substring filters alphabetically by their most selective ngram relaxation. By sorting alphabetically, the second policy causes ngrams that share prefixes to end up on the same node, improving both the memory and computation efficiency of the Aho-Corasick index. The random policy achieves good load balancing but suffers from lower average throughput than alphabetic. Alphabetic exhibits higher average throughput but suffers from load imbalance. From our measurements using the Wikipedia filter set, a 5 million filter index using random placement requires 13.9GB of memory, while a 5 million filter index using alphabetic placement requires 12.1GB, a reduction of 13%.

In Figure 6, we measure the relationship between the number of naïve evaluations that must be executed per document when using staged relaxation and the throughput a node can sustain. As the number of naïve executions increases, throughput begins to drop, until eventually the node spends most of its time performing these evaluations. In practice, the number of naïve evaluations can increase for two reasons. First, a given relaxation can be shared by many independent filters. If the relaxation matches, all associated filters must be executed fully. Second, a given relaxation might match a larger-than-usual fraction of incoming documents. In Section 4.2, we further quantify these two effects and propose strategies for mitigating them.

Given all of these issues, our implementation takes the following strategy to partition filters. Initially, an index-aware partitioning is chosen; for example, alphabetically-sorted prefix grouping is used for substring filters. Filters are packed onto nodes until memory is exhausted. Over time, the load of nodes within each pod is monitored. If load imbalances appear within a pod, then groups of filters are moved from slow nodes to faster nodes to rectify the imbalance. Note that moving filters from one node to another requires the indexes on both nodes to be recomputed. The expense of doing this bounds how often we can afford to rebalance.

A final consideration is that the filter set of an extensible crawler changes over time as new filters are added and existing filters are modified or removed. We currently take a simple approach to dealing with filter set changes: newly added or modified filters are accumulate in "overflow" nodes within each pod and are initially executed naïvely without the benefit of indexing. We then take a generational approach to re-indexing and re-partitioning filters: newly added and modified filters that appear to be stable are periodically incorporated into the non-overflow nodes.

3.4 Additional implementation details

The data path of the extensible crawler starts as documents are dispatched from a web crawler into our system and ends as matching documents are collected from workers and transmitted to client crawler applications (see Figure 1). Our current prototype does not fully ex-

plore the design and implementation issues of either the dispatching or collection components.

In our experiments, we use the open source Nutch spider to crawl the web, but we modified it to store documents locally within each crawler node's filesytem rather than storing them within a distributed Hadoop filesystem. We implemented a parallel dispatcher that runs on each crawler node. Each dispatcher process partitions documents across pods, replicates documents across pod nodes, and uses backpressure from nodes to decide the rate at which documents are sent to each pod. Each pod node keeps local statistics about filter matching rates, annotates matching documents with a list of filters that matched, and forwards matching documents to one of a static set of collection nodes.

An interesting configuration problem concerns balancing the CPU, memory, and network capacities of nodes within the system. We ensure that all nodes within a pod are homogeneous. As well, we have provisioned each node to ensure that the network capacity of nodes is not a system bottleneck. Doing so required provisioning each filter processing node with two 1-gigabit NICs. To take advantage of multiple cores, our filter processing nodes use two threads per core to process documents against indexes concurrently. As well, we use one thread per NIC to pull documents from the network and place them in a queue to be dispatched to filter processing threads. We can add additional memory to each node until the cost of additional memory becomes prohibitive. Currently, our filter processing nodes have 3GB of RAM, allowing each of them to store approximately a half-million filters.

Within the extensible crawler itself, all data flows through memory; no disk operations are required. Most memory is dedicated to filter index structures, but some memory is used to queue documents for processing and to store temporary data generated when matching a document against an index or an individual filter.

We have not yet explored fault tolerance issues. Our prototype currently ignores individual node failures and does not attempt to detect or recover from network or switch failures. If a filter node fails in our current implementation, documents arriving at the associated pod will fail to be matched against filters that resided on that node. Note that our overall application semantics are best effort: we do not (yet) make any guarantees to client applications about when any specific web page is crawled. We anticipate that this will simplify fault tolerance issues, since it is difficult for clients to distinguish between failures in our system and the case that a page has not yet been crawled. Adding fault tolerance and strengthening our service guarantees is a potentially challenging future engineering topic, but we do not anticipate needing to invent fundamentally novel mechanisms.

3.5 Future considerations

There are several interesting design and implementation avenues for the extensible crawler. Though they are beyond the scope of this paper, it is worth briefly mentioning a few of them. Our system currently only indexes textual documents; in the future, it would be interesting to consider the impact of richer media types (such as images, videos, or flash content) on the design of the filter language and on our indexing and execution strategy. We currently consider the crawler itself to be a black box, but given that clients already specify content of interest to them, it might be beneficial to allow clients to focus the crawler on certain areas of the Web of particular interest. Finally, we could imagine integrating other streams of information into our system besides documents gathered from a Web crawler, such as real-time "firehoses" produced by systems such as Twitter.

4 Evaluation

In this section, we describe experiments that explore the performance of the extensible crawler, we investigate the effect of different filter partitioning policies. As well, we demonstrate the need to identify and reject non-selective filters. Finally, we present early experience with three prototype Web crawler applications.

All of our experiments are run on a cluster of 8-core, 2GHz Intel Xeon machines with 3GB of RAM, dual gigabit NICs, and a 500 GB 7200-RPM Barracuda ES SATA hard drive. Our systems are configured to run 32bit Linux kernel version 2.6.22.9-91.fc7, and to use Sun's 23 bit JVM version 1.6.0_12 in server mode. Unless stated otherwise, the filter workload for our experiments consists of 9,204,600 unique sentences extracted from Wikipedia; experiments with relaxation and staging used 32 byte prefix ngrams extracted from the filter sentences.

For our performance oriented experiments, we gathered a 3,349,044 Web document crawl set on August 24th, 2008 using the Nutch crawler and pages from the DMOZ open directory project as our crawl seed. So that our experiments were repeatable, when testing the performance of the extensible crawler we used on a custom tool to stream this document set at high throughput, rather than re-crawling the Web. Of the 3,349,044 documents in our crawl set, 2,682,590 contained textual content, including HTML and PDF files; the rest contain binary content, including images and executables. Our extensible crawler prototype does not yet notify wide-area clients about matching documents; instead, we gather statistic about document matches, but drop the matching documents instead of transmitting them.

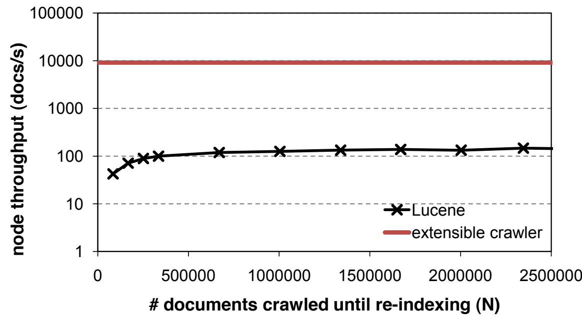


Figure 7: **Lucene vs. extensible crawler.** This graph compares the document processing rates of a single-node extensible crawler and a single-node Lucene search engine. The x-axis displays the number of documents crawled between reconstructions of the Lucene index. Note that the y-axis is logarithmically scaled.

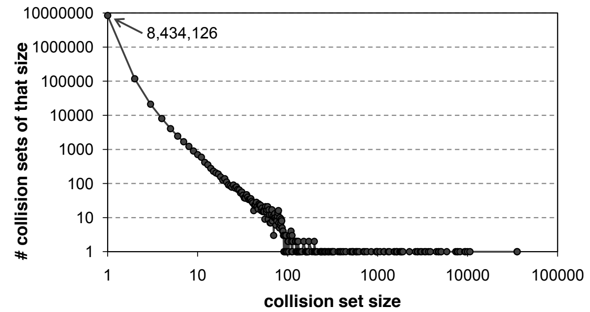


Figure 8: **Collision set size.** This histogram shows the distribution of collision set sizes when using 32-byte ngrams over the Wikipedia filter set.

4.1 Nutch vs. the extensible crawler

In Section 2.1, we described architectural, workload, and expected performance differences between the extensible crawler and an alternative implementation of the service based on a conventional search engine. To demonstrate these differences quantitatively, we ran a series of experiments directly comparing the performance of our prototype to an alternative implementation based on the Lucene search engine, version 2.1.0 [7].

The construction of a Lucene-based search index is typically performed as part of a Nutch map-reduce pipeline that crawls web pages, stores them in the HDFS distributed filesystem, builds and stores an index in HDFS, and then services queries by reading index entries from HDFS. To make the comparison of Lucene to our prototype more fair, we eliminated overheads introduced by HDFS and map-reduce by modifying the system to store crawled pages and indexes in nodes' local filesystems. Similarly, to eliminate variation introduced by the wide-area Internet, we spooled our pre-crawled Web page data set to Lucene's indexer or to the extensible crawler over the network.

The search engine implementation works by periodically constructing an index based on the N most recently crawled web pages; after constructing the index and partitioning it across nodes, each node evaluates the full filter set against its index fragment. The implementation uses one thread per core to evaluate filters. By increasing N, the implementation indexes less frequently, reducing overhead, but suffers from a larger latency between the downloading of a page by the crawler and the evaluation of the filter set against that page. In contrast, the extensible crawler implementation constructs and index over its filters once, and then continuously evaluates pages against that index.

In Figure 7, we compare the single node throughput of the two implementations, with 400,000 filters from the Wikipedia workload. Note that the x-axis corresponds to N, but this parameter only applies to the Lucene crawler. The extensible crawler implementation has nearly two orders of magnitude better performance than Lucene; this is primarily due to the fact that Lucene must service queries against its disk-based document index, while the extensible crawler's filter index is served out of memory. As well, the Lucene implementation is only able to achieve asymptotic performance if it indexes batches of $N > 1,000,000$ documents.

Our head-to-head comparison is admittedly still unfair, since Lucene was not optimized for fast, incremental, memory-based indexing. Also, we could conceivably bridge the gap between the two implementations by using SSD drives instead of spinning platters to store and serve Lucene indexes. However, our comparison serves to demonstrate some of the design tensions between conventional search engines and extensible crawlers.

4.2 Filter partitioning and blacklisting

As mentioned in Section 3.3.2, two different aspects of a filter set contribute to load imbalances between otherwise identical machines: first, a specific relaxation might be shared by many different filters, causing a partial hit to result in commensurately many naïve filter executions, and second, a given relaxation might match a large number of documents, also causing a large number of naïve filter executions. We now quantify these effects.

We call the a set of filters that share an identical relaxation a *collision set*. A collision set of size 1 implies the associated filter's relaxation is unique, while a collision set of size N implies that N filters share a specific relaxation. In Figure 8, we show the distribution of collision set sizes when using a 32-byte prefix relaxation of the Wikipedia filter set. The majority of filters (8,434,126 out of 9,204,600) have a unique relaxation, but some relaxations collide with many filters. For example, the largest collision set size was 35,585 filters. These filters all shared the prefix "The median income for a house-

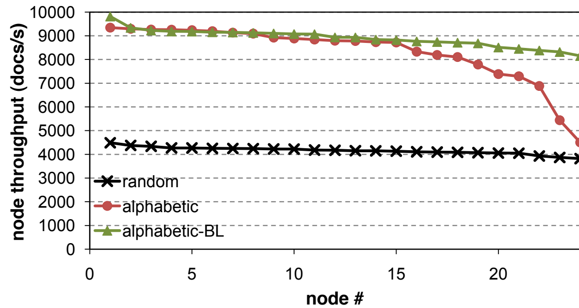


Figure 9: **Node throughput.** This (sorted) graph shows the maximum throughput each node within a pod is capable of sustaining under three different policies: random filter placement, alphabetic filter placement, and alphabetic filter placement with blacklisting.

hold in the"; this sentence is used in many Wikipedia articles describing income in cities, counties, and other population centers. If a document matches against this relaxation, the extensible crawler would need to naïvely execute all 35,585 filters.

Along a similar vein, some filter relaxations will match a larger-than-usual fraction of documents. One notably egregious example from our Web malware detection application was a filter whose relaxation contained was the 32-character sequence `<meta http-equiv="Content-Type">`. Unsurprisingly, a very large fraction of Web pages contain this substring!

To deal with these two sources of load imbalance, our implementation blacklists specific relaxations. If our implementation notices a collision set containing more than 100 filters, we blacklist the associated relaxation, and compute alternate relaxations for those filters. In the case of our Wikipedia filter set, this required modifying the relaxation of only 145 filters. As well, if our implementation notices that a particular relaxation has an abnormally high document partial hit rate, that "hot" relaxation is blacklisted and new filter relaxations are chosen.

Blacklisting rectifies these two sources of load imbalance. Figure 9 revisits the experiment previously illustrated in Figure 5, but with a new line that shows the effect of blacklisting on the distribution of document processing throughputs across the nodes in our cluster. Without blacklisting, alphabetic filter placement demonstrates significant imbalance. With blacklisting, the majority of the load imbalance is removed, and the slowest node is only 17% slower than the fastest node.

4.3 Experiences with Web crawler applications

To date, we have prototyped three Web crawler applications: vanity alters that detect pages containing a user's name, a copyright detection application that finds Web objects that match ClamAV's malware signature database, and a copyright violation detection service that looks for pages containing copies of Reuters or Wikipedia articles. Table 3 summarizes the high-level features of these applications and their filter workloads; we now discuss each in turn, relating additional details and anecdotes.

		Vanity alerts	Copyright violation	Malware detection
# filters		10,622	251,647	3,128
relaxation only	doc hit rate	68.98%	0.664%	45.38%
	false +ves per doc	15.76	0.0386	0.851
	throughput (docs/s)	7,244	8,535	8,534
	# machines needed	13.80	11.72	11.72
with relaxation and staging	doc hit rate	13.1%	0.016%	0.009%
	throughput (docs/s)	592	8,229	6,354
	# machines needed	168.92	12.15	15.74

Table 3: **Web crawler application features and performance.** This table summarizes the high-level workload and performance features of our three prototype Web crawler applications.

4.3.1 Vanity alerts

For our vanity filter application, we authored 10,622 filters based on names of university faculty and students. Our filters were constructed as regular expressions of the form ``` ``first.{1,20}last'' ```, i.e., the user's first name followed by their last name, with the constraint of no more than 20 characters separating the two parts. The filters were first relaxed into a conjunct of substring 32-grams, and from there the longest substring conjunct was selected as the final relaxation of the filter.

This filter set matched against 13.1% of documents crawled. This application had a modest number of filters, but its filter set nonetheless matched against a large fraction of Web pages, violating our assumption that a crawler application should have highly selective filters. Moreover, when using relaxed filters without, there were many additional false partial hits (an average of 15.76 per document, and an overall document hit rate of 69%). Most false hits were due to names that are contained in commonly found words, such as Tran, Chang, or Park.

The lack of high selectivity of its filter set leads us to conclude that this application is not a good candidate for the extensible crawler. If millions of users were to use this service, most Web pages would likely match and need to be delivered to the crawler application.

4.3.2 Copyright violation detection

For our second application, we prototyped a copyright violation detection service. We evaluated this application by constructing a set of 251,647 filters based on 30,534 AP and Reuters news articles appearing between July and October of 2008. Each filter was a single sentence extracted from an article, but we extracted multiple fil-

ters from each article. We evaluated the resulting filter set against a crawl of 3.68 million pages.

Overall, 590 crawled documents (0.016%) matched against the AP/Reuters filter set, and 619 filters (0.028% of the filter set) were responsible for these matches. We manually determined that most matching documents that matched were original news articles or blogger pages that quoted sections of articles with attribution. We did find some sites that appeared to contain unauthorized copies of entire news stories, and some sites that plagiarized news stories by integrating the story body but replacing the author's byline.

If a document hit, it tended to hit against a single filter (50% of document hits were for a single filter). A smaller number of documents hit against many sentences (13% of documents matched against more than 6 filters). Documents that matched against many filters tended to contain full copies of the original news story, while documents that match a single sentence tended to contain boilerplate prose, such as a specific author's byline, legal disclosures, or common phrases such as "The officials spoke on condition of anonymity because they weren't authorized to release the information."

4.3.3 Web malware detection

The final application we prototyped was Web malware detection. We extracted 3,128 text-centric regular expressions from the February 20, 2009 release of the ClamAV open-source malware signature database. Because many of these signatures were designed to match malicious JavaScript or HTML, some of their relaxations contain commonly occurring substrings, such as <a href=‘‘http://’’. As a result, blacklisting was a particularly important optimization for this workload; the system was always successful at finding suitably selective relaxations of each filter.

Overall, this filter set matched 342 pages from the same crawl of 3.68 million pages, with an overall rate of 0.009%. The majority of hits (229) were for two similar signatures that capture obfuscated JavaScript code that emits an iframe in the parent page. We examined all of the pages that matched this signature; in each case, the iframe contained links to other pages that are known to contain malicious scripts. Most of the matching pages appeared to be legitimate business sites that had been compromised. We also found several pages that matched a ClamAV signature designed to detect Web bugs.

In addition to genuinely malicious Web pages, we found a handful of pages that were application-level false positives, i.e., they correctly matched a ClamAV filter, but the page did not contain the intended attack. Some of these application-level false positives contained blog entries discussing virulent spam, and the virulent spam itself was represented in the ClamAV database.

5 Related work

The extensible crawler is related to several classes of systems: Web crawlers and search engines, publish-subscribe systems, packet filtering engines, parallel and streaming databases, and scalable Internet content syndication protocols. We discuss each in turn.

Web crawlers and search engines. The engineering issues of high-throughput Web crawlers are complex but well understood [21]. Modern Web crawlers can retrieve thousands of Web pages per second per machine. Our work leverages existing crawlers, treating them as a black box from which we obtain a high throughput document stream. The Mercator project explored the design of an extensible crawler [20], though Mercator's notion of extensibility is different than ours: Mercator has well-defined APIs that simplify the job of adding new modules that extend the crawler's set of network protocols or type-specific document processors. Our extensible crawler permits remote third parties to dynamically insert new filters into the crawling pipeline.

Modern Web search engines require complex engineering, but the basic architecture of a scalable search engine has been understood for more than decade [8]. Our extensible crawler is similar to a search engine, but inverted, in that we index queries rather than documents. As well, we focus on in-memory indexing for throughput. Cho and Rajagopalan described a technique for supporting fast indexing of regular expressions by reducing them to ngrams [12]; our notion of filter relaxation is a generalization of their approach.

Though the service is now discontinued, Amazon.com offered programmatic search access to a 300TB archive containing 4 billion pages crawled by Alexa Internet [5] and updated daily. By default, access was restricted to queries over a fixed set of search fields, however, customers could pay to re-index the full data set over custom fields. In contrast, the extensible crawler permits customers to write custom filters over any attribute supported by our document extractors, and since we index filters rather than pages, our filters are evaluated in real-time, at the moment a page is crawled.

Publish-subscribe systems. The extensible crawler can be thought of as a content-based publish-subscribe system [22] designed and optimized for a real-time Web crawling workload. Content-based pub-sub systems have been explored at depth, including in the Gryphon [2], Siena [9], Elvin [31], and Le Subscribe [16, 27] projects. Many of these projects explore the trade-off between filter expressiveness and evaluation efficiency, though most have a wide-area, distributed event notification context in mind. Le Subscribe is perhaps closest to our own system; their language is also a conjunction of predicates, and like us, they index predicates in main-memory for scal-

able, efficient evaluation. In contrast to these previous projects, our work explores in depth the partitioning of documents and filters across machines, the suitability of our expression language for Web crawling applications, the impact of disproportionately high hit rate filters, and evaluates several prototype applications.

Web-based syndication protocols, such as RSS and Atom, permit Web clients to poll servers to receive feeds of new articles or document elements. Cloud-based aggregation and push notification services such as rssCloud and PubSubHubbub allow clients to register interest in feeds and receive notifications when updates occur, relieving servers from pull-induced overload. These services are roughly equivalent to channel-based pub-sub systems, whereas the extensible crawler is more equivalent to a content-based system.

The Google alerts system [18] allows users to specify standing search queries to be evaluated against Google's search index. Google alerts periodically emails users newly discovered search results relevant to their queries. Alerts uses two different approaches to gather new results: it periodically re-executes queries against the search engine and filters previously returned results, and it continually matches incoming documents against the body of standing user queries. This second approach has similarities to the extensible crawler, though details of Google alert's architecture, workload, performance, and scalability have not been publicly disclosed, preventing an in-depth technical comparison.

Cobra [29] perhaps most similar to our system. Cobra is a distributed system that crawls RSS feeds, evaluates articles against user-supplied filters, and uses reflectors to distributed matching articles to interested users. Both Cobra and the extensible crawler benefit from a filter language design to facilitate indexing. Cobra focused on issues of distribution, provisioning, and network-aware clustering, whereas our work focuses on a single-cluster implementation, efficiency through filter relaxation and staging, and scalability through document and filter set partitioning. As well, Cobra was oriented towards scalable search and aggregation of Web feeds, whereas the extensible crawler provides a platform for more widely varied crawling applications, such as malware and copyright violation detection.

Packet filters and NIDS. Packet filters and network intrusion detection systems (NIDS) have similar challenges as the extensible crawler: both classes of systems must process a large number of filters over a high bandwidth stream of unstructured data with low latency. The BSD packet filter allowed control-flow graph filters to be compiled down to an abstract filtering machine, and executed safely and efficiently in an OS kernel [25]. Packet filtering systems have also confronted the problem of efficiently supporting more expressive filters, while preventing state space explosion when representing large filter sets as DFAs or NFAs [23, 32]. Like an extensible crawler, packet filtering systems suffer from the problem of normalizing documents content before matching against filters [30], and of providing additional execution context so that byte-stream filters can take advantage of higher-level semantic information [34]. Our system can benefit from the many recent advances in this class of system, though our applications require orders of magnitude more filters and therefore a more scalable implementation. As well, our application domain is more robust against false positives.

Databases and SDIs. The extensible crawler shares some design considerations, optimizations, and implementation techniques with parallel databases such as Bubba [13] and Gamma [14], in particular our need to partition filters (queries) and documents (records) over machines, and our focus on high selectivity as a path to efficiency. Our workload tends to require many more concurrent filters, but does not provide the same expressiveness as SQL queries. We also have commonalities with streaming database systems and continuous query processors [1, 10, 11], in that both systems execute standing queries against an infinite stream of data. However, streaming database systems tend to focus on semantic issues of queries over limited time windows, particularly when considering joins and aggregation queries, while we focus on scalability and Web crawling applications.

Many databases support the notion of triggers that fire when matching records are added to the database. Prior work has examined indexing techniques for efficiently supporting a large number of such triggers [19].

Selective Dissemination of Information (SDI) systems [35], including those that provide scalable, efficient filtering of XML documents [4], share our goal of executing a large number filters over semi-structured documents, and rely on the same insight of indexing queries to match against individual documents. These systems tend to have more complex indexing schemes, but have not yet been targeted at the scale, throughput, or application domain of the extensible crawler.

6 Conclusions

This paper described the design, prototype implementation, and evaluation of the *extensible crawler*, a service that crawls the Web on behalf of its many client applications. Clients extend the crawler by injecting filters that identify pages of interest to them. The crawler continuously fetches a stream of pages from the Web, simultaneously executes all clients' filters against that stream, and returns to each client those pages selected by its filter set.

An extensible crawler provides several benefits. It relieves clients of the need to operate and manage their

own private crawler, greatly reducing a client's bandwidth and computational needs when locating pages of interest. It is efficient in terms of Internet resources: a crawler queries a single stream of Web pages on behalf of many clients. It also has the potential for crawling highly dynamic Web pages or real-time sources of information, notifying clients quickly when new or interesting content appears.

The evaluation of XCrawler, our early prototype system, focused on scaling issues with respect to its number of filters and crawl rate. Using techniques from related work, we showed how we can support rich, expressive filters using relaxation and staging techniques. As well, we used microbenchmarks and experiments with application workloads to quantify the impact of load balancing policies and confirm the practicality of our ideas. Overall, we believe that the low-latency, high selectivity, and scalable nature of our system makes it a promising platform for many applications.

Acknowledgments

We thank Tanya Bragin for her early contributions, Raphael Hoffman for his help in gathering the Wikipedia filter set, and Paul Gauthier and David Richardson for their helpful feedback on drafts of this paper. We also thank our anonymous reviews and our shepherd, Stefan Savage, for their guidance. This work was supported in part by an NDSEG Fellowship, by the National Science Foundation under grants CNS-0132817, CNS-0430477 and CNS-0627367, by the Torode Family Endowed Career Development Professorship, by the Wissna-Slivka Chair, and by gifts from Nortel Networks and Intel Corporation.

References

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[2] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '99)*, Atlanta, GA, 1999.

[3] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[4] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB '00)*, San Francisco, CA, 2000.

[5] Amazon Web Services. Alexa web search platform. `http://www.amazon.com/b?ie=UTF8&node=16265721`.

[6] E. Amitay, D. Carmel, M. Herscovici, R. Lempel, and A. Soffer. Trend detection through temporal link analysis. *Journal of the American Society for Information Science and Technology*, 55(14):1270–1281, December 2004.

[7] Apache Software Foundation. Apache lucene. `http://lucene.apache.org/`.

[8] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the Seventh International Conference on the World Wide Web (WWW7)*, Brisbane, Australia, 1998.

[9] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Achieving scalability and expressiveness in an Internet-scale event notification service. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '00)*, Portland, OR, 2000.

[10] J. Chen, D. J. Dewitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for Internet databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, Dallas, TX, May 2000.

[11] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR '03)*, Asilomar, CA, January 2003.

[12] J. Cho and S. Rajagopalan. A fast regular expression indexing engine. In *Proceedings of the 18th International Conference on Data Engineering (ICDE '02)*, San Jose, CA, February 2002.

[13] G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data placement in Bubba. *ACM SIGMOD Record*, 17(3):99–108, 1988.

[14] D. J. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, March 1990.

[15] O. Etzioni, M. Cafarella, D. Downey, S. Kok, A.-M. Popescu, T. Shaked, S. Soderland, D. S. Weld, and A. Yates. Web-scale information extraction in KnowItAll. In *Proceedings of the 13th International Conference on the World Wide Web*, New York, NY, May 2004.

[16] F. Fabret, A. Jacobsen, F. Llirbat, J. Pereira, K. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, Santa Barbara, CA, May 2001.

[17] R. Geambasu, S. D. Gribble, and H. M. Levy. CloudViews: Communal data sharing in public clouds. In *Proceedings of the Workshop on Hot Topics in Cloud Computing (HotCloud '09)*, San Diego, CA, June 1999.

[18] Google. Google alerts. `http://www.google.com/alerts`.

[19] E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. Park, and A. Vernon. Scalable trigger processing. In *Proceedings of the 15th International Conference on Data Engineering*, Sydney, Austrialia, March 1999.

[20] A. Heydon and M. Najork. Mercator: A scalable, extensible web crawler. *World Wide Web*, 2(4):219–229, 1999.

[21] H.-T. Lee, D. Leonard, X. Wang, and D. Loguinov. IRLbot: Scaling to 6 billion pages and beyond. In *Proceedings of the 17th International World Wide Web Conference*, Beijing, China, April 2008.

[22] Y. Liu and B. Plale. Survey of publish subscribe event systems. Technical Report TR574, Indiana University, May 2003.

[23] A. Majumder, R. Rastogi, and S. Vanama. Scalable regular expression matching on data streams. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, Vancouver, Canada, June 2008.

[24] McAfee, Inc. McAfee SiteAdvisor. `http://www.siteadvisor.com/`.

[25] S. McCanne and V. Jacobson. The BSD packet filter: a new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference*, San Diego, California, January 1993.

[26] A. Moshchuk, T. Bragin, S. Gribble, and H. Levy. A crawler-based study of spyware on the Web. In *Proceedings of the 13th Annual Network and Distributed Systems Security Symposium (NDSS '06)*, San Diego, CA, February 2006.

[27] J. Pereira, F. Fabret, F. Llirbat, and D. Shasha. Efficient matching for web-based publish/subscribe systems. In *Proceedings of the 7th International Conference on Cooperative Information Systems (CoopIS 2000)*, Eilat, Israel, September 2000.

[28] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iFRAMEs point to us. In *Proceedings of the 17th USENIX Security Symposium*, San Jose, CA, July 2008.

[29] I. Rose, R. Murty, P. Pietzuch, J. Ledlie, M. Roussopoulos, and M. Welsh. Cobra: Content-based filtering and aggregation of blogs and RSS feeds. In *Proceedings of the 4th USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2007)*, Cambridge, MA, April 2007.

[30] S. Rubin, S. Jha, and B. P. Miller. Protomatching network traffic for high throughput network intrusion detection. In *Proceedings of the 13th ACM conference on Computer and Communications Security (CCS '06)*, October 2006.

[31] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. Content based routing with Elvin4. In *Proceedings of the 2000 AUUG Annual Conference (AUUG2K)*, Canberra, Australia, June 2000.

[32] R. Smith, C. Estan, and S. Jha. XFA: Faster signature matching with extended automata. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (Oakland '08)*, Oakland, CA, May 2008.

[33] N. Snavely, S. M. Seitz, and R. Szeliski. Photo tourism: Exploring photo collections in 3D. In *Proceedings of the 33rd International Conference an Exhibition on Computer Graphics and Interactive Techniques (SIGGRAPH '06)*, Boston, MA, July 2006.

[34] R. Sommer and V. Paxson. Enhancing byte-level network intrusion detection signatures with context. In *Proceedings of the 10th ACM conference on Computer and Communications Security (CCS '03)*, October 2003.

[35] T. W. Yan and H. Garcia-Molina. Index structures for selective dissemination of information under the boolean model. *ACM Transactions on Database Systems*, 19(2):332–364, June 1994.

Prophecy: Using History for High-Throughput Fault Tolerance

Siddhartha Sen, Wyatt Lloyd, and Michael J. Freedman
Princeton University

Abstract

Byzantine fault-tolerant (BFT) replication has enjoyed a series of performance improvements, but remains costly due to its replicated work. We eliminate this cost for read-mostly workloads through Prophecy, a system that interposes itself between clients and any replicated service. At Prophecy's core is a trusted *sketcher* component, designed to extend the semi-trusted load balancer that mediates access to an Internet service. The sketcher performs fast, load-balanced reads when results are historically consistent, and slow, replicated reads otherwise. Despite its simplicity, Prophecy provides a new form of consistency called *delay-once consistency*. Along the way, we derive a distributed variant of Prophecy that achieves the same consistency but without any trusted components.

A prototype implementation demonstrates Prophecy's high throughput compared to BFT systems. We also describe and evaluate Prophecy's ability to scale-out to support large replica groups or multiple replica groups. As Prophecy is most effective when state updates are rare, we finally present a measurement study of popular websites that demonstrates a large proportion of static data.

1 Introduction

Replication techniques are now the norm in large-scale Internet services, in order to achieve both reliability and scalability. However, leveraging active agreement to *mask* failures, whether to handle fail-stop behavior [41, 50] or fully malicious (Byzantine) failures [42], is not yet widely used. There is some movement in this direction from industry—such as Google's Chubby [10] and Yahoo!'s Zookeeper [66] coordination services, based on Paxos [41]—but both are used to manage infrastructure, not directly mask failures in customer-facing services.

And yet non-fail-stop failures in customer-facing services continue to occur, much to the chagrin and concern of system operators. Failures may arise from malicious break-ins, but they also may occur simply from system misconfigurations: Facebook leaking source code due to *one* misconfigured server [60], or Flickr mixing up returned images due to *one* improper cache server [24]. In fact, both of these examples could have been prevented through redundancy and agreement, without requiring full N-version programming [8]. The perceived need for systems robust to Byzantine faults—a superset of misconfigurations and Heisenbugs—has spurned almost a cottage industry on improving performance results of Byzantine fault tolerant (BFT) algorithms [1, 6, 12, 17, 30, 37, 38, 56, 62, 64, 65, 67].

While the latency of recent BFT algorithms has approached that of unreplicated reads to individual servers [15, 38, 64], the throughput of such systems falls far short. This is simple math: a minimum of four replicas [12] (or sometimes even six [1]) are required to tolerate one faulty replica, at least three of which must participate in each operation. For datacenters in the thousands or tens of thousands of servers, requiring four times as many servers without increasing throughput may be a non-starter. Even services that already replicate their data, such as the Google File System [25], would see their throughput drop significantly when using BFT agreement.

But if the replication cost of BFT is provably necessary [9], something has to give. One might view our work as a thought experiment that explores the potential benefit of placing a small amount of trusted software or hardware in front of a replicated service. After all, wide-area client access to an Internet service is typically mediated by some middlebox, which is then at least trusted to provide access to the service. Further, a small and simple trusted component may be less vulnerable to problems such as misconfigurations or Heisenbugs. And by treating the back-end service as an abstract entity that exposes a limited interface, this simple device may be able to interact with both complex and varied services. Our implementation of such a device has less than 3000 lines of code.

Barring such a solution, most system designers opt either for cheaper techniques (to avoid the costs of state machine replication) or more flexible techniques (to ensure service availability under heavy failures or partitions). The design philosophies of Amazon's Dynamo [18], GFS [25], and other systems [20, 23, 61] embrace this perspective, providing only eventually-consistent storage. On the other hand, the tension between these competing goals persists, with some systems in industry re-introducing stronger consistency properties. Examples include timeline consistency in Yahoo!'s PNUTS [16] and per-user cache invalidation on Face-

book [21]. Nevertheless, we are unaware of any major use of *agreement* at the front-tier of customer-facing services. In this paper, we challenge the assumption that the tradeoff between strong consistency and cost in these services is fundamental.

This paper presents Prophecy, a system that lowers the performance overhead of fault-tolerant agreement for customer-facing Internet services, at the cost of slightly weakening its consistency guarantees. At Prophecy's core is a trusted *sketcher* component that mediates client access to a service replica group. The sketcher maintains a compact history table of observed request/response pairs; this history allows it to perform fast, load-balanced reads when state transitions do not occur (that is, when the current response is identical to that seen in the past) and slow, replicated reads otherwise (when agreement is required). The sketcher is a flexible abstraction that can *interface with any replica group*, provided it exposes a limited set of defined functionality. This paper, however, largely discusses Prophecy's use with BFT replica groups. Our contributions include the following:

- When used with BFT replica groups that guarantee linearizability [32], Prophecy significantly increases throughput through its use of fast, load-balanced reads. However, it relaxes the consistency properties to what we term *delay-once* semantics.
- We also derive a distributed variant of Prophecy, called D-Prophecy, that similarly improves the throughput of traditional fault-tolerant systems. D-Prophecy achieves the same delay-once consistency but *without any trusted components*.
- We introduce the notion of *delay-once consistency* and define it formally. Intuitively, it implies that faulty nodes can at worst return only stale (not arbitrary) data.
- We demonstrate how to scale-out Prophecy to support large replica groups or many replica groups.
- We implement Prophecy and apply it to BFT replica groups. We evaluate its performance on realistic workloads, not just null workloads as typically done in the literature. Prophecy adds negligible latency compared to standard load balancing, while it provides an almost linear-fold increase in throughput.
- Prophecy is most effective in read-mostly workloads where state transitions are rare. We conduct a measurement study of the Alexa top-25 websites and show that over 90% of requests are for mostly static data. We also characterize the dynamism in the data.

Table 1 summarizes the different properties of a traditional BFT system, D-Prophecy, and Prophecy. The remainder of this paper is organized as follows. In §2 we motivate the design of D-Prophecy and Prophecy, and we describe this design in §3. In §4 we define delay-once consistency and analyze Prophecy's implementation of this consistency model. In §5 we discuss extensions to the basic system model that consider scale and complex component topologies. We detail our prototype implementation in §6 and describe our system evaluation in §7. In §8 we present our measurement study. We review related work in §9 and conclude in §10.

Property	BFT	D-Prophecy	Prophecy
Trusted components	No	No	Yes
Modified clients	Yes	Yes	No
Session length	Long	Long	Short, long
Load-balanced reads	No	Yes	Yes
Consistency	Linearized	Delay-once	Delay-once

Table 1: **Comparison of a traditional BFT system, D-Prophecy, and Prophecy.**

2 Motivating Prophecy's Design

One might rightfully ask whether Prophecy makes unfair claims, given that it achieves performance and scalability gains at the cost of additional trust assumptions compared to traditional fault-tolerant systems. This section motivates our design through the lens of BFT systems, in two steps. First, we improve the performance of BFT systems on realistic workloads by introducing a cache at each replica server. By optimizing the use of this cache, we derive a distributed variant of Prophecy that does not rely on any trusted components. Then, we apply this design to customer-facing Internet services, and show that the constraints of these services are best met by a shared, trusted cache that proxies client access to the service replica group. The resulting system is Prophecy.

In our discussion, we differentiate between *write requests*, or those that modify service state, and *read requests*, or those that simply access state.

2.1 Traditional BFT Services and Real Workloads

A common pitfall of BFT systems is that they are evaluated on null workloads. Not only are these workloads unrealistic, but they also misrepresent the performance overheads of the system. Our evaluation in §7 shows that the cost of executing a non-null read request in the PBFT system [12] dominates the cost of agreeing on the ordering of the request, even when the request is served entirely from main memory. Thus the PBFT read optimization, which optimistically avoids agreement on read requests, offers little or no benefit for most realistic workloads. Improving the performance of read requests requires optimizing the *execution* of the reads themselves.

Unlike write requests, which modify service state and hence must be executed at each replica server, read requests can benefit from causality tracking. For example, if there are no causally-dependent writes between two identical reads, a replica server could simply cache the response of the first read and avoid the second read altogether.[1] However, this requires (1) knowledge of the causal dependencies of all write requests, and (2) a response cache of all prior reads at each replica server. The first requirement is unrealistic for many applications: a single write may modify the service state in complex ways. Even if we address this problem by invalidating the entire response cache upon receiving any write, the space needed by such a cache could be prohibitive: a cache of Facebook's 60+ billion images on April 30, 2009 [49], assuming a scant 1% working-set size, would occupy approximately 15TB of memory. Thus, the second requirement is also unrealistic.

Instead of caching each response r, the replica servers can store a compact, collision-resistant sketch $s(r)$ to enable *cache validation*. That is, when a client issues a read request for r, only one replica server executes the read and replies with r, while the remaining replica servers reply with $s(r)$ from their caches. The client accepts r only if the replica group agrees on $s(r)$ and if $s(r)$ validates r. Thus, even if the replica that returns r is faulty, it cannot make the client accept arbitrary data; in the worst case, it causes the client to accept a stale version of r. Therefore we only need to ask one replica to execute the read, effectively implementing what we call a *fast read*. Fast reads drastically improve the throughput of read requests and can be load-balanced across the replica group to avoid repeated stale results. The replica servers maintain a fresh cache by updating it during regular (replicated) reads, which are issued when fast reads fail. Using a compact cache reduces the memory footprint of the Facebook image working set to less than 27GB.

We call the resulting system Distributed Prophecy, or D-Prophecy, and call the consistency semantics it provides *delay-once consistency*.

2.2 BFT Internet Services

An oft-overlooked issue with BFT systems, including D-Prophecy, is that they are *implicitly* designed for services with long-running sessions between clients and replica servers (or at least always presented and evaluated as such). Clients establish symmetric session keys with each replica server, although the overhead of doing so is not typically included when calculating system performance. Figure 1 shows the throughput of the PBFT im-

[1] Other causality-based optimizations, such as client-side speculation [64] or server-side concurrent execution [37] are also possible, but are complementary to any cache-based optimizations.

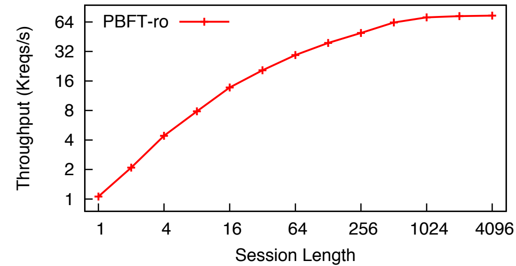


Figure 1: **PBFT's throughput in the thousands of requests per second for null requests in sessions of varying length. Note that both axes are log scale.**

plementation as a function of session length, with all relevant optimizations enabled including the read optimization (indicated by 'ro'). As sessions get shorter, throughput is drastically reduced because replicas need to decrypt and verify clients' new session keys. For PBFT sessions consisting of 128 read requests, throughput is half of its maximum, and for sessions consisting of 8 read requests, throughput is one-tenth of its maximum.

The assumption of long-lived sessions breaks down for Internet services, however, which are mostly characterized by *short-lived sessions* and *unmodified clients*. These properties make it impractical for clients to establish per-session keys with each replica. Moreover, depending on clients to perform protocol-specific tasks leads to poor backwards compatibility for legacy clients of Internet services (*e.g.*, web browsers), where cryptographic support is not easily available [2]. Instead, we might turn to using an entity knowledgeable of the BFT protocol to proxy client requests to a service replica group. And since Internet services already rely on the correct operation of local middleboxes (at least with respect to service availability), we extend this reliance by converting the middlebox into a trusted proxy. The trusted proxy interfaces multiple short-lived sessions between clients and itself with a single long-lived session between itself and the replica group, acting as a client in the traditional BFT sense.

When using proxied client access to a D-Prophecy group, there is no need to maintain redundant caches at each replica server: a shared cache at the trusted proxy suffices, and it preserves delay-once consistency. A fast read now mimics the performance of an unreplicated read, as the proxy only asks one replica server for r and validates the response with its (local) copy of $s(r)$. Since the cache is compact, the proxy remains a small and simple trusted component, amenable to verification. We call this system Prophecy, and present its design in §3.

2.3 Applications

The delay-once semantics of Prophecy imply that faulty nodes can at worst return stale (not arbitrary) data. This

semantics is sufficient for a variety of applications. For example, Prophecy would be able to protect against the Facebook and Flickr mishaps mentioned in the introduction, because it would not allow arbitrary data to reach the client. Applications that serve inherently static (write-once) data are also good candidates, because here a "stale" response is as fresh as the latest response. In §8 we demonstrate the propensity for static data in today's most popular websites.

Social networks and "Web 2.0" applications are good candidates for delay-once consistency because they typically do not require all writes to be immediately visible. Consider the following example from Yahoo!'s PNUTS system [16]. A user wants to upload spring-break photos to an online photo-sharing site, but does not want his mother to see them. So, he first removes her from the permitted access list of his database record and then adds the spring-break photos to this record. A consistency model that allows these updates to appear in different orders at different replicas, such as eventual consistency [22], is insufficient: it violates the user's intention of hiding the photos from his mother. Delay-once consistency only allows stale data to be returned, not data out-of-order: if the photos are visible, then the access control update must have already taken place. Further, once the user has "refreshed" his own page and sees the photos, he is guaranteed that his friends will also see them.

For applications where writes are critical, such as a bank account, delay-once consistency is appropriate because it ensures that writes follow the protocol of the replica group. Although reads may return stale results, they can only do so in a limited way, as we discuss in §4. Prophecy limits the duration of staleness in practice using load balancing. On the other hand, there are some applications for which delay-once consistency is not beneficial, such as those that critically depend on reading the latest data (*e.g.*, a rail signaling service), or those that return non-deterministic content (*e.g.*, a CAPTCHA generator).

3 System Design

We first define a sketcher abstraction that lies at the heart of Prophecy. For a more traditional setting, we use this sketcher to design a distributed variant of Prophecy, or D-Prophecy. We then present the design of Prophecy.

3.1 The Sketcher

Prophecy and D-Prophecy use a sketcher to improve the performance of read requests to an existing replica group. A *sketcher* maintains a history table of compact, collision-resistant sketches of requests and responses processed by a replica group. Each entry in the history table is of the form $(s(q), s(r))$, where q is a request, r is the response to q, and s is the sketching function used for compactness (s typically makes use of a secure hash function like SHA-1). The sketcher computes sketches and looks up or updates entries in the history table using a standard get/set interface, keyed by $s(q)$. In Prophecy, only read requests and responses are stored in the history table.

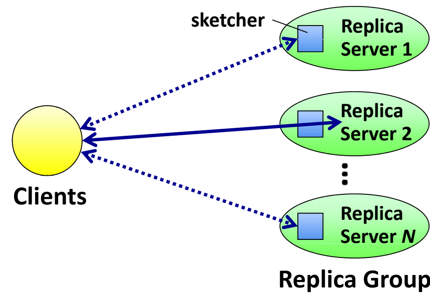


Figure 2: **Executing a fast read in D-Prophecy. Only one replica server executes the read (bold line); the others return the response sketch in the history table (dashed lines).**

The specific use of the sketcher and its interaction with the replica group differs between Prophecy and D-Prophecy. However, both systems require the replica group to support the following request interface:

- `RESP` $\leftarrow$ *fast*(`REQ` q)
- (`RESP` r, `SEQ_NO` σ) $\leftarrow$ *replicated*(`REQ` q)

We expect the *fast* interface to be new for most replica groups. The *replicated* interface should already exist, but may need to be extended to return sequence numbers. No modifications are made to the replica group beyond what is necessary to support the interfaces, in either system.

3.2 D-Prophecy

Figure 2 shows the system model of D-Prophecy. Except for the sketcher, all other entities are standard components of a replicated service: clients send requests to (and receive responses from) a service implemented by N replica servers, according to some replication protocol like PBFT. Each replica server is augmented with a sketcher that maintains a history table for read requests. The history table is read by the *fast* interface and updated by the *replicated* interface, as follows.

A client issues a fast read q by sending it to all replica servers and choosing one of them to execute q and return r. The policy for selecting a replica server is unspecified, but a uniformly random policy has especially useful properties (see §4.2). The other replicas use their sketcher to lookup the entry for $s(q)$ and return the corresponding response sketch $s(r)$, or null if the entry does

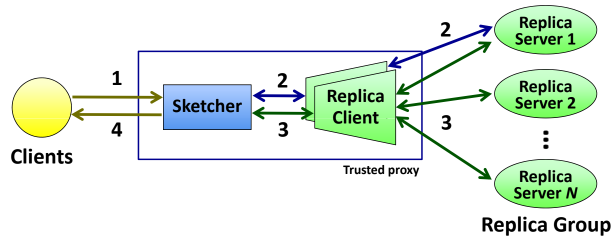


Figure 3: **Prophecy mediating access to a replica group.**

not exist. If the client receives a quorum of non-null response sketches that match the sketch of the actual response, it accepts the response. The quorum size depends on the replication protocol; we give an example below. Otherwise, we say a *transition* has occurred and the client reissues the request as a replicated read. A replicated read is executed according to the protocol of the replica group, with one additional step: all replica servers use their sketcher to update the entry for $s(q)$ with the new value of $s(r)$, before sending a response to the client.

Readers familiar with the PBFT protocol will notice that fast reads in D-Prophecy look very similar to PBFT optimized reads. However, there is a crucial difference: PBFT requires every replica server to execute the read, while D-Prophecy requires only one such execution, performing in-memory lookups of $s(r)$ at the rest. For non-null workloads, this represents a significant performance improvement, as shown in §7. On the flip side, each replica server requires additional memory to store its history table, though in practice this overhead is small. The quorum size required for fast reads is identical to the quorum size required for optimized reads: $(2N+1)/3$ responses suffices with some caveats (see §5.1.3 of [11]), and N always suffices.

The architecture of D-Prophecy resembles that of a traditional BFT system: clients establish session keys with the replica servers and participate fully in the replication protocol. As we observed in §2.2, this makes D-Prophecy unsuitable for Internet services, with their environment of short-lived sessions and unmodified clients. This motivates the design of Prophecy, discussed next.

3.3 Prophecy

Figure 3 shows the simplest realization of Prophecy's system model. (We consider extensions to the basic model in §5.) There are four types of entities: clients, sketchers, replica clients, and replica servers. Unmodified clients' requests to a service are handled by the sketcher; together with the replica clients, this serves as the trusted proxy described in §2.2. The replica clients interact with the service, implemented by a group of N replica servers, according to some replication protocol.

The sketcher issues each request through a replica client; the next subsection details the handling of requests. Functionally, the sketcher in Prophecy plays the same role as the per-replica-server sketchers in D-Prophecy. Architecturally, however, its role is quite different. In Prophecy, a fast read is sent only to the single replica server that executes it, and neither the *fast* nor *replicated* interface accesses the history table directly. Thus, the replica group is treated as a black box. Since the sketcher is external to the replica group, writes processed by the group may no longer be visible or discernible to the sketcher; *i.e.*, there may exist an *external write channel*. Since only replica clients interact directly with the replica servers, each replica client can maintain a single, long-lived session with each replica server. Wide-area clients are shielded from any churn in the replica group and are unaware of the replication protocol: the only responses they see are those that have already been accepted by the sketcher.

The type of session used between clients and the sketcher is left open by our design, as it may vary from service to service. For example, services that only allow read or simple write operations (*e.g.*, HTTP GETs and POSTs) may use unauthenticated sessions. A service like Facebook may use authentication only during user login, and use unauthenticated cookie-based sessions after that. Finally, services that store private or protected data, such as an online banking system, may secure sessions at the application level (*e.g.*, using HTTPS). Prophecy's architecture makes it easy to cope with the overhead of client-sketcher authentication, because one can simply add more sketchers if this overhead grows too high (see §5). To achieve the same scale-out effect, traditional BFT systems like PBFT and D-Prophecy would need to add entire replica groups.

3.3.1 Handling a Request

The sketcher stores two additional fields with each entry $(s(q), s(r))$ in the history table: the sequence number σ associated with r, and a 2-bit value b indicating whether $s(q)$ is *whitelisted* (always issued as a fast read), *blacklisted* (always issued as a replicated request), or neither (the default). The sketch $s(r)$ is empty for whitelisted or blacklisted requests. Algorithm 1 describes the processing of a request and is illustrated in Figure 3 (numbers on the right correspond to the numbered steps in the figure).

Prophecy requires a sequence number to be returned by *replicated*, as it seeks to issue concurrent requests to the replica group using multiple replica clients. Concurrency allows reads to execute in parallel to improve throughput. Unfortunately, a sketcher that issues requests concurrently has no way of discerning the correct order of replicated reads by itself, *i.e.*, the order they were processed by the replica group. Thus, it relies on

Algorithm 1 Processing a request at the sketcher.

```
Receive request q from client                                  (1)
if q is a read request then
    (s(q), s(r), σ, b) ← Lookup s(q) in history table
    if (s(r) ≠ null) and (b ≠ blacklisted) then
        r′ ← fast(q)                                           (2)
        if (s(r′) = s(r)) or (b = whitelisted) then
            return r′ to client                                (4)
        end if
    end if
    (r′, σ′) ← replicated(q)                                   (3)
    if (s(r) = null) or (σ′ > σ) then
        Update history table with (s(q), s(r′), σ′, b)
    end if
else
    (r′, σ′) ← replicated(q)                                   (3)
end if
return r′ to client                                            (4)
```

the sequence number returned by *replicated* to ensure that entries in the history table always reflect the latest system state.

The sketcher requires some application-specific knowledge of the format of q and r. This information is used to determine if q is a read or write request, and to discard extraneous or non-deterministic information from q or r while computing $s(q)$ or $s(r)$. For example, in our prototype implementation of Prophecy, an HTTP request is parsed by an HTTP protocol handler to extract the URL and HTTP method of the request; the same handler removes the date/time information from HTTP headers of the response. In practice, the required application-specific knowledge is minimal and limited to parsing protocol headers; the payload of the request or response (*e.g.*, the HTTP body) is treated opaquely by the sketcher.

Whitelisting and blacklisting add flexibility to the handling of requests, but may require additional application-specific knowledge. One use of blacklisting that does not require such knowledge is to dynamically blacklist requests that exhibit a high frequency of transitions (*e.g.*, dynamic content). This allows the sketcher to avoid issuing fast reads that are very likely to fail. (We do not currently implement this optimization.)

3.4 Performance

In our analysis and evaluation, the sketcher is able to accommodate all read requests in its history table without evicting any entries. If needed, a replacement policy such as LRU may be used, but this is unlikely: our current implementation can store up to 22 million unique entries using less than 1GB of memory.

The performance savings of a sketcher come from the ability to execute fast, load-balanced reads whose responses match the entries of the history table. Thus, Prophecy and D-Prophecy are most effective in read-mostly workloads. We can estimate the savings by looking at the cost, in terms of per-replica processing time, of executing a read in these systems. Let t be the probability that a state transition occurs in a given workload. Let C_R be the cost of a replicated read and C_r the cost of a fast read (excluding any sketcher processing in the case of D-Prophecy), and let C_{hist} be the cost of computing a sketch and performing a lookup/update in a history table. Below, we calculate the expected cost of a read in Prophecy and D-Prophecy when used with a BFT replica group that uses PBFT's read optimization. For comparison, we include the cost of the unmodified BFT group; here, t' is the probability that a PBFT optimized read fails.

$$\begin{aligned}
\text{Prophecy:} \quad & [C_r + 2C_{hist}] + [t(NC_R + C_{hist})] \\
\text{D-Prophecy:} \quad & [C_r + (N-1)C_{hist}] + [t(NC_R + NC_{hist})] \\
\text{BFT:} \quad & [NC_r] + [t'NC_R]
\end{aligned}$$

The addends on the left and right of each equation show the cost of a fast read and a replicated read, respectively. The equations do not include optimizations that benefit all systems equally, such as separating agreement from execution [67]. Prophecy performs two lookups in the history table during a fast read (one before and one after executing the read), and one update to the history table during a replicated read. D-Prophecy performs a history table lookup at all but one replica server during a fast read, and an update to the history table of each replica server during a replicated read. These equations show that Prophecy operates at maximum throughput when there are no transitions, because only one replica server processes each request, as compared to over $2/3$ of the replica servers in the BFT system (assuming, idealistically, that only a necessary quorum of replica servers execute the optimized read, and the remaining replicas ignore it). Since $C_{hist} \ll C_r$ for non-null workloads—the former involves an in-memory table lookup, the latter an actual read—this is a factor of over $(2/3)N$ improvement. D-Prophecy's savings are similar for the same reason. Although t' may be significantly less than t in practice—given that PBFT optimized reads may still succeed even when a state transition occurs—our evaluation in §7 reveals that the benefit of PBFT optimized reads over replicated reads is small for real workloads. Finally, while Prophecy's throughput advantage degrades as t increases, we demonstrate in §8 that t is indeed low for popular web services.

4 Consistency Properties

Despite their relatively simple designs, the consistency properties of Prophecy and D-Prophecy are only slightly weaker than those of the (unmodified) replica group. In this section, we formalize the notion of *delay-once consistency* introduced in §2. Delay-once consistency is a derived consistency model; here, we derive it from linearizability [32], the consistency model of most BFT protocols, and obtain *delay-once linearizability*. Then, we show how Prophecy implements delay-once linearizability.

4.1 Delay-once Linearizability

A history of requests and responses executed by a service is linearizable if it is equivalent to a sequential history [40] that respects the irreflexive partial order on requests imposed by their real-time execution [32]. Request X precedes request Y in this order, written $X \prec Y$, if the response of X is received before Y is sent. Suppose one client sends requests (R^a, W^b, R^c) to the service and another client sends requests (W^d, R^e, R^f, W^g), with partial order $\{R^a \prec R^e, W^g \prec R^c\}$. Then a valid linearized history could look like the following:

$$\langle R_0^a, W_1^d, W_2^b, R_2^e, R_2^f, W_3^g, R_3^c \rangle.$$

The R's and W's represent read and write requests, and subscripts represent the service state reflected in the response to each request (following [28]). In contrast to this history, the following is a valid delay-once linearizable history, though it is not linearizable:

$$\langle R_0^a, W_1^d, W_2^b, R_0^e, R_2^f, W_3^g, R_2^c \rangle.$$

Requests R^e and R^c have stale responses because they do not reflect the state update caused by sequentially precedent writes (note that the staleness of R^e's response is discernible to the issuing client, whereas the staleness of R^c's response is not). At a high level, a delay-once history looks like a linearized history with reads that reflect the state of prior reads, but not necessarily prior writes. The manner in which reads can be stale is not arbitrary, however. Specifically, a history H is *delay-once linearizable* if the subsequence of write requests in H, denoted by $H|_W$, satisfies linearizability, and if read requests satisfy the following property:

> **Delay-once property**. For each read request R_x in H, let R_y and W_z be the read and write request of maximal order in H such that $R_y \prec R_x$ and $W_z \prec R_x$. Then either $x = y$ or $x = z$.

Delay-once linearizability implies both monotonic read and monotonic write consistency, but not read-after-write consistency. If $\prec_H$ is the partial order of the history H, delay-once linearizability respects $\prec_{H|_W}$ but not $\prec_H$, due to the possible presence of stale reads.

The delay-once property ensures two things: first, reads never reflect state older than that of the latest read (they are only *delayed* to *one* stale state), and second, reads that are updated reflect the latest state immediately. Thus, a system that implements delay-once consistency is *responsive*. To verify if a read in a delay-once consistent history H is stale, one can check the following:

> **Staleness indicator**. Given a read request R_x in H, let W_y be the write request of maximal order in H such that $W_y \prec R_x$. R_x is stale if and only if $x < y$.

The staleness property explains why object-based systems like web services fare particularly well with delay-once consistency. In these systems, state updates to one object are isolated from other objects, so staleness can only occur between writes and reads that affect the same object.

The above derivation of delay-once consistency is based on linearizability, but derivations from other consistency models are possible. For example, a weaker condition called read-after-write consistency also yields meaningful delay-once semantics.

4.2 Prophecy's Consistency Semantics

We now show that Prophecy implements delay-once linearizability when used with a replica group that guarantees linearizability, such as a PBFT replica group. A similar (but simpler) argument shows that D-Prophecy achieves delay-once linearizability, omitted here due to space constraints.

Prophecy inherits the system and network model of the replica group. When used with a PBFT replica group, we assume an asynchronous network between the sketcher and the replica group that may fail to deliver messages, may delay them, duplicate them, or deliver them out-of-order. Replica clients issue requests to the replica group one at a time; requests are retransmitted until they are received. We do not make any assumptions about the organization of the service's state; for example, the service may be a monolithic replicated state machine [39, 58] or a collection of numerous, isolated objects [32]. The sketcher may process requests concurrently. We model this concurrency by allowing the sketcher to issue requests to multiple replica clients simultaneously; the order in which these requests return from replica clients is arbitrary. Updates to service state may not be discernible or visible to the sketcher—*i.e.*, there may exist an external write channel—as discussed in §3.3. We show that Prophecy achieves delay-once linearizability despite concurrent requests and external writers.

Our analysis of Prophecy's consistency requires a non-standard approach because it is the sketcher, not the

replica servers, that enforces this consistency, and because fast reads are executed by individual replicas. In particular, we introduce the notion of an *accepted history*. Let H_i for $1 \leq i \leq N$ be the history of all write requests executed by replica server i and all fast read requests executed by i that were accepted by the sketcher. Let R_s be the history of all replicated read requests accepted by the sketcher. An accepted history A_i is the union of H_i and R_s, for each replica server i. The position in A_i of each replicated read in R_s is well defined because all reads are accepted at a single location (the sketcher) and all replicated requests are totally ordered by linearizability. We claim that the accepted history A_i is delay-once linearizable.

To see this, observe that replicated requests satisfy linearizability because they follow the protocol of the replica group. The sketcher ensures that replicated reads update the history table according to this order by using the sequence numbers returned by the *replicated* interface. Further, the sketcher only accepts a fast read if it reflects the state of the latest replicated read. Since A_i contains all replicated reads accepted by the sketcher (not just those accepted by i), and since accepted fast reads never reflect new state, it follows that all fast reads in A_i must satisfy the delay-once property. While A_i may not contain all write requests accepted by the replica group (*e.g.*, if i is missing an update), this only affects i's ability to participate in replicated reads, and does not violate delay-once linearizability. Thus, we conclude that A_i is delay-once linearizable.

Limiting staleness via load balancing. Stale responses are returned by faulty replica servers or correct replica servers that are out-of-date. We can easily verify if an accepted history contains stale responses by checking the staleness indicator defined in §4.

To limit the number of stale responses, the *fast* interface dispatches fast reads from all clients uniformly at random over the replica servers.[2] Let g be the fraction of faulty or out-of-date replica servers currently in the replica group. If g is a constant, then g^k, the probability that k consecutive fast reads are sent to these servers, is exponentially decreasing. For BFT protocols, $g < 2/3$ assuming a worst-case scenario where the maximum number of correct nodes are out-of-date. For a replica group of size 4, the probability that $k > 6$ is less than 1.6%.

[2]We assume for simplicity that the random selection is secure, though in practice faulty replica servers may hamper this process. The latter is an interesting problem, but outside the scope of this paper.

5 Scale and Complex Architectures

This section describes extensions to the basic Prophecy model in order to integrate fault tolerance into larger-scale and more complex environments.

Scaling through multiple sketchers. In the basic system model of Prophecy (Figure 3), the sketcher is a single bottleneck and point-of-failure. We address this limitation by using multiple sketchers to build a sketching core, as follows. First, we horizontally partition the global history table, based on $s(q)$'s, into non-overlapping regions, *e.g.*, using consistent hashing [33]. We assign each region to a distinct sketcher, which we refer to as *response sketchers*. The partitioning preserves delay-once semantics because only a single sketcher stores the entry for each $s(q)$. Second, we build a two-level sketching system as shown in Figure 4, where the first tier of *request sketchers* demultiplex client requests. That is, given a request q, any of a small number of request sketchers computes $s(q)$ and forwards q to the appropriate response sketcher. Using a one-hop distributed hash table (DHT) [27, 33] to manage the partitioning works well, given the network's small, highly-connected nature. The response sketchers (the members of this DHT) issue requests to the replica group(s) and sketch the responses, ultimately returning them to the clients. (Importantly, the replica servers in Figure 4 need not be part of a single replica group, but may instead be organized into multiple, smaller groups.) The larger number of response sketchers reflects the asymmetric bandwidth requirements of network protocols like HTTP. We evaluate the scaling benefits of multiple response sketchers in §7.7.

Handling sketcher failures. The sketching core handles failure and recovery of sketchers seamlessly, because it can rely on the join and leave protocol of the underlying DHT. Since request sketchers direct client requests, they maintain the partitioning of the DHT. To preserve delay-once semantics, this partitioning must be kept consistent [10, 66] to avoid sending requests from the same region of the history table to multiple response sketchers. Prophecy's support for blacklisting simplifies this task, however. In particular, whenever a region of the history table is being relinquished or acquired between response sketchers, we can allow more than one response sketcher to serve requests from the same region provided the entire region is blacklisted (forcing all requests to be replicated). Once the partitioning has stabilized, the new owner of the region can unset the blacklist bit. As a result, membership dynamics can be handled smoothly and simply, at the cost of transient inefficiency but not inconsistency.

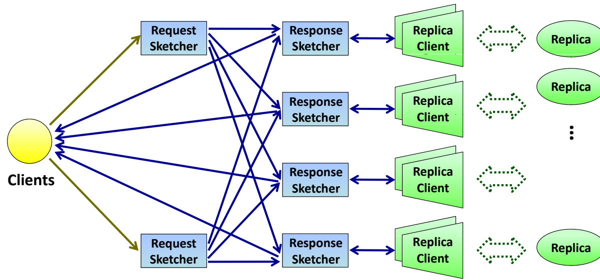


Figure 4: **Scaling out Prophecy using multiple sketchers.**

Mediating loosely-coupled groups. A sketching core can be shared by the multiple, loosely-coupled components that typically comprise a real service. Alternatively, components that operate in parallel can use Prophecy via dedicated sketchers. Components that operate in series, such as multi-tier web services, can use Prophecy prior to each tier. However, applying agreement protocols in series introduces nontrivial consistency issues. We leave treatment of this problem to future work.

6 Implementation

Our implementation of Prophecy and D-Prophecy is based on PBFT [12]. We used the PBFT codebase given its stable and complete implementation, as well as newer results [6] showing its competitiveness with Zyzzyva and other recent protocols (much more so than was originally indicated [38]). We implemented and compared three proxied systems (Prophecy, proxied PBFT without optimized reads, and proxied PBFT with optimized reads), as well as three non-proxied ("direct") systems (D-Prophecy, PBFT without optimized reads, and PBFT with optimized reads). In our evaluation, we will compare proxied systems only with other proxied systems, and similarly for direct systems, as the architectures and assumptions of the two models are fundamentally different. The proxied systems do not authenticate communication between clients and the sketcher, though they easily can be modified to do so with equivalent overheads.

We implemented a user-space Prophecy sketcher in about 2,000 lines of C++ code using the Tamer asynchronous I/O library [36]. The sketcher forks a process for each core in the machine (8 in our test cluster), and the processes share a single history table via shared memory. The sketcher interacts with PBFT replica clients through the PBFT library. The pool of replica clients available to handle requests is managed as a queue. The sketching function uses a SHA-1 hash [48] over parts of the HTTP header (for requests) and the entire response body (for responses). The proxied PBFT variants share the same code base as the sketcher, but do not perform sketching, issue fast reads, or create or use the history table.

We modified the PBFT library in three ways: to add support for fast reads (about 20 lines of code), to return the sequence numbers (about 20 LOC), and to add support for D-Prophecy (about 100 LOC). Additional modifications enabled the same process to use multiple PBFT clients concurrently (500 LOC), and modified the simple server distributed with PBFT to simulate a webserver and allow "null" writes (500 LOC), as null operations actually have 8-byte payloads in PBFT. We also wrote a PBFT client in about 1000 lines of C++/Tamer that can be used as a client in direct systems and as a replica client in proxied systems.

7 Evaluation

This section quantifies the performance benefits and costs of Prophecy and D-Prophecy, by characterizing their latency and throughput relative to PBFT under various workloads. We explore how the system's throughput characteristics change when we modify a few key variables: the processing time of the request, the size of the response, and the client's session length. Finally, we examine how Prophecy scales out in terms of replica group size.

7.1 Experimental Setup

All of our experiments were run in a 25-machine cluster. Each machine has eight 2.3GHz cores and 8GB of memory, and all are connected to a 1Gbps switch.

The proxied systems are labeled Prophecy, pr-PBFT (proxied PBFT), and pr-PBFT-ro (proxied PBFT with the read optimization). The direct systems are labeled D-Prophecy, PBFT, and PBFT-ro (PBFT with the read optimization). Multicast and batching are not used in our experiments, as they do not impact performance when using read optimizations; all other PBFT optimizations are employed. Unless otherwise specified, all experiments used four replica servers, a single sketcher/proxy machine for the proxied systems, and a single client machine. The proxied experiments used 40 replica clients across eight processes at the sketcher/proxy, and had 100 clients establish persistent HTTP connections with the sketcher/proxy. The direct experiments used 40 clients across eight processes. These numbers were sufficient to fully saturate each system without degrading performance. All experiments use infinite-length sessions between communicating entities (except for the one evaluating the effect of session length). Throughput experiments were run for 30-second intervals and throughput was averaged over each second.

System	median	1st	99th
pr-PBFT	433	379	706
pr-PBFT-ro	296	255	544
Prophecy	256	216	286
Prophecy-100	617	553	768
PBFT	286	272	309
PBFT-ro	144	135	168
D-Prophecy	144	129	197
D-Prophecy-100	429	412	574

Table 2: **Latency in μs for serial null reads.**

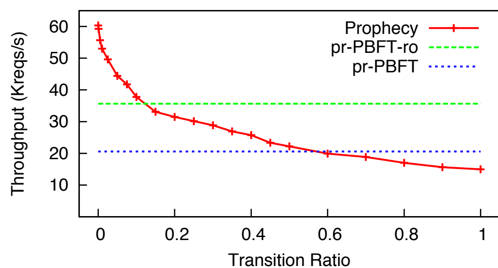


Figure 5: **Throughput of null reads for proxied systems (Prophecy, pr-PBFT, and pr-PBFT-ro).**

In some experiments, we report numbers for Prophecy-X or D-Prophecy-X, which signifies that the systems experienced state transitions $X\%$ of the time.

7.2 Null Workload

Latency. Table 2 shows the median and 99th percentile latencies for 100,000 serial null requests sent by a single client. All systems displayed low latencies under $1ms$, although the proxied systems have higher latencies as each request must traverse an extra hop. Prophecy, pr-PBFT-ro, D-Prophecy, and PBFT-ro all avoid the agreement phase during request processing and thus have notably lower latency than their counterparts. Prophecy-100 and D-Prophecy-100 represent a worst-case scenario where every fast read fails and is reissued as a replicated read.

Throughput. Figure 5 shows the aggregate throughput of the proxied systems for executing null requests. We achieve the desired transition ratio by failing that fraction of fast reads at the sketcher.

Since replica servers can execute null requests cheaply, the sketcher/proxy becomes the system bottleneck in these experiments. Nevertheless, Prophecy achieves 69% higher throughput than pr-PBFT-ro due to its load-balanced fast reads, which require fewer packets to be processed by replica servers. As the transition ratio increases, however, Prophecy's advantage decreases because fewer fast reads match the history table. For example, when transitions occur 15% of the time—a representative ratio from our measurement study in §8—

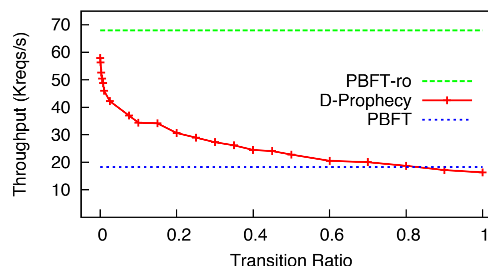


Figure 6: **Throughput of null reads for direct systems (D-Prophecy, PBFT, and PBFT-ro).**

Prophecy's throughput is 7% lower than that of pr-PBFT-ro.

Figure 6 depicts the aggregate throughput of the direct systems. In this experiment, 40 clients across two machines concurrently execute null requests. D-Prophecy's throughput is 15% lower than PBFT-ro's when there are no transitions, and 50% lower when there are 15% transitions. D-Prophecy derives no performance advantage from its fast reads because the optimized reads of PBFT take no processing time, while D-Prophecy incurs additional overhead for sketching and history table operations.

7.3 Server Processing Time

The previous subsection shows that when requests take almost no time to process, Prophecy improves throughput only by decreasing the number of packets at each replica server, while D-Prophecy fails to achieve better throughput. However, when the replicas perform real work, such as the computation or disk I/O associated with serving a webpage, Prophecy's improvement is more dramatic.

Figures 7 and 8 demonstrate how varying processing time affects the throughput of proxied systems (normalized against pr-PBFT-ro) and direct systems (normalized against PBFT-ro), respectively. As the processing time increases—implemented using a busy-wait loop—the cost of executing requests begins to dominate the cost of agreeing on their order. This decreases the effectiveness of PBFT's read optimization, as evidenced by the increase in pr-PBFT's throughput relative to pr-PBFT-ro, and similarly between PBFT and PBFT-ro. At the same time, the higher execution costs dramatically increase the effectiveness of load balancing in Prophecy and D-Prophecy. Their throughput approaches 3.9 times the baseline, which is only 2.5% less than the theoretical maximum.

The effectiveness of load-balancing is more pronounced in Prophecy than in D-Prophecy for two main reasons. First, Prophecy's fast reads involve only one replica server, while D-Prophecy's fast reads involve all replicas, even though only a single replica actually exe-

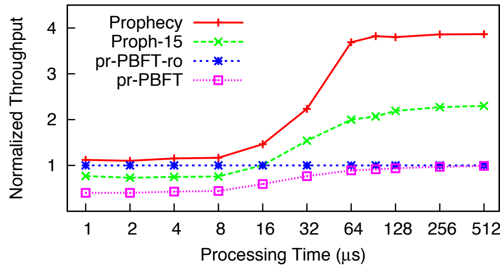


Figure 7: **Throughput of proxied systems as processing time increases, normalized against pr-PBFT-ro.**

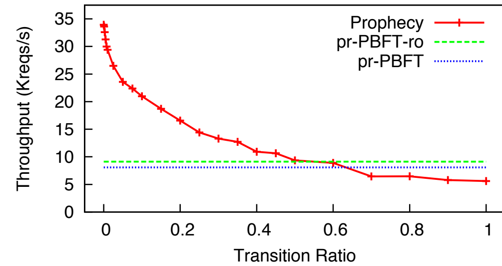


Figure 9: **Throughput of reads of a 1-byte webpage to Apache webservers for proxied systems.**

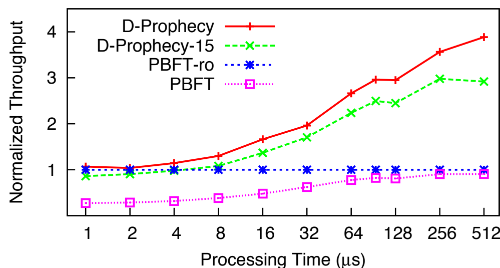


Figure 8: **Throughput of direct systems as processing time increases, normalized against PBFT-ro.**

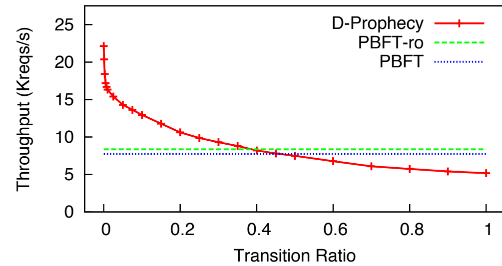


Figure 10: **Throughput of concurrent reads of a 1-byte webpage to Apache webservers for direct systems.**

cutes the request. Second, Prophecy performs sketching and history table operations at the sketcher, whereas D-Prophecy implements such functionality on the replica servers, stealing cycles from normal processing.

7.4 Integration with Apache Webserver

We applied Prophecy to a replica group in which each server runs the Apache webserver [7], appropriately modified to return deterministic results. Upon receiving a request, a PBFT server dispatches the request body to Apache via a persistent TCP connection over localhost.

Figure 9 shows the aggregate throughput of the proxied systems for serving a 1-byte webpage. When there are no transitions, Prophecy's throughput is 372% that of pr-PBFT-ro. At the representative ratio of 15%, Prophecy's throughput is 205% that of pr-PBFT-ro. The processing time of Apache is enough to dominate all other factors, causing Prophecy's use of fast reads to significantly boost its throughput.

Figure 10 shows the throughput of direct systems. With no transitions, D-Prophecy's throughput is 265% that of PBFT-ro, and 141% when there are 15% transitions.

In these experiments, the local HTTP requests to Apache took an average of $94\mu s$. For the remainder of this section, we use a simulated processing time of $94\mu s$ within replica servers when answering requests.

7.5 Response Size

Next, we evaluate the proxied systems' performance when serving webpages of increasing size, as shown by Figure 11. As the response size increases, fewer replica clients were needed to maximize throughput. At the same time, Prophecy's throughput advantage decreases as the response size increases, as the sketcher/proxy becomes the bottleneck in each scenario. Increasing the replica servers' processing time shifts this drop in Prophecy's throughput to the right, as it increases the range of response sizes for which processing time is the dominating cost. Note that we only evaluate the systems up to 64KB responses, because PBFT communicates via UDP, which has a maximum packet size of 64KB.

7.6 Session Length

Our experiments with direct systems so far did not account for the cost of establishing authenticated sessions between clients and replica servers. To establish a new session, the client must generate a symmetric key that it encrypts with each replica server's public key, and each replica server must perform a public-key decryption. Given the cost of such operations, the performance of short-lived sessions can be dominated by the overhead of session establishment, as we discussed in §2.2.

Figure 12 demonstrates the effect of varying session length on the direct systems, in which each request per session returns a 1-byte webpage. We find that the throughput of PBFT and PBFT-ro are indistinguishable

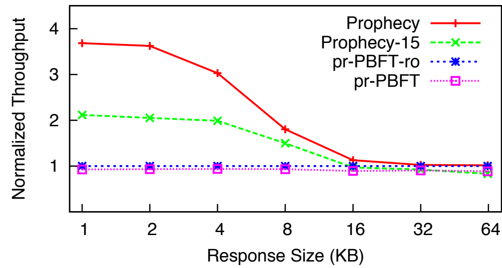


Figure 11: **Throughput of proxied systems as response size increases, normalized against pr-PBFT-ro.**

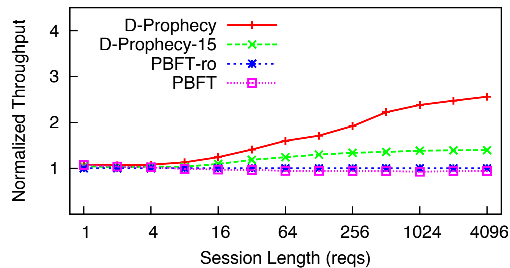


Figure 12: **Throughput of direct systems as session length increases, normalized against PBFT-ro.**

for short sessions, but as session length increases, the cost of session establishment is amortized over a larger number of requests, and PBFT-ro gains a slight throughput advantage. Similarly, D-Prophecy achieves its full throughput advantage only when sessions are very long.

We do not evaluate the effect of session lengths in the proxied systems, because they currently do not authenticate communication with the clients. Authentication can easily be incorporated into these systems, however, at a similar cost to Prophecy and pr-PBFT. That said, proxied systems can better scale up the maximum rate of session establishment than direct systems, as we observed in §3.3: each additional proxy provides a linear rate increase, while direct systems require an entire new replica group for a similar linear increase.

7.7 Scaling Out

Finally, we characterize the scaling behavior of Prophecy and proxied PBFT systems. By increasing the size of their replica groups, PBFT systems gain resilience to a greater number of Byzantine faults (*e.g.*, from one fault per 4 replicas, to four faults per 13 replicas). However, their throughput does not increase, as each replica server must still execute every request. On the other hand, Prophecy's throughput can benefit from larger groups, as it can load balance fast reads over more replica servers. As the sketcher can become a bottleneck in the system at higher read rates, we used two sketchers for a 7-replica group and three sketchers for a 10- and 13-replica group.

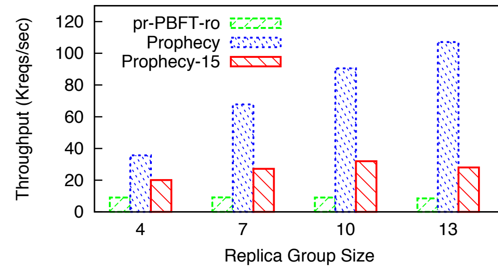


Figure 13: **Throughput of Prophecy and pr-PBFT-ro with varying replica group sizes.**

Figure 13 shows the throughput of proxied systems for increasing group sizes. Prophecy's throughput is 395%, 739%, 1000%, and 1264% that of pr-PBFT-ro, for group sizes of 4, 7, 10, and 13 replicas, respectively. Prophecy does not achieve such a significant throughput improvement when experiencing transitions, however. We see that a 15% transition ratio prevents Prophecy from handling more than 32,000 req/s, which it achieves with a replica group of size 10. Thus, under moderate transition rates, further increasing the replica group size will only increase fault tolerance, not throughput.

8 Measurement Study of Alexa Sites

The performance savings of Prophecy are most pronounced in read-mostly workloads, such as those involving DNS: of the 40K names queried by the ConfiDNS system [52], 95.6% of them returned the same set of IP addresses every time over the course of one day. In web services, it is less clear that transitions are rare, given the pervasiveness of so-called "dynamic content".

To investigate this dynamism, we collected data from the Alexa top 25 websites by scripting a Firefox browser to reload the main page of each site every 20 seconds for 24 hours on Dec. 29, 2008. Among the top sites were `www.youtube.com`, `www.facebook.com`, `www.skyrock.com`, `www.yahoo.co.jp`, and `www.ebay.com`.[3] The browser loads and executes all embedded objects and scripts, including embedded links, JavaScript, and Flash, with caching disabled. We captured all network traffic using the tcpflow utility [19], and then ran our HTTP parser and SHA-1-based sketching algorithm to build a compact history of requests and responses, similar to the real sketcher.

Our measurement results show that transitions are rare in most of the downloaded data. We demonstrate a clear

[3] While one might argue that BFT agreement is overkill for many of the sites in our study, our examples in the introduction show that Heisenbugs and one-off misconfigurations can lead to embarrassing, high-profile events. Prophecy protects against these mishaps without the performance penalty normally associated with BFT agreement.

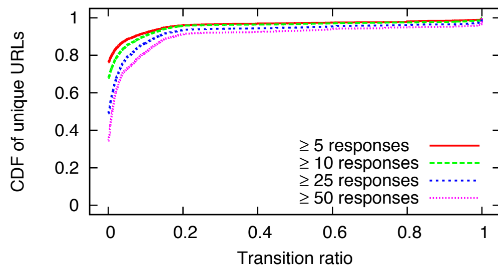


Figure 14: **A CDF of requests over transition ratios.**

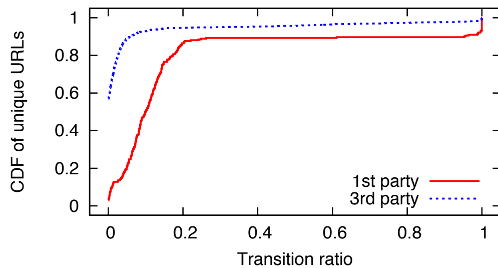


Figure 15: **A CDF over transition ratios of first-party vs. third-party URLs.**

divide between very static and very dynamic data, and use Rabin fingerprinting [55] to characterize the dynamic data. Finally, we isolate the results of individual geographic "sites" using a CIDR prefix database.

8.1 Frequency of Transitions

For each unique URL requested during the experiment, we measured the ratio of state transitions over repeated requests. Figure 14 shows a CDF of unique URLs at different transition ratios. We separately plotted those URLs based on the number of requests sent to each one, given that embedded links generate a variable number of requests to some sites. (Where not specified, the minimum number of requests used is 25.) We see that roughly 50% of all data accessed is purely static, and about 90% of all requests have fewer than 15% state transitions. These numbers confirmed our belief that most dynamic websites are actually dynamic compositions of very static content. The same graph scaled by the average response size of each request yields very similar curves (omitted), suggesting that Figure 14 also reflects the total response throughput at each transition ratio.

Figure 15 is the same plot as Figure 14 but divided into first-party URLs, or those targeted at an Alexa top website, and third-party URLs, or those targeted at other sites (given that first-party sites can embed links to other domains for image hosting, analytics, advertising, etc.). The graph shows that third-party content is much more static than first-party content, and thus third-party content providers like CDNs and advertisers could benefit substantially from Prophecy.

The results in this section are conservative for two reasons. First, they reflect a workload of only three requests per minute per site, when in reality there may be tens or hundreds of thousands of requests per minute. Second, many URLs—though not enough to cause space problems in a real history table—saw only a few requests, but returned identical responses, suggesting that our HTTP parser was conservative in parsing them as unique URLs. An important characteristic of all of the graphs in this section is the relatively flat line across the middle: this suggests that most data is either very static or very dynamic. The next section discusses dynamic data in more detail.

8.2 Characterizing Dynamic Data

Dynamic data degrades the performance of Prophecy because it causes failed fast reads to be resent as replicated reads. Often, however, the amount of dynamism is small and may even be avoidable. To investigate this, we characterized the dynamism in our data by using Rabin fingerprinting to efficiently compare responses on either side of a transition. We divided each response into chunks of size 1K in expectation [47], or a minimum of 20 chunks for small requests.

Our measurements indicate that 50% of all transitions differ in at least 30% of their chunks, and about 13% differ in all of their chunks. Interestingly, the *edit distance* of these transitions was much smaller: we determined that 43% of all transitions differ by a single contiguous insertion, deletion, or replacement of chunks, while preserving at least half or no more than doubling the number of original chunks. By studying transitions with low edit distance, we can identify sources of dynamism that may be refactorable. For example, a preliminary analysis of around 4,000 of these transitions (selected randomly) revealed that over half of them were caused by load-balancing directives (*e.g.*, a number appended to an image server name) and random identifiers (*e.g.*, client IDs) placed in embedded links or parameters to JavaScript functions. In fact, most of the top-level pages we downloaded, including seemingly static pages like `www.google.com`, were highly dynamic for this exact reason. A more in-depth analysis is slated for future work.

8.3 Site-Based Analysis

A "site" represents a physical datacenter or cluster of machines in the same geographic location. A single site may host large services or multiple services. Having demonstrated Prophecy's ability to scale out in such environments, we now study the potential benefit of deploying Prophecy at the sites in our collected data. To organize our data into geographic sites, we used forward

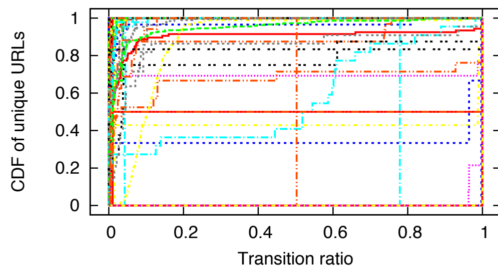


Figure 16: **A CDF of URLs over transition ratios for all sites for which CIDR data was available.**

and reverse DNS lookups on each requested URL and matched the resulting IP addresses against a CIDR prefix database. (This database, derived from data supplied by Quova [54], included over 2 million distinct prefixes, and is thus significantly finer-grained than those provided by RouteViews [57].) Requests that mapped to the same CIDR prefix were considered to be part of the same site. Figure 16 shows an overlay of the transition plots of each site. From the figure, a few sites serve very static data or very dynamic data only, but most sites serve a mix of very static and very dynamic data. All but one site (`view.atdmt.com`) show a clear divide between very static and very dynamic data.

9 Related Work

A large body of work has focused on providing strong consistency and availability in distributed systems. In the fail-stop model, state machine replication typically used primary copies and view change algorithms to improve performance and recover from failures [41, 50]. Quorum systems focused on tradeoffs between overlapping read and write sets [26, 31]. These protocols have been extended to malicious settings, both for Byzantine fault-tolerant replicated state machines [12, 42, 56], Byzantine quorum systems [1, 46], or some hybrid of both [17]. Modern approaches have optimized performance via various techniques, including by separating agreement from execution [67], using optimistic server-side speculation on correct operation [38], reducing replication costs by optimizing failure-free operation [65], and allowing concurrent execution of independent operations [37]. Prophecy's history table is motivated by the same assumption as this last approach—namely, that many operations/objects are independent and hence often remain static over time.

Given the perceived cost of achieving strong consistency and a particular desire to provide "always-on" write availability, even in the face of partitions, a number of systems opted for cheaper techniques. Several BFT replicated state machine protocols were designed with weaker consistency semantics, such as BFT2F [44], which weakens linearizability to fork* consistency, and Zeno [59], which weakens linearizability to eventual consistency. Several filesystems were designed in a similar vein, such as SUNDR [45] and systems designed for disconnected [29, 35] or partially-connected operation [51]. BASE [53] explored eventual consistency with high scalability and partition tolerance; the foil to database ACID properties. More recently, highly-scalable storage systems being built out within datacenters have also opted for cheaper consistency techniques, including the Google File System [25], Yahoo!'s PNUTS [16], Amazon's Dynamo [18], Facebook's Cassandra [20], eBay's storage techniques [61], or the popular approach of using Memcached [23] with a backend relational database. These systems take this approach partly because they view stronger consistency properties as infeasible given their performance (throughput) costs; Prophecy argues that this tradeoff is not necessary for read-mostly workloads.

Recently, several works have explored the use of trusted primitives to cope with Byzantine behavior. A2M [13] prevents faulty nodes from lying inconsistently by using a trusted append-only memory primitive, and TrInc [43] uses a trusted hardware primitive to achieve the same goal. Chun *et al.* [14] introduced a lightweight BFT protocol for multi-core single-machine environments that runs a trusted coordinator on one core, similar in philosophy to Prophecy's approach of extending the trusted computing base to include the sketcher.

Prophecy is unique in its application to customer-facing Internet services and its ability to load-balance read requests across a replica group while retaining good consistency semantics. Perhaps closest to Prophecy's semantics is the PNUTS system [16], which supports a load-balanced read primitive that satisfies timeline consistency (all copies of a record share a common timeline and only move forward on that timeline). Delay-once linearizability is strictly stronger than timeline consistency, however, because it does not allow a client to see a copy of a record that is more stale than a copy the client has already seen (whereas timeline consistency does).

There has been some work on using history as a consistency or security metric for particular applications. Aiyer et al. [4, 5] develop k-quorum systems that bound the staleness of a read request to one of the last k written values. Using Prophecy with a k-quorum system may be synergistic: Prophecy's load-balanced reads are less costly than quorum reads, and k-quorum systems can protect against an adversarial scheduler that attempts to hamper Prophecy's load balancing. The Farsite file system [3] uses historical sketches to validate read requests, but requires a lease-based invalidation protocol to keep sketches strongly consistent. The system modifies clients extensively and requires knowledge of causal

dependencies, violating Prophecy's design constraints (if these constraints are ignored, then D-Prophecy can easily be modified to achieve the same consistency as Farsite). Pretty Good BGP [34] whitelists BGP advertisements whose new route to a prefix includes its previous originating AS, while other routes require manual inspection. ConfiDNS [52] uses both agreement and history to make DNS resolution more robust. It requires results to be static for a number of days and agreed upon by some number of recursive DNS resolvers. Perspectives [63] combines history and agreement in a similar way to verify the self-signed certificates of SSH or SSL hosts on first contact. Prophecy can be viewed as a framework that leverages history and agreement in a general manner.

10 Conclusions

Prophecy leverages history to improve the throughput of Internet services by expanding the trusted middlebox between clients and a service replica group, while providing a consistency model that is very promising for many applications. D-Prophecy achieves the same benefits for more traditional fault-tolerant services. Our prototype implementations of Prophecy and D-Prophecy easily integrate with PBFT replica groups and are demonstrably useful in scale-out topologies. Performance results show that Prophecy achieves 372% of the throughput of even the read optimized PBFT system, and scales linearly as the number of sketchers increases. Our evaluation demonstrates the need to consider a variety of workloads, not just null workloads as typically done in the literature. Finally, our measurement study of the Internet's most popular websites demonstrates that a read-mostly workload is applicable to web service scenarios.

Acknowledgments

We thank our shepherd Petros Maniatis for helpful comments on earlier versions of this paper. Siddhartha Sen was supported through a Google Fellowship in Fault Tolerant Computing. Equipment and other funding was provided through the Office of Naval Research's Young Investigator program. None of this work reflects the opinions or positions of these organizations.

References

[1] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable byzantine fault-tolerant services. In *SOSP*, Oct. 2005.

[2] B. Adida. Helios: Web-based open-audit voting. In *USENIX Security*, July 2008.

[3] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *OSDI*, Dec 2002.

[4] A. S. Aiyer, L. Alvisi, and R. A. Bazzi. On the availability of non-strict quorum systems. In *DISC*, Sept. 2005.

[5] A. S. Aiyer, L. Alvisi, and R. A. Bazzi. Byzantine and multi-writer K-quorums. In *DISC*, Sept. 2006.

[6] L. Alvisi, A. Clement, M. Dahlin, M. Marchetti, and E. Wong. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *NSDI*, Apr. 2009.

[7] Apache HTTP Server. `http://httpd.apache.org/`, 2009.

[8] A. Avizienis and L. Chen. On the implementation of n-version programming for software fault tolerance during execution. In *IEEE COMPSAC*, Nov. 1977.

[9] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4), 1985.

[10] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI*, Nov. 2006.

[11] M. Castro. *Practical Byzantine Fault-Tolerance*. PhD thesis, Mass. Inst. of Tech., 2000.

[12] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *OSDI*, Feb. 1999.

[13] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: making adversaries stick to their word. In *SOSP*, Oct. 2007.

[14] B.-G. Chun, P. Maniatis, and S. Shenker. Diverse replication for single-machine Byzantine-Fault Tolerance. In *USENIX Annual*, June 2008.

[15] A. Clement, M. Marchetti, E. Wong, L. Alvisi, and M. Dahlin. BFT: The time is now. In *LADIS*, Sept. 2008.

[16] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. In *VLDB*, Aug. 2008.

[17] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *OSDI*, Nov. 2006.

[18] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lak-shman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[19] J. Elson. tcpflow—A TCP Flow Recorder. `http://www.circlemud.org/~jelson/software/tcpflow/`, 2009.

[20] Facebook. Facebook release cassandra: A structured storage system on a p2p network. `http://code.google.com/p/the-cassandra-project/`, 2008.

[21] Facebook. Scaling out. `http://www.facebook.com/note.php?note_id=23844338919`, Aug. 2008.

[22] A. Fekete, D. Gupta, V. Luchangco, N. Lynch, and A. Shvartsman. Eventually-serializable data services. *Theoretical Computer Science*, 220(1), 1999.

[23] B. Fitzpatrick. Memcached: a distributed memory object caching system. `http://www.danga.com/memcached/`, 2009.

[24] Flickr. Flickr phantom photos. http://flickr.com/help/forum/33657/, Feb. 2007.

[25] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, Oct. 2003.

[26] D. K. Gifford. Weighted voting for replicated data. In *SOSP*, Dec. 1979.

[27] A. Gupta, B. Liskov, and R. Rodrigues. Efficient routing for peer-to-peer overlays. In *NSDI*, Mar. 2004.

[28] J. V. Guttag, J. J. Horning, and J. M. Wing. Larch in five easy pieces. Technical Report 5, DEC Systems Research Centre, 1985.

[29] J. Heidemann and G. Popek. File system development with stackable layers. *ACM Trans. Comp. Sys.*, 12(1), Feb. 1994.

[30] J. Hendricks, G. Ganger, and M. Reiter. Low-overhead Byzantine fault-tolerant storage. In *SOSP*, Oct. 2007.

[31] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Trans. Comp. Sys.*, 4(1), Feb. 1986.

[32] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Sys.*, 12(3), 1990.

[33] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *STOC*, May 1997.

[34] J. Karlin, S. Forrest, and J. Rexford. Pretty Good BGP: Improving BGP by cautiously adopting routes. In *ICNP*, Nov. 2006.

[35] J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Trans. Comp. Sys.*, 10 (3), Feb. 1992.

[36] E. Kohler. Tamer. http://read.cs.ucla.edu/tamer/, 2009.

[37] R. Kotla and M. Dahlin. High-throughput Byzantine fault tolerance. In *DSN*, June 2004.

[38] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *SOSP*, Oct. 2007.

[39] L. Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. Program. Lang. Sys.*, 6(2), 1984.

[40] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9), Sept. 1979.

[41] L. Lamport. The part-time parliament. *ACM Trans. Comp. Sys.*, 16(2), 1998.

[42] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Sys.*, 4 (3), 1982.

[43] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. Trinc: small trusted hardware for large distributed systems. In *NSDI*, Apr. 2009.

[44] J. Li and D. Mazières. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *NSDI*, Apr. 2007.

[45] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, Dec. 2004.

[46] D. Malkhi and M. Reiter. Byzantine quorum systems. In *STOC*, May 1997.

[47] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *SOSP*, Oct. 2001.

[48] NIS95. *FIPS Publication 180-1: Secure Hash Standard*. Natl. Institute of Standards and Technology, Apr. 1995.

[49] F. E. Notes. Needle in a haystack: efficient storage of billions of photos. http://www.facebook.com/note.php?note_id=76191543919.

[50] B. M. Oki and B. H. Liskov. Viewstamped replication: a general primary copy. In *PODC*, 1988.

[51] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible update propagation for weakly consistent replication. In *SOSP*, Oct. 1997.

[52] L. Poole and V. S. Pai. ConfiDNS: Leveraging scale and history to improve DNS security. In *WORLDS*, Nov. 2005.

[53] D. Pritchett. BASE: An ACID alternative. *ACM Queue*, 6(3), 2008.

[54] Quova. http://www.quova.com/, 2006.

[55] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Harvard Aiken Computation Laboratory, 1981.

[56] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using abstraction to improve fault tolerance. In *SOSP*, Oct. 2001.

[57] RouteViews. http://www.routeviews.org/, 2006.

[58] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computer Surveys*, 22(4), Dec. 1990.

[59] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: eventually consistent byzantine-fault tolerance. In *NSDI*, apr 2009.

[60] TechCrunch. Facebook source code leaked. http://www.techcrunch.com/2007/08/11/facebook-source-code-leaked/, Aug. 2007.

[61] F. Travostino and R. Shoup. eBay's scalability odyssey: Growing and evolving a large ecommerce site. In *LADIS*, Sept. 2008.

[62] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating Byzantine faults in database systems using commit barrier scheduling. In *SOSP*, Oct. 2007.

[63] D. Wendlandt, D. G. Andersen, and A. Perrig. Perspectives: Improving SSH-style host authentication with multi-path probing. In *USENIX Annual*, June 2008.

[64] B. Wester, J. Cowling, E. B. Nightingale, P. M. Chen, J. Flinn, and B. Liskov. Tolerating latency in replicated state machines through client speculation. In *NSDI*, Apr. 2009.

[65] T. Wood, R. Singh, A. Venkataramani, and P. Shenoy. ZZ: Cheap practical bft using virtualization. Technical Report TR14-08, University of Massachusetts, 2008.

[66] Yahoo! Hadoop Team. Zookeeper. http://hadoop.apache.org/zookeeper/, 2009.

[67] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *SOSP*, Oct. 2003.

Carousel: Scalable Logging for Intrusion Prevention Systems

Vinh The Lam[†], *Michael Mitzenmacher*[⋆], *George Varghese*[†]

[†]*University of California, San Diego* [⋆] *Harvard University*
{vtlam,varghese}@cs.ucsd.edu *michaelm@eecs.harvard.edu*

Abstract

We address the problem of collecting unique items in a large stream of information in the context of Intrusion Prevention Systems (IPSs). IPSs detect attacks at gigabit speeds and must log infected source IP addresses for remediation or forensics. An attack with millions of infected sources can result in hundreds of millions of log records when counting duplicates. If logging speeds are much slower than packet arrival rates and memory in the IPS is limited, *scalable logging* is a technical challenge. After showing that naïve approaches will not suffice, we solve the problem with a new algorithm we call Carousel. Carousel randomly partitions the set of sources into groups that can be logged without duplicates, and then cycles through the set of possible groups. We prove that Carousel collects almost all infected sources with high probability in close to optimal time as long as infected sources keep transmitting. We describe details of a Snort implementation and a hardware design. Simulations with worm propagation models show up to a factor of 10 improvement in collection times for practical scenarios. Our technique applies to *any* logging problem with non-cooperative sources as long as the information to be logged appears repeatedly.

1 Introduction

With a variety of networking devices reporting events at increasingly higher speeds, how can a network manager obtain a coherent and succinct view of this deluge of data? The classical approach uses a *sample* of traffic to make behavioral inferences. However, in many contexts the goal is *complete or near-complete collection* of information — MAC addresses on a LAN, infected computers, or members of a botnet. While our paper presents a solution to this abstract logging problem, we ground and motivate our approach in the context of Intrusion Prevention Systems.

Originally, Intrusion Detection Systems (IDSs) implemented in software worked at low speeds, but modern Intrusion Prevention Systems (IPSs) such as the Tipping Point Core Controller and the Juniper IDP 8200 [5] are implemented in hardware at 10 Gbps and are standard in many organizations. IPSs have also moved from being located only at the periphery of the organizational network to being placed throughout the organization. This allows IPSs to defend against internal attacks and provides finer granularity containment of infections. Widespread, cost-effective deployment of IPSs, however, requires using streamlined hardware, especially if the hardware is to be integrated into routers (as done by Cisco and Juniper) to further reduce packaging costs. By streamlined hardware, we mean ideally a single chip implementation (or a single board with few chips) and small amounts of high-speed memory (less than 10 Mbit).

Figure 1 depicts a logical model of an IPS for the purposes of this paper. A bad packet arrives carrying some key. Typically the key is simply the source address, but other fields such as the destination address may also be used. For the rest of the paper we assume the key is the IP source address. (We assume the source information is not forged. Any attack that requires the victim to reply cannot use a forged source address.) The packet is coalesced with other packets for the same flow if it is a TCP packet, normalized [16] to guard against evasions, and then checked for whether the packet is indicative of an attack. The most common check is *signature-based* (e.g., Snort [13]) which determines whether the packet content matches a regular expression in a database of known attacks. However, the check could also be *behavior-based*. For example, a denial of service attack to a destination may be detected by some state accumulated across a set of past packets.

In either case, the bad packet is typically dropped, but the IPS is required to *log* the relevant information on disk at a remote management console for later analysis and reporting. The information sent is typically the key K plus a report indicating the detected attack. Earlier work has shown techniques for high speed implementations of reassembly [4],

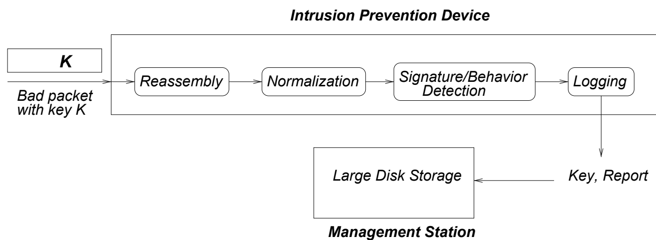


Figure 1: IPS logical model including a logging component that is often implemented naïvely

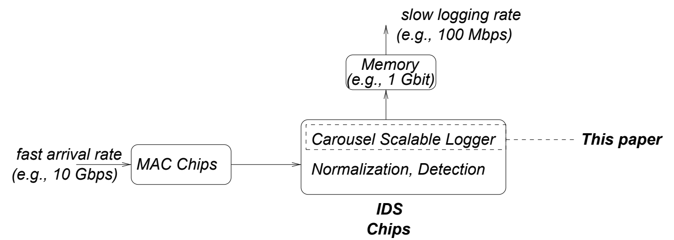


Figure 2: IPS hardware model in which we propose adding a scalable logger facility called Carousel. Carousel focuses on a small random subset of the set of keys at one time, thereby matching the available logging speed.

normalization [15, 16], and fast regular expression matching (e.g., [12]). However, to the best of our knowledge, there is no prior work in scalable logging for IPS systems or networking.

To see why logging may be a bottleneck, consider Figure 2, which depicts a physical model of a streamlined hardware IPS implementation, either stand-alone or packaged in a router line card. Packets arrive at high speed (say 10 Gbps) and are passed from a MAC chip to one or more IDS chips that implement detection by for example signature matching. A standard logging facility, such as in Snort, logs a report each time the source sends a packet that matches an attack signature and writes it to a memory buffer, from which it is written out later either to locally attached disk in software implementations or to a remote disk at a management station in hardware implementations. A problem arises because the logging speed is often much slower than the bandwidth of the network link. Logging speeds less than 100 Mbps are not uncommon, especially in 10 Gbps IDS line cards attached to routers. Logging speeds are limited by physical considerations such as control processor speeds and disk bandwidths. While logging speeds can theoretically be increased by striping across multiple disks or using a network service, the increased costs may not be justified in practice.

In hardware implementations where the memory buffer is necessarily small for cost considerations, the memory can fill during a large attack and newly arriving logged records may be dropped. A typical current configuration might include only 20 Mbits of on-chip high speed SRAM of which the normalizer itself can take 18 Mbits [16]. Thus, we assume that the logger may be allocated only a small amount of high speed memory, say 1 Mbit. Note that the memory buffer may include duplicate records already in the buffer or previously sent to the remote device.

Under a standard naïve implementation, unless the logging rate matches the arrival rate of packets, there is no guarantee that all infected sources will be logged. It is easy to construct worst-case timing patterns where some set of sources A are never logged because another set of sources B always reaches the IDS before sources in the set A and fills the memory. Even in a random arrival model, intuitively as more and more sources are logged, it gets less and less probable that a new unique source will be logged. In Section 3 we show that, even with a fairly optimistic random model, a standard analysis based on the coupon collector's problem (e.g., [8]) shows that the expected time to collect all N sources is a *multiplicative* factor of $\ln N$ worse than the optimal time. For example, when N is in the millions, which is not unusual for a large worm, the expected time to collect all sources can be 15 times larger than optimal. We also show similar poor behavior of the naïve implementation, both through analysis and simulation, in more complex settings.

The main contribution of this paper, as shown in Figure 2, is a scalable logger module that interposes between the detection logic and the memory buffer. We refer to this module and the underlying algorithm as *Carousel*, for reasons that will become apparent. Our logger is scalable in that it can collect almost all N sources with high probability with very small memory buffers in close to optimal time, where here the optimal time is N/b with b being the logging speed. Further, Carousel is simple to implement in hardware even at very high speeds, adding only a few operations to the main processing path. We have implemented Carousel in software both in Snort as well as in simulation in order to evaluate its performance.

While we focus on the scalable logging problem for IPSs in this paper, we emphasize that the problem is a general one that can arise in a number of measurement settings. For example, suppose a network monitor placed in the core of an organizational network wishes to log all the IP sources that are using TCP Selective Acknowledgment option (SACK). In general, our mechanism applies to any monitoring setting where a source is identified by a predicate on a packet (e.g., the packet contains the SACK_PERMITTED option, or the packet matches the Slammer signature), memory is limited, and sources do not cooperate with the logging process. It does, however, require sources to keep transmitting packets with the predicate in order to be logged. Thus Carousel does

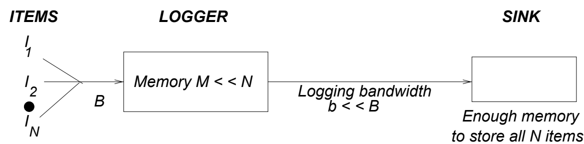


Figure 3: Abstract logging model: N keys to be logged enter the logging device repeatedly at a speed B that is much greater then the logging speed b and in a potentially adversarial timing pattern. At the same time, the amount of memory M is much less than the N, number of distinct keys to be logged. Source cooperation is *not* assumed.

not guarantee the logging of one-time events.

The rest of the paper is organized as follows. In Section 2 we describe a simple abstract model of the scalable logging problem that applies to many settings. In Section 3 we describe a simple analytical model that shows that even with an optimistic random model of packet arrivals, naïve logging can incur a multiplicative penalty of $\ln N$ in collection times. Indeed, we show this is the case even if naïve logging is enhanced with a Bloom filter in the straightforward way. In Section 4 we describe our new scalable logging algorithm Carousel, and in Section 5 we describe our Snort implementation. We evaluate Carousel using a simulator in Section 6 and using a Snort implementation in Section 7. Our evaluation tests both the setting of our basic analytical model, which assumes that all sources are sending at time 0, and a more realistic logistic worm propagation model, in which sources are infected gradually. Section 8 describes related work while Section 9 concludes the paper.

2 Model

The model shown in Figure 3 abstracts the scalable logging problem. First, there are N distinct keys that arrive repeatedly and with arbitrary timing frequency at a cumulative speed of B keys per second at the logger. There are *two* resources that are in scarce supply at the logger. First, there is a limited logging speed b (keys per second) that is much smaller than the bandwidth B at which keys arrive. Even this might not be problematic if the logger had a memory M large enough to hold all the distinct keys N that needed to be logged (using methods we discuss below, such as Bloom filters [1, 3], to handle duplicates), but in our setting of large infections and hardware with limited memory, we must also assume that $N >> M$.

Eliminating all duplicates before transmitting to the sink is *not* a goal of a scalable logger. We assume that the sink has a hash table large enough to store all N unique sources (by contrast to the logger) and eliminate duplicates.

Instead, the ultimate goal of the scalable logger is *near-complete collection*: the logging of all N sources. We now adopt some of the terminology of competitive analysis [2] to describe the performance of practical logger systems. The best possible logging time $T_{optimal}$ for an omniscient algorithm is clearly N/b. We compare our algorithms against this omniscient algorithm as follows.

Definition 2.1 *We say that a logging algorithm is* (ϵ, c)*-scalable if the time to collect at least* $(1-\epsilon)N$ *of the sources is at most* $cT_{optimal}$*. In the case of a randomized algorithm, we say that an algorithm is* (ϵ, c)*-scalable if in time* $cT_{optimal}$ *the expected number of sources collected is at least* $(1-\epsilon)N$*.*

Note that in the case $\epsilon = 0$ all sources are collected. While obviously collecting all sources is a desirable feature, some relaxation of this requirement can naturally lead to much simpler algorithms.

These definitions have some room for play. We could instead call a randomized algorithm (ϵ, c)-scalable if the expected time to collect at least $(1-\epsilon)N$ is at most $cT_{optimal}$, and we may be concerned only with asymptotic algorithmic performance as either or both of N/M and B/b grow large. As our focus here is on practically efficient algorithms rather than subtle differences in the definitions we avoid such concerns where the meaning is clear.

The main goal of this paper is to provide an effective and practical (ϵ, c)-scalable randomized algorithm. To emphasize the value of this result, we first show that simple naïve approaches are not (ϵ, c)-scalable for any constants $\epsilon, c > 0$. Our positive results will require the following additional assumption for our model:

Persistent Source Assumption: We assume that any distinct key X to be logged will keep arriving at the logger.

For sources infected by worms this assumption is often reasonable until the source is "disinfected" because the source continues to attempt to infect other computers. The time for remediation (days) is also larger than the period in which the attack reaches its maximum intensity (hours). Further, if a source is no longer infected, then perhaps it matters less that the source is not logged. In fact, we conjecture that no algorithm can solve the scalable logging problem without the Persistent Source assumption.

The abstract logger model is a general one and applies to other settings. In the introduction, we mentioned one other possibility, logging sources using SACK. As another example, imagine a monitor that wishes to log all the sources in a network. The monitor issues a broadcast request to all sources asking them to send a reply with their ID. Such messages do exist, for example the SYSID message in 802.1. Unfortunately, if all sources reply at the same time, some set of sources can consistently be lost.

Of course, if the sources could randomize their replies, then better guarantees can be made. The problem can be

viewed as one of congestion control: matching the speed of arrival of logged keys to the logging speed. Congestion control can be solved by standard methods like TCP slow start or Ethernet backoff *if* sources can be assumed to cooperate. However, in a security setting we cannot assume that sources will cooperate, and other approaches, such as the one we provide, are needed.

3 Analysis of a Naïve Logger

3.1 The Naïve Logger Alone

Before we describe our scalable logger and Snort implementation, we present a straw man naïve logger, and a theoretical analysis of the expected and worst-case times. The theoretical analysis makes some simplifications that only benefit the naïve logger, but still its performance is poor. The naïve logger motivates our approach.

We start with a model of the naïve logger shown in Figure 4. We assume that the naïve logger only has a memory buffer in the form of a queue. Keys, which again are usually source addresses, arrive at a rate of B per second. When the naïve logger receives a key, it is placed at the tail of the queue. If the queue is full, the key is dropped. The size of the queue is M. Periodically, at a smaller rate of b keys per second, the naïve logger sends the key (and any associated report) at the head of the queue to a disk log. Let L_D denote the set of keys logged to disk, and L_M the set of keys that are in the memory.

The naïve logger works very poorly in an adversarial setting. In an adversarial model, after the queue is full of M keys, and when an empty slot opens up at the tail, the adversary picks a duplicate key that is part of the M keys already logged. When the queue is full, the adversary cycles through the remaining unique sources to pick them to arrive and be dropped, thus fulfilling the persistent source assumption in which every source must arrive periodically. It is then easy to see the following result.

Theorem 3.1 Worst-case time for naïve logger: *The worst-case time to collect all N keys is infinity. In fact, the worst-case time to collect more than M keys is infinite.*

We believe the adversarial models can occur in real situations especially in a security setting. Sources can be synchronized by design or accident so that certain sources always transmit at certain times when the logger buffers are full. While we believe that resilience to adversarial models is one of the strengths of Carousel, we will show that even in the most optimistic random models, Carousel significantly outperforms a naïve logger.

The simplest random model for key arrival is one in which the next key to arrive is randomly chosen from the N possible keys, and we can find the expected collection time of the naïve logger in this setting.

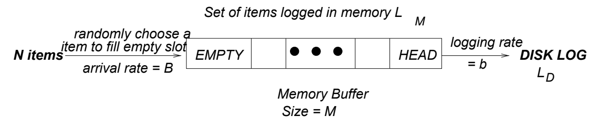


Figure 4: Model of naïve logging using an optimistic random model. When space opens up in the memory log, a source is picked uniformly and randomly from the set of all possible N sources. Unfortunately, that source may already be in the memory log (L_M) or in the disk log (L_D). Thus as more sources are logged it gets increasing less probable that a new unique source will be logged, leading to a logarithmic increase in collection time over optimal

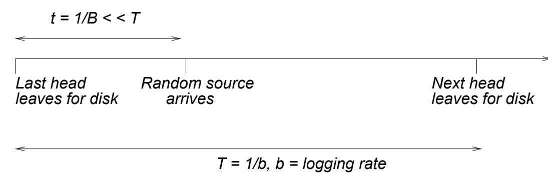


Figure 5: Portion of timeline for random model shown in Figure 4. We divide time into cycles of time T where T is the time to send one piece of logged information at the logging rate b. The time for a new randomly chosen source to first arrive is much smaller $t = 1/B$, where B is the faster packet arrival rate.

Let us assume that $M < B/b$, so that initially the queue fills entirely before the first departure. (The analysis is easily modified if this is not the case.) Figure 5 is a timeline which shows that the dynamics of the system evolve in cycles of length T seconds, where $T = 1/b$. Every T seconds the current head of the memory queue leaves for the disk log, and within the smaller time $t = 1/B$, a new randomly selected key arrives to the tail of the queue. In other words, the queue will always be full except when a key leaves from the head, leaving a single empty slot at the tail as shown in Figure 4. The very next key to be selected will then be chosen to fill that empty slot as shown in Figure 5.

The analysis of this naïve setting now follows from a standard analysis of the coupon collector's problem [8]. Let $L = L_M \cup L_D$ denote the set of unique keys logged in either memory or disk. Let T_i denote the time for L to grow from size $i - 1$ to i (in other words, the time for the i-th new key to be logged). If we optimistically assume that the first M keys that arrive are distinct, we have $T_i = T$ for $1 \leq i \leq M$, as the queue initially fills. Subsequently, since the newly arriving key is chosen randomly from the set of N keys, it will get increasingly probable (as i gets larger) that the chosen key already belongs to the logged set L.

The probability that a new key will not be a duplicate of $i - 1$ previously logged keys is is $P_i = (N - i + 1)/N$. If

a key is a duplicate the naïve logger simply wastes a cycle of time T. (Technically, it might be $T - t$ where $t = 1/B$, but this distinction is not meaningful and we ignore it.) The expected number of cycles before the i-th key is not a duplicate is the reciprocal of the probability or $1/P_i$. Hence for $i > M, i \leq N$ the expected value of T_i is

$$\mathbf{E}(T_i) = \frac{N}{b(N - i + 1)}.$$

Using the linearity of expectation, the collection time for the last $N - M$ keys is

$$\sum_{i=M+1}^{N} \frac{N}{b(N-i+1)} = \frac{N}{b} \sum_{j=1}^{N-M} \frac{1}{j} = \frac{N}{b} \left(\ln(N - M) + O(1) \right),$$

using the well-known result for the sum of the harmonic series. Hence if we let $T^{naive}_{collect}$ be the time to collect all N keys for the naïve collector then $T^{naive}_{collect} > \frac{N}{b} \ln(N - M)$, and so the naïve logger is a multiplicative factor of $\ln(N - M)$ worse than the optimal algorithm.

It might be objected that it is not clear that N/b is in fact the optimal time in this random model, and that this $\ln N$ factor is due entirely to the embedded coupon collector's problem arising from the random model. For example, if $B = b = 1$, then you cannot collect the N keys in time N, since they will not all appear until after approximately $N \ln N$ keys have passed [8]. However, as long as $B/b > \ln N$ (and $M > 1$), for any $\gamma > 0$, with high probability an omniscient algorithm will be able to collect all keys after at most $(1 + \gamma)NB/b$ keys have passed in this random model, so the optimal collection time can be made arbitrarily close to N/b. Hence, this algorithm is indeed not truly scalable in the sense we desire, namely in a comparison with the optimal omniscient algorithm.

Even if we seek only to obtain $(1 - \epsilon)N$ keys, by the same argument we have the collection time is

$$\frac{N}{b} \left(\ln((1 - \epsilon)N - M) + O(1) \right).$$

Hence when $M = o(N)$, the logger is still not (ϵ, c)-scalable for any constants ϵ and c. We can summarize the result as follows:

Theorem 3.2 Expected time for naïve logger: *The expected time to collect $(1 - \epsilon)N$ keys is at least a multiplicative factor of $\ln((1 - \epsilon)N - M)$ worse than the optimal time for sufficiently large N, M, and ratios B/b.*

As stated in the introduction, for large worm outbreaks, the naïve logger can be prohibitively slow. For example, as $\ln 1,000,000$ is almost 14, if the optimal time to log 1 million sources is 1 hour, the naïve logger will take almost 14 hours.

The results for the random model can be extended to situations that naturally occur in practice and appear somewhere between the random model and an adversarial model. For example, suppose that we have two sets of sources, of sizes N_1 and N_2, but the first source sends at a speed that is j times the second. This captures, at a high level, the issue that sources may be sending at different rates. We assume each source individually behaves according to the random model. Let T_1 be the expected time to collect all the keys in the fast set, and T_2 the expected time for the slow set. Then clear the expected time to collect all sources is at least $\max(T_1, T_2)$, and indeed this lower bound will be quite tight when T_1 and T_2 are not close. As an example, suppose $N_1 = N_2 = N/2$, and $j > 1$. Then T_2 is approximately

$$\frac{N(j+1)}{2b} \ln \left(\frac{N}{2} - \frac{M}{j+1} \right).$$

The time to collect in this case is dominated by the slow sources, and is still a logarithmic factor from optimal.

3.2 The Naïve Logger with a Bloom Filter

A possible objection is that our naïve logger is far too naïve. It may be apparent to many readers that additional data structures, such as a Bloom filter, could be used to prevent logging duplicate sources and improve performance. This is true, and we shall use such measures in our scalable approaches. However, we point out that as the Bloom filter of limited size, it cannot by itself prevent the problems of the naïve logger, as we now explain.

To frame the discussion, consider 1 million infected sources that keep sending to an IPS. The solution to the problem may appear simple. First, since all the sources may arrive at a very fast rate of B before even a few are logged, the scheme must have a memory buffer that can hold keys waiting to be logged. Second, we need a method of avoiding sending duplicates to the logger, specifically one that takes small space, in order to make efficient use of the small speed of the logger.

To avoid sending duplicates, one naturally would think of a solution based on Bloom filters or hashed fingerprints. (We assume familiarity with Bloom filters, a simple small-space randomized data structure for answering queries of the form "Is this an item in set X" for a given set X. See [3] for details.) For example, we could employ a Bloom filter as follows. For concreteness, assume that a source address is 32 bits, the report associated with a source is 68 bits, and that we use a Bloom filter [1] of 10 bits per source.[1] Thus we need a total of 100 bits of memory for each source waiting to be logged, and 10 bits for each source that has been logged. (Instead of a Bloom filter, we could keep a table of hash-based fingerprints of the sources, with different tradeoffs but similar results, as we discuss in Section 4.2.2.)

[1] This is optimistic because many algorithms would require not just a Bloom filter but instead a counting Bloom filter [7] to support deletions, which would require more than 10 bits per entry.

Unfortunately, the memory buffer and Bloom filter have to operate at Gigabit speeds. Assume that the amount of IDS high speed memory is limited to storing say 1 Mbit. Then, assuming 100 bits per source, the IPS can only store information about a burst of 10,000 sources pending their transmission to a remote disk. This does not include the size of the Bloom filter, which can only store around 100,000 sources if scaled to 1 Mbit of size; after this point, the false positive rate starts increasing significantly. In practice one has to share the memory between the sources and the Bloom filter.

The inclination would be to clear the Bloom filter after it became full and start a second phase of logging. One concern is that timing synchronization could result in the same sources that were logged in phase 1 being logged and filling up the Bloom filter again, and this could happen repeatedly, leading to missing several sources. Even without this potential problem, there is danger in using a Bloom filter, as we can see by again considering the random model.

Consider enhancing the naïve logger with a Bloom filter to prevent the sending of duplicates. We assume the Bloom filter has a counter to track the number of items placed in the filter, and the filter is cleared when the counter reaches a threshold F to prevent too many false positives. Between each clearing, we obtain a group of F distinct random keys, but keys may be appear in multiple groups. Effectively, this generalizes the naïve logger, which simply used groups of size $F = 1$.

Not surprisingly, this variation of the coupon collector's problem has been studied; it is know as the coupon subset collection problem, and exact results for the problem are known [11, 14]. Details can be examined by the interested reader. A simple analysis, however, shows that for reasonable filter sizes F, there will be little or no gain over the naïve logger. Specifically, suppose $F = o(\sqrt{N})$. Then in the random model, the well-known birthday paradox implies that with high probability the first F keys to be placed in the Bloom filter will be distinct. While there may still be false positives from the Bloom filter, for such F the filter fills without detecting any true duplicates with high probability. Hence, in the random case, the expected collection time even using a Bloom filter of this size is still $\frac{N}{b}\ln(N - M) + O(1)$. With larger filters, some true duplicates will be suppressed, but one needs very large filters to obtain a noticeable gain. The essential point of this argument remains true even in the setting considered above where different sets of sources arrive at different speeds.

The key problem here is that we cannot supply the IDS with the list of all the sources that have been logged, even using a Bloom filter or a hashed set of fingerprints. Indeed, when $M << N$ no data structure can track a meaningful fraction of the keys that have already been stored to disk. Our solution to this problem is to partition the population of keys to be recorded into subsets of the right size, so that the logger can handle each subset without problem. The logger then iterates through all subsets in *phases*, as we now describe. This repeated cycling through the keys is reminiscent of a Carousel, yielding our name for our algorithm.

4 Scalable logging using Carousel

4.1 Partitioning and logging

Our goal is to partition the keys into subsets of the right size, so that during each phase we can concentrate on a single subset. The question is how to perform the partitioning. We want the size of each partition to be the right size for our logger memory, that is approximately size M. We suggest using a randomized partition of the sources into subsets using a hash function that uses very little memory and processing. This randomized partitioning would be simple if we initially knew the population size N, but that generally will not be the case; our system must find the current population size N, and indeed should react as the population size changes.

We choose a hash-based partition scheme that is particularly memory and time-efficient. Let $H(X)$ be a hash function that maps a source key X to an r-bit integer. Let $H_k(X)$ be the lower order k bits of $H(X)$. The size of the partition can be controlled by adjusting k.

For example, if $k = 1$, we divide the sources into two subsets, one subset whose low order bit (after hashing) is 1, and one whose lower order bit is a 0. If the hash function is well-behaved, these two sets will be approximately half the original size N. Similarly, $k = 2$ partitions the sources approximately into four equally sized subsets whose hash values have low order bits 00, 01, 10, and 11 respectively. This allows only very coarse-grained partitioning, but that is generally suitable for our purposes, and the simplicity of using the lower order k bits of $H(X)$ is particularly compelling for implementation and analysis. To begin we will assume the population size is stable but unknown, in which case the basic Carousel algorithm can be outlined as follows:

- *Partition:* Partition the population into groups of size 2^k by placing all sources which have the same value of $H_k(X)$ in the same partition.
- *Iterate:* A phase is assigned time $T_{phase} = M/b$ which is the time to log M sources, where M is the available memory in keys and b is the logging time. The i-th phase is defined by logging only sources such that $H_k(s) = i$. Other sources are automatically dropped during this phase. The algorithm must also utilize some means of preventing the same source from being logged multiple times in the phase, such as a Bloom filter or hash fingerprints.

- *Monitor:* If during phase i, the number of keys that match $H_k() = i$ exceeds a high threshold, then we return to the Partition step and increase k. While our algorithms typically use $k = k + 1$, higher jumps can allow faster response. If the number of number of keys that match $H_k() = i$ falls below a low threshold, then we return to the Partition step and decrease k.

In other words, Carousel initially tries to log all sources without hash partitioning. If that fails because of memory overflow, the algorithm then works on half the possible sources in a phase. If that fails, it works on a quarter of the possible sources, and so on. Once it determines the appropriate partition size, the algorithm iterates through all subsets to log all sources.

As described, we could in the monitoring stage change k by more than 1 if our estimate of the number of keys seen during that phase suggests that would be an appropriate choice. Also, of course, we can choose to decrease k if our estimate of the keys in that phase is quite small, as would happen if we are logging suspected virus sources and these sources are stopped. There are many variations and optimizations we could make, and some will be explored in our experiments. The important idea of Carousel, however, is to partition the set of keys to match the logger memory size, updating the partition as needed.

4.2 Collection Times for Carousel

We assume that the memory includes, for each key to be recorded, the space for the key itself, the corresponding report, and some number of bits for a Bloom filter. This requires slightly more memory space that we assumed when analyzing the random model, where we did not use the Bloom filter. The discrepancy is small, as we expect the Bloom filter to be less than 10% of the total memory space (on the order of 10 bits or less per item, against 100 or more bits for the key and report). This would not effectively change the lower bounds on performance of the naïve logger. We generally ignore the issue henceforth; it should be understood that the Bloom filter takes a small amount of additional space.

Recall that Carousel has 3 components: partition, iterate, and monitor. Faced with an unknown population N, the scalable logger will keep increasing the number of bits chosen k until each subset is less than size M, the memory size available for buffering logged keys.

We sketch an optimistic analysis, and then correct for the optimistic assumptions. Let us assume that all N keys are present at the start of time, that our hash function splits the keys perfectly equally, and that there is no failed recording of keys due to false positives from the Bloom filter (or whatever structure suppresses duplicates). In that case it will take at most $\lceil \log_2 \frac{N}{M} \rceil$ partition steps for Carousel to get the right number of subsets. Each such step required time for a single logging phase, $T_{phase} = M/b$. The logger then reaches the right subset size, so that k is the smallest value such that $N/2^k \leq M$. The collector then goes through 2^k phases to collect all N sources. Note that $2^k \leq 2N/M$, or else k would not be the smallest value with $N/2^k \leq M$. Hence, after the initial phases to find the right value of k, the additional collection time required is just $2N/b$, or a factor of two more than optimal. The total time is thus at most

$$\frac{M\lceil \log_2(N/M) \rceil}{b} + \frac{2N}{b},$$

and the generally the second term will dominate the first. Asymptotically, when $N >> M$, we are roughly within a factor of 2 of the optimal collection time.

Note that the factor of 2 in the $2N/b$ term could in fact be replaced in theory by any constant $a > 1$, by increasing the number of sets in the partition by a factor of a rather than 2 at each partition step. This would increase the number of partition steps to $\lceil \log_a \frac{N}{M} \rceil$. In practice we would not want to choose a value of a too close to 1, because keys will not be partitioned equally into sets, as we describe in the next subsection. Also, as we have described a factor of 2 is convenient in terms of partitioning via the low order bits of a hash. In what follows we continue to use the factor 2 in describing our algorithm, although it should be understood smaller constants (with other tradeoffs) are possible.

In some ways our analysis is actually pessimistic. Early phases that fail can still log some items, and we have assumed that we could partition to require $2N/M$ phases, when generally the number of phases required will be smaller. However, we have also made some optimistic assumptions that we now revisit more carefully.

4.2.1 Unequal Partitioning: Maximum Subset Analysis

If the logger uses k bits to partition keys, then there are $K = 2^k$ subsets. While the expected number of sources in a subset is $\frac{N}{K}$, even assuming a perfectly random hash function, there may be deviations in the set sizes. Our algorithm will actually choose the value of k such that the biggest partition is fit in our memory budget M, not the average partition, and we need to take this into account. That is, we need to analyze the *maximum* number of keys being assigned to a subset at each phase interval.

In general, this can be handled using standard Chernoff bound analysis [8]. In this specific case, for example, [10] proves that with very high probability, the maximum number of sources in any subset is less than $\frac{N}{K} + \sqrt{\frac{2N \ln K}{K}}$. Therefore we can assume that the smallest integer k satisfying

$$\frac{N}{K} + \sqrt{\frac{2N \ln K}{K}} \leq M, \quad (1)$$

where $K = 2^k$, is greater than or equal to the k eventually

found by the algorithm.

Note that the difference between our optimistic analysis, where we required the smallest k such that $N/K \leq M$, and this analysis is generally very small, as $\sqrt{\frac{2N \ln K}{K}}$ is generally much less than N/K. That is, suppose that $N/K \leq M$, but $\frac{N}{K} + \sqrt{\frac{2N \ln K}{K}} > M$, so that at some point we might increase the value k to more than the smallest value such that $N/K \leq M$, because we unluckily have a subset in our partition that is bigger than the memory size. The key here is that in this case $N/K \approx M$, or more specifically

$$M \geq \frac{N}{K} > M - \sqrt{\frac{2N \ln K}{K}},$$

so that our collection time is now

$$\frac{2KM}{b} < \frac{2N}{b} + \frac{2}{b}\sqrt{\frac{2N \ln K}{K}}.$$

That is, the collection time is still, at most, very close to $2N/b$, with the addition of a smaller order term that contributes negligibly compared to $2N/b$ for large N. Hence, asymptotically, we are still with a factor of c of the optimal collection time, for any $c > 2$.

4.2.2 Effects of False Positives

So far, our analysis has not taken into account our method of suppressing duplicates. One natural approach is to use a Bloom filter, in which case false positives can lead to a source not being logged in a particular phase. This explains our definition of an (ϵ, c)-scalable logger. We have already seen that c can be upper bounded by any number larger than 2 asymptotically. Here ϵ can be bounded by the false positive rate of the corresponding Bloom filter. As long as the number of elements per phase is no more than $M' = \frac{N}{K} + \sqrt{\frac{2N \ln K}{K}}$ with high probability, then given the number of bits used for our Bloom filter, we can bound the false positive rate. For example, using $10M'$ bits in the Bloom filter, the false positive rate is less than 1%, so our logger asymptotically converges to a $(0.01, 2)$-scalable logger.

We make note of some additions one can make to improve the analysis. First, this analysis assumes only a single *major cycle* that logs each subset in the partition once. If one rerandomized the chosen hash functions each major cycle, then the probability a persistent source is missed each major cycle is independently at most ϵ each time. Hence, after two such cycles, the probability of a source being missed is at most ϵ^2, and so on.

Second, this analysis is pessimistic, in that in this setting, items are gradually added to an empty Bloom filter each phase; the Bloom filter is not in its full state at all times, so the false positive probability bound for the full filter is a large overestimate. For completeness we offer the following more refined analysis (which is standard) to obtain the expected false positive rate. (As usual, the actual rate is concentrated around its expectation with high probability.)

Assume the Bloom filter has m bits and uses h hash functions. Consider whether the $(i+1)$st item added to the filter causes a false positive. First consider a particular bit in the Bloom filter. The probability that it is not set to 1 by one of the hi hash functions thus far is $(1 - \frac{1}{m})^{hi}$. Therefore the probability of a false positive at this stage is $(1 - (1 - \frac{1}{m})^{hi})^h \approx (1 - e^{-\frac{hi}{m}})^h$.

Suppose M' items are added into the Bloom filter within a phase interval. The expected fraction of false positives is then (approximately) $\sum_{i=0}^{M'-1}(1 - e^{-\frac{hi}{m}})^h$, compared to the $(1 - e^{-\frac{hM'}{m}})^h$ given by the standard analysis for the false positive rate after M' elements have been added. As an example, with $M' = 312$, $h = 5$, and $m = 5000$, the standard analysis gives a false positive rate of $1.4 \cdot 10^{-3}$, while our improved analysis gives a false positive rate of $2.5 \cdot 10^{-4}$.

Third, if collecting all or nearly all sources is truly paramount, instead of using a Bloom filter, one can use hash-based fingerprints of the sources instead. This requires more space than a Bloom filter ($\Theta(\log M')$ bits per source if there are M' per phase) but can reduce the probability of a false positive to inverse polynomial in M'; that is, with high probability, all sources can be collected. We omit the standard analysis.

4.2.3 Carousel and Dynamic Adaptation

Under our persistent source assumption, any distinct key keeps arriving at the logger. In fact, for our algorithm as described, we need an even stronger assumption: each key must appear during the phase in which it is recorded, which means each key should arrive every N/b steps. Keys that do not appear this frequently may miss their phase and not be recorded. In most settings, we do not expect this to be a problem; any key that does not persist and appear this frequently does not likely represent a problematic source in terms of, for example, virus outbreaks. Our algorithm could be modified for this situation in various ways, which we leave as future work. One approach, for example, would be to sample keys in order to estimate the 95% percentile for average interarrival times between keys, and set the time interval for the phase time to gather a subset of keys accordingly.

A more pressing issue is that the persistent source assumption may not hold because external actions may shut down infected sources, effectively changing the size of the set of keys to record dynamically. For example, during a worm outbreak, the number of infected sources rises rapidly at first but then they can go down due to external actions (for example, network congestion, users shutting down slow machines due to infection, and firewalling traffic or blocking a

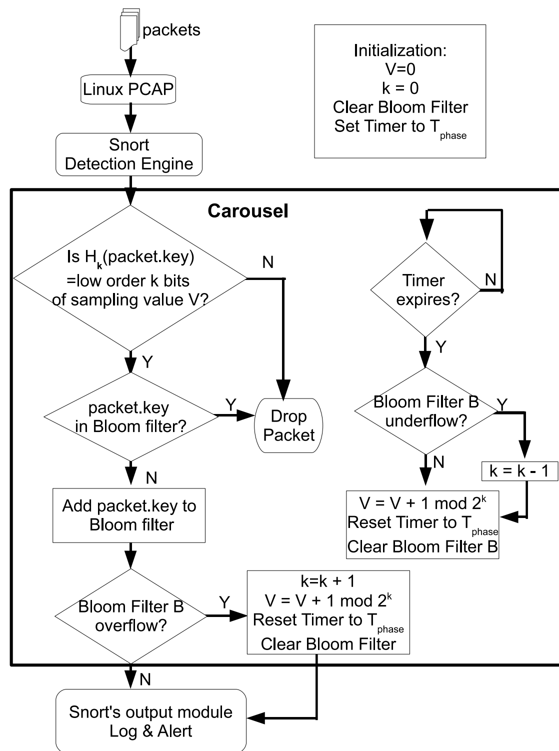


Figure 6: Flowchart of Carousel within Snort packet flow

part of the network). In that case, the scalable logger may pick a large number of sampling bits k at first due to large outbreak traffic. However, the logger should correspondingly increase the value of k subsequently as the number of sources to record declines, to avoid inefficient logging based on too large a number of phases.

5 Carousel Implementations

We describe our Snort evaluation in Section 5.1 and a sketch of a hardware implementation in Section 5.2.

5.1 Snort Implementation

In this section, we describe our implementation of Carousel integrated into the Snort [13] IDS. We need to first understand the packet processing flow within Snort to see where we can interpose the Carousel scalable logger scheme. As in Figure 6, incoming packets are captured by *libpcap*, queued in a kernel buffer, and then processed by the callback function *ProcessPacket*.

ProcessPacket first passes the packet to preprocessors, which are components or plug-ins serving to filter out suspicious activity and prepare the packet to be further analyzed. The detection engine then matches the packet against the rules loaded during Snort initialization. Finally, the Snort output module performs appropriate actions such as logging to files or generating alerts. Note that Snort is designed to be strictly single-threaded for multiplatform portability.

The logical choice is to place Carousel module between the detection engine and output module so that the traffic can either go directly to the output plugin or get diverted through the Carousel module. We cannot place the logger module before the detection engine because we need to log only after a rule (e.g., a detected worm) is matched. Similarly, we cannot place the logger after the output module because by then it is too late to affect which information is logged. Our implementation also allows a rule to bypass Carousel if needed and go directly to the output module.

Figure 6 is a flowchart of Carousel module for Snort interposed between the detection engine and the output model. The module uses the variables $T_{phase} = M/b$ (time for each phase) and k (number of sampling bits) described in Section 4.1. M is the number of keys that can be logged in a partition and b is the logging rate; in our experiments we use $M = 500$. The module also uses a 32-bit integer V that represents the hash value corresponding to the current partition. Initially, $k = 0$, $V = 0$, the Bloom filter is empty, and a timer T is set to fire after T_{phase}. The Bloom filter uses 5000 bits, or 10 bits per key that can fit in M, and employs 5 hash functions (SDBM, DJP, DEK, JS, PJW) taken from [9].

The Carousel scalable logger first compares the low-order k bits of the hash of the packet key (we use the IP source address in all our experiments) to the low order k bits of V. If they do not match, the packet is not in the current partition and is not passed to the output logging. If the value matches but the key yields a positive from the Bloom filter (so it is either already logged, or a false positive), again the packet is not passed to the output module. If the value matches and the key does not yield a positive from the Bloom filter, then the module adds the key to the Bloom filter. If the Bloom filter overflows (the number of insertions exceeds M), then k is incremented by 1, to create smaller size partitions.

When the timer T expires, a phase ends. We first check for underflow by testing whether the number of insertions is less than M/x. We found empirically that a factor $x = 2.3$ worked well without causing oscillations. (A value slightly larger than 2 is sensible, to prevent oscillating because of the variance in partition sizes.) If there is no underflow, then the sampling value V is increased by 1 mod 2^k to move to the next partition.

5.2 Hardware Implementation

Figure 7 shows a schematic of the base logic that can be inserted between the detector and the memory buffer used to store log records in an IPS ASIC. Using 1 Mbit for the Bloom filter, we estimate that the logic takes less than 5% of a low-end 10mm by 10 mm networking ASIC. All re-

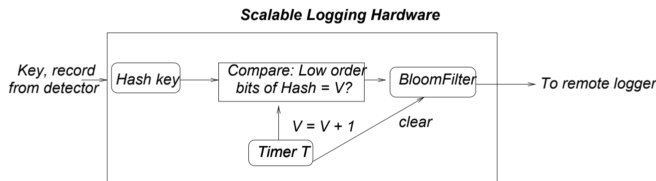


Figure 7: Schematic of the Carousel Logger logic as part of an IPS Chip.

sults are reported for a standard 400 Mhz 65 nm process currently being used by networking vendors. The logic is flow-through: in other words, it can inserted between the detector and logging logic without changing any other logic. This allows the hardware to be incrementally deployed *within* an IPS without changing existing chip sets.

We assume the detector passes a key (e.g., a source IP address) and a detection record (e.g., signature that matched) to the first block. The hash blocks computes a 64-bit hash of the key. Our estimates use a Rabin hash whose loop is unrolled to run at 40 Gbps using 20K gates.

The hash output supplies a 64-bit number which is passed to the Compare block. This block masks out the low-order k bits of the hash (a simple XOR) and then compares it (comparator) to a register value V that denotes the current hash value for this phase. If the comparison fails, the log attempt is dropped. If it succeeds, the key and record are passed to the Bloom filter logic. This is the most expensive part of the logic. Using 1 Mbit of SRAM to store the Bloom filter and 3 parallel hash functions (these can be found by taking bits 1-20, 21-40, 41-60 etc of the first 64-bit hash computed without any further hash computations), the Bloom filter logic takes less than a few percent of a standard ASIC.

As in the Snort implementation, a periodic timer module fires every $T_{phase} = M/b$ time and causes the value V to be incremented. Thus the remaining logic other than the Bloom filter (and to a smaller extent the hash computation) is very small. We use two copies of the Bloom filter and clear one copy while the other copy is used in a phase. The Bloom filter should be able to store a number of keys equal to the number of keys that can be stored in the memory buffer. Assuming 10 bits per entry, a 1 Mbit Bloom filter allows approximately 100,000 keys to be handled in each phase with the targeted false positive probability. Other details (underflow, overflow etc.) are similar to the Snort implementation and are not described here.

6 Simulation Evaluation

To evaluate Carousel under more realistic settings in which the population grows, we simulate the logger behavior when faced with a typical worm outbreak as modeled by a logistic

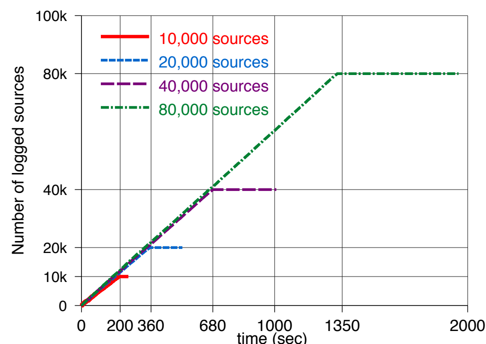


Figure 8: Performance of Carousel with different logging populations

equation. We used a discrete event simulation engine that is a stripped down (for efficiency) version of the engine found in ns-2. We implement the Carousel scalable logger as described in Section 4. The simulated logger maintains the sampling bit count k and only increases k when the Bloom filter overflows; k stabilizes when all sources sampled during T_{phase} fit the into memory budget M with logging speed b. Simulation allows us to investigate the effect of various input parameters such as varying worm speed and whether the worm uses a hit list. Again, in all the simulations below, the Bloom filter uses 5000 bits and 5 hash functions (SDBM, DJP, DEK, JS, PJW) taken from [9]. For each experiment, we plot the average of 50 runs of simulation.

We start by confirming the theory with a baseline experiment in Section 6.1 when all sources are present at time 0. We examine the performance of our logger with the logistic model in Section 6.2. We evaluate the impact of non-uniform source arrivals in Section 6.3. In Section 6.4, we examine a tradeoff between using a smaller number of bits per Bloom filter element and taking more more major cycles to collect all sources. Finally, in Section 6.5, we demonstrate the benefit of reducing k in the presence of worm remediation.

6.1 Baseline Experiment

In Figure 8, we verify the underlying theory of Carousel in Section 4 assuming all sources are present at time 0. We consider various starting populations $N = 10000$ to 80000 sources, a memory budget of $M = 500$ items, and a logging speed $b = 100$ items per second.

Figure 8 shows that the Carousel scalable logger collects *almost all* (at least 99.9%) items by $t = 189, 354, 679$ and 1324 seconds for $N = 10000, 20000, 40000$ and 80000 respectively. This is no more than $\frac{2N}{b}$ in all cases, matching the predictions of our optimistic analysis in Section 4.

With these settings, the $10,000$ sources will be partitioned into 32 subsets, each of size approximately 312 (in

expectation). In fact, our experiment trace shows that the number of sources per phase is in the range of 280 to 340. Since the Bloom filter uses 5000 bits, essentially we have more than 10 bits per item once the right number of partitions is found. As we calculated previously (in Section 4.2.2), the accumulated false positive rate of 312 sources in a 5000-bit Bloom filter with 5 hash functions is $2.5 \cdot 10^{-4}$. We also verified that most phases have no false positives. However, the Carousel algorithm may need additional major cycles to collect these remaining sources. Since a major cycle is 2^k iterations, the theory predicts that Carousel requires more time to collect missed false positives for larger k and hence for larger N. We observe that the length of horizontal segment of each curve in Figure 8, which represents the collection time of all sources missed in the first major cycle, is longer for larger populations N.

6.2 Logger Performance with Logistic Model

In the logistic model, a worm is characterized by H, the size of the initial hit list, the scanning rate, and a probability p of a scan infecting a vulnerable node. In our simulations below, we use a population of $N = 10,000$, a memory size $M = 500$ with Bloom filter and $M = 550$ without Bloom filter, and logging speed $b = 100$ packets/sec; the best possible logging time to collect all sources is $N/b = 100$ seconds.

For our first 3 experiments, shown in Figures 9, 10 and 11, we use an initial hit list of $H = 10,000$. Since the hit list is the entire population, as in the baseline, all sources are infected at time $t = 0$. We use these simulations to see the effect of increasing the scan rate and monitoring ability assuming all sources are infected. Our subsequent experiments will assume a much smaller hit list, more closely aligned with a real worm outbreak.

For the first experiment, shown in Figure 9 we use 6 scans per second (to model a worm outbreak that matches the Code Red scan rate [17]) and $p = 0.01$. Figure 9 shows that Carousel needs 200 seconds to collect the $N = 10,000$ sources whereas the naïve logger takes $4,000$ seconds. Further, the difference between Carousel and the naïve logger increases with the fraction of sources logged. For example, Carousel is 6 times faster at logging 90% level of all sources but 20 times faster to log 100% of all sources. This is consistent with the analysis in Section 3.1.

In Figure 10 we keep all the same parameters but increase the scan rate ten times to 60 scans/sec. The higher scan rate allows naïve logging a greater chance to randomly sample packets and so the difference between scalable and naïve logging is less pronounced. Figure 11 uses the same parameters as Figure 9 but assumes that only 50% of the scanning packets are seen by the IPS. This models the fact that a given IPS may not see all worm traffic. Notice again that the difference between naïve and Carousel logging decreases when the amount of traffic seen by the IPS decreases.

The remaining simulations assume a logistic model of worm growth starting with a hit list of $H = 10$ infected sources when the logging process starts. The innermost curve illustrates the infected population versus time, which obeys the well-known logistic curve. Even under this propagation model, Carousel still outperforms naïve logging by a factor of almost 5. Carousel takes around 400 seconds to collect all sources while naïve logger takes 2000 seconds.

Figure 13 shows a slower worm. A slower worm can be modeled in many ways, such using a lower initial hit list, a lower scan rate, or a lower victim hitting probability. In Figure 13, we used a smaller hitting probability of 0.001. Intuitively, the faster the propagation dynamics, the better the performance of the Carousel scalable logger when compared to the naïve logger. Thus the difference is less pronounced.

Figure 14 demonstrates the scalability of Carousel, as we scale up N from $10,000$ to $100,000$ with all other parameters staying the same (i.e., 6 scans per second and $p = 0.01$). Carousel takes around 9,000 seconds to collect all sources, while the naïve logger takes 40,000 seconds. Note also that in all simulations with the logistic model (and indeed in all our experiments) the performance of the naïve logger with a Bloom filter is indistinguishable from that of the naïve logger by itself — as the theory predicts.

6.3 Non-uniform source arrivals

In this section, we study logging performance when the sources arrive at different rates as described in Section 3.1. In particular, we experiment with two equal sets of sources in which one set sends at ten times as fast as the other set. Figure 15b shows the result for the naïve logger. We observe that the naïve logger has a significant problem in logging the slow sources, which are responsible for dragging down the overall performance. As predicted by our model, the times taken to log all slow sources is ten times slower than the time taken to log all fast sources. The times to log all and almost all sources are $8,000$ and $4,000$ seconds respectively.

Simply adding a Bloom filter only slightly increases the performance of the naïve logger as predicted by the theory . On the other hand, Carousel is able to consistently log all sources as shown in Figure 15a. Carousel is not susceptible to source arrival rates: sources from both the fast and slow sets are logged equally in each minor cycle once the appropriate number of sampling bits has been determined.

6.4 Effect of Changing Hash Functions

In this section, we study the effect of randomly changing the hash functions for the Bloom filter on each major cycle (that is, each pass through all of the sets of the partition). Recall that this prevents similar arrival patterns between major cycles from causing the same source to be missed repeatedly.

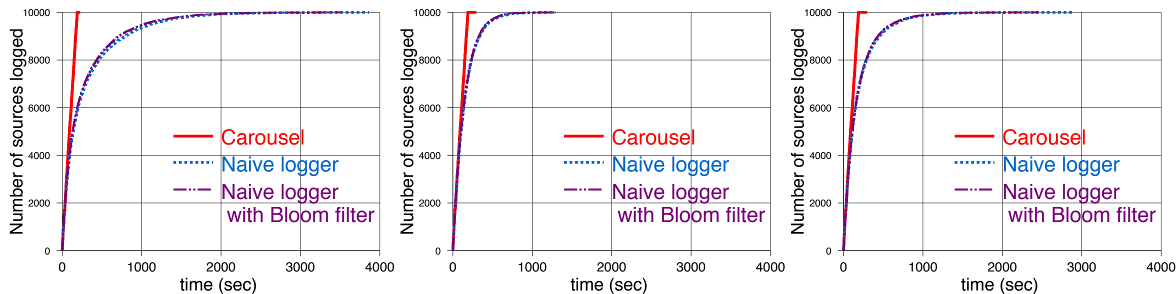


Figure 9: Performance of the Carousel scalable logger. Scan rate = 6/s, victim hit=1%, $M = 500$, $N = 10,000$, $b = 100$

Figure 10: High scan rate (60 scans/s)

Figure 11: Reduced monitoring space (50%)

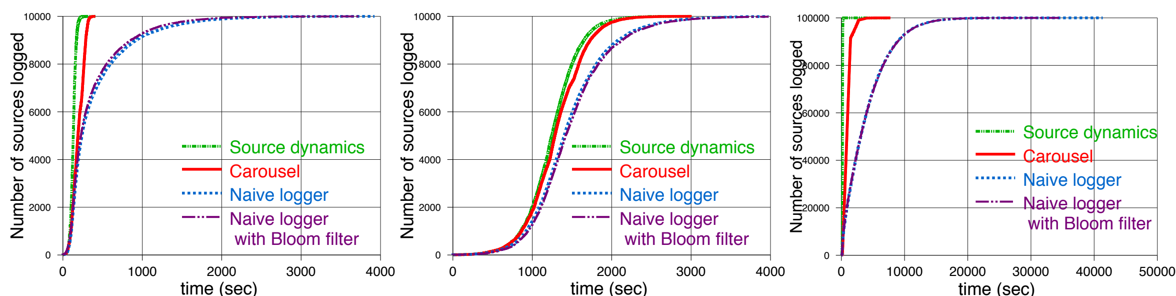


Figure 12: Logistic model of propagation - fast worm

Figure 13: Logistic model of propagation - slow worm

Figure 14: Scaling up the vulnerable population

Figure 17abc compares the performance in Carousel of using fixed hash functions throughout and changing the hash functions each major cycle with 1-bit, 5-bit and 10-bit Bloom filters respectively. We changed the hash functions randomly by simply XORing each hash value with a new random number after each major cycle. In these experiments, a major cycle is approximately 160 seconds. For the 1-bit results, one can clearly see knees in the curves at $t = 160$, 320, and 480 corresponding to each major cycle in which the logger collects sources missed in previous cycles.

Carousel instrumented with changing hash functions is much faster in collecting *all sources* across several major cycles. For example, for the 1-bit case, with changing hash functions each major cycle, it takes 1500 seconds to log all sources while using fixed hash functions takes 2500 seconds to log all sources.

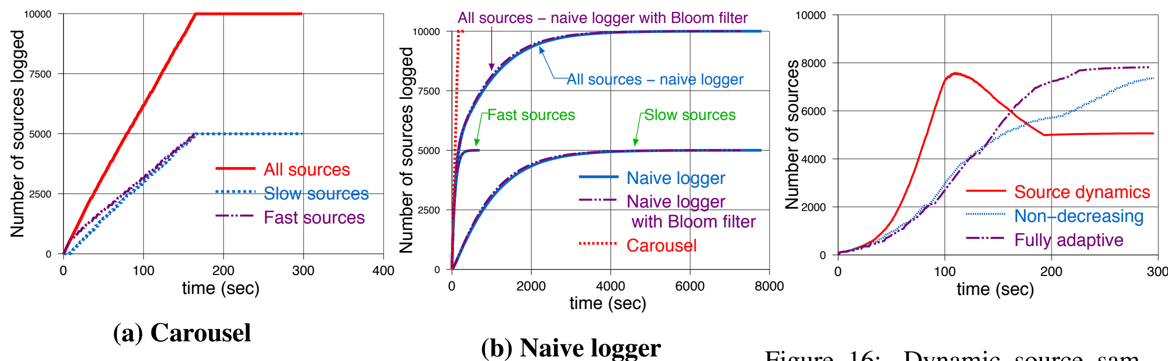
Should one prefer using a smaller number of bits per Bloom filter element and a greater number of major cycles or using a larger number of Bloom filter elements? This depends on the exact goals; for a fixed amount of memory, using a smaller number of Bloom filter bits per element allows the logger to log slightly more keys in every phase at the cost of a somewhat increased false positive probability. Based on our experiments, we believe using 5 bits per element provides excellent performance, although our Snort implementation (built before this experiment) currently uses 10 bits per element.

6.5 Adaptively Adjusting Sampling Bits

As described in Section 4.2, an optimization for Carousel is to dynamically adapt the number of sampling bits k to match the currently active source population. In a worm outbreak, the value of k needs to be large as the when the population of infected sources is large, but it should be decreased when the scope of the outbreak declines.

To study this effect, we use the *two-factor worm model* [17] to model the dynamic process of worm propagation coexisting with worm remediation. The two-factor worm model augments the standard worm model with two realistic factors: dynamic countermeasures by network administrators/users (such as node immunization and traffic firewalls) and additional congestion due to worm traffic that makes scan rates reduce when the worm grows. The model was validated using measurements of actual Internet worms (see [17]).

In Figure 16, we apply the two-factor worm model. The curve labeled "Source dynamics" records the number of infected sources as time progresses. Observe the exponential increase in the number of infected sources prior to $t = 100$.

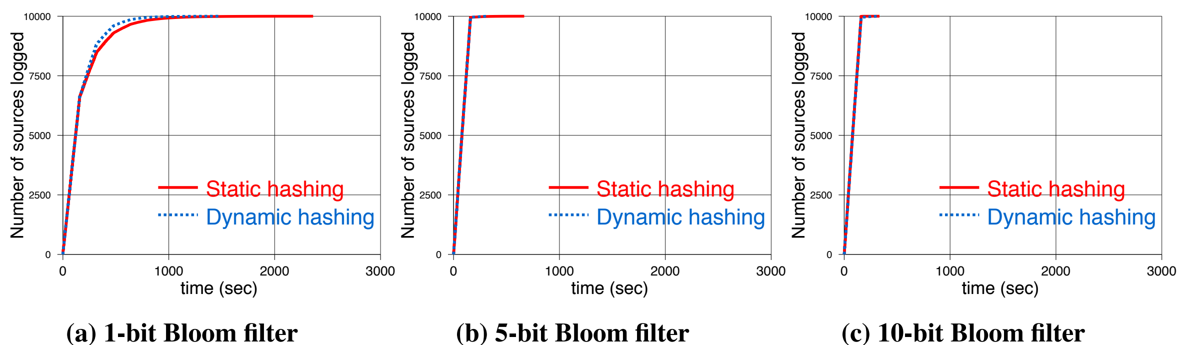


(a) Carousel

(b) Naive logger

Figure 15: Logger performance under non-uniform source arrivals

Figure 16: Dynamic source sampling in Carousel



(a) 1-bit Bloom filter

(b) 5-bit Bloom filter

(c) 10-bit Bloom filter

Figure 17: Comparison of fixed vs. changing hash functions in Carousel

However, the infected population then starts to decline.

If we let the two-factor model run to completion, the number of infected sources will eventually drop to zero, which makes logging sources less meaningful. In practice, however, it is the logging that makes remediation possible. Thus to illustrate the efficacy of using fully adaptive sampling within the logger, we only apply the two-factor model until the infectious population drops to half of the initial vulnerable tally. We then look at the time to collect the final infected population. Note that a non-decreasing logger will choose a sampling factor based on the peak population and thus may take unnecessarily long to collect the final population of infected sources.

Figure 16 shows that the fully adaptive scheme (increment k on overflow, decrement on underflow) enhances performance in terms of logging time and also the capability to collect more sources before they are immunized. In particular, the fully adaptive scheme collects almost all sources at 220 seconds while the non-decreasing scheme (only increments k on overflow, no decrements) takes more than 300 seconds to collect all sources. Examining the simulation results more closely, we found the non-decreasing scheme adapted to $k = 5$ (32 partitions) and stayed there, while the fully adaptive scheme eventually reduced to $k = 4$ (16 partitions) at time $t = 130$.

7 Snort Evaluation

We evaluate our implementation of Carousel in Snort using a testbed of two fast servers (Intel Xeon 2.8 GHz, 8 cores, 8 GB RAM) connected by a 10 Gbps link. The first server sends simulated packets to be logged according to a specified model while the second server runs Snort, with and without Carousel, to log packets.

We set the timer period $T_{phase} = 5$ seconds. The vulnerable population is $N = 10,000$ sources and the memory buffer has $M = 500$ entries. In the first experiment, the pattern of traffic arrival is random: each incoming packet is assigned a source that is uniformly and randomly picked from the population of N sources.

Figure 18 shows the logging performance of Snort instrumented with Carousel. Traffic arrives at the rate (B) of 100 Mbps. All packets have a fixed size of 1000 bytes. The logging rate is $b = 100$ events per second, i.e., $b \approx 1$ Mbps and $\frac{B}{b} = 100$. Figure 18 shows the improvements in logging from our modifications. Specifically, our scalable implementation is able to log all sources within 300 seconds

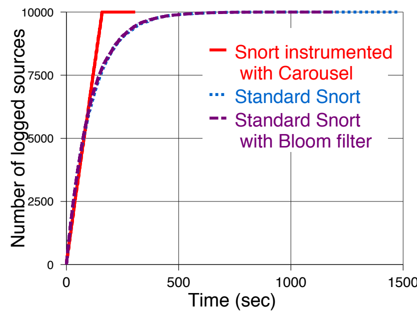


Figure 18: Logging performance of Snort instrumented with Carousel under a random traffic pattern

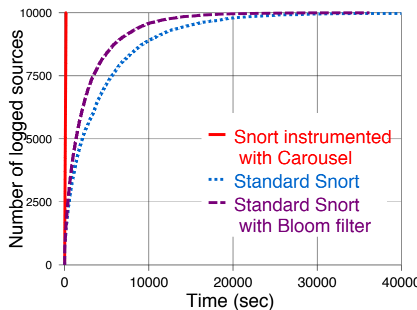


Figure 19: Logging performance of Snort instrumented with Carousel under a periodic traffic pattern

while standard Snort needs 1500 seconds. Also, adding a Bloom filter does not significantly improve the performance of Snort, matching our previous theory.

Figure 19 shows the logging performance when the sources are perpetually dispatched in a periodic pattern 1, 2, ..., N, 1, 2..., N, ... Such highly regular traffic patterns are common in a number of practical scenarios, such as synchronized attacks or periodic broadcasts of messages in the communication fabric of large distributed systems. We observe that the performance of standard Snort degrades by one order of magnitude as compared to the random pattern shown in Figure 18. Further examination shows that the naïve logger keeps missing certain sources due to the regular timing of the source arrivals. On the other hand, Carousel performance remains consistent in this setting.

We also performed an experiment with two equally sized sets of sources arriving at different rates, with fast sources arriving at 1 Gbps and slow sources at 100 Mbps, as shown in Figure 20. Our observations are consistent with the simulation results in Section 6.3. Note that in this setting standard Snort takes about 20 times longer to collect all sources than Snort with Carousel (300 seconds versus 6000 seconds); in contrast, Snort took only about 5 times longer in our experiment with random arrivals.

8 Related Work

A number of recent papers have focused on high speed implementations of IPS devices. These include papers on fast reassembly [4], fast normalization [15, 16], and fast regular expression matching (e.g., [12]). To the best of our knowledge, we have not seen prior work in network security that focuses on the problem of scalable logging. However, network managers are not just interested in detecting whether an attack has occurred but also in determining which of their computers is already infected for the purposes of remediation and forensics.

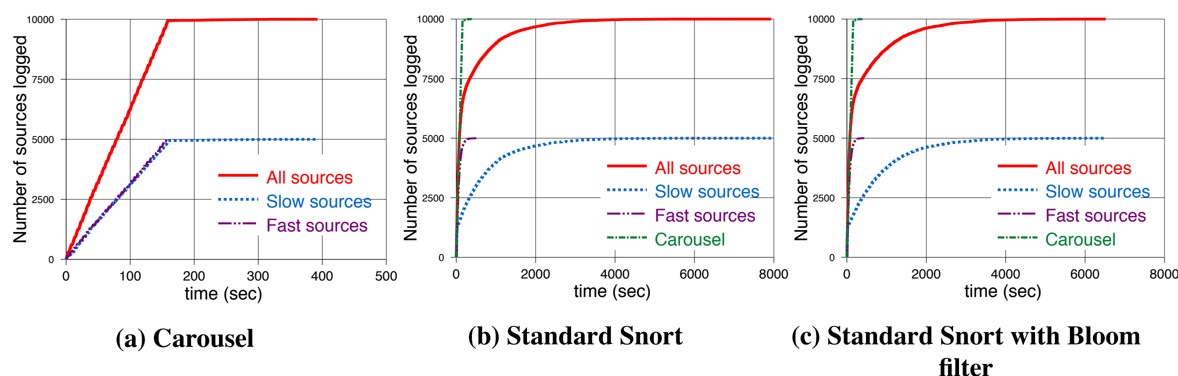
The use of random partitions, where the size is adjusted dynamically, is probably used in other contexts. We have found a reference to the Alto file system [6], where if the file system is too large to fit into memory (but is on disk), then the system resorts to a random partition strategy to rebuild the file index after a crash. Files are partitioned randomly into subsets until the subsets are small enough to fit in main memory. While the basic algorithm is similar, there are differences: we have *two* scarce resources (logging speed and memory) while the Alto algorithm only has one (memory). We have duplicates while the Alto algorithm has no duplicate files; we have an analysis, the Alto algorithm has none.

9 Conclusions

In the face of internal attacks and the need to isolate parts of an organization, IPS devices must be implementable cheaply in high speed hardware. IPS devices have successfully tackled hardware reassembly, normalization, and even Reg-Ex and behavior matching. However, when an attack is detected it is also crucial to also detect who the attacker was for potential remediation. While standard IPS devices can log source information, the slow speed of logging can result in lost information. We showed a naïve logger can take a multiplicative factor of $\ln N$ more time than needed, where N is the infected population size, for small values of memory M required for affordable hardware.

We then described the Carousel scalable logger that is easy to implement in software or hardware. Carousel collects nearly all sources, assuming they send persistently, in nearly optimal time. While large attacks such as worms and DoS attacks may be infrequent, the ability to collect a list of infected sources and bots without duplicates and loss seems like a useful addition to the repertoire of functions available to security managers.

While we have described Carousel in a security setting, the ideas applies to other monitoring tasks where the sources of all packets that match a predicate must be logged in the



(a) Carousel **(b) Standard Snort** **(c) Standard Snort with Bloom filter**

Figure 20: Snort under non-uniform source arrivals

face of high incoming speeds, low memory, and small logging speeds. The situation is akin to congestion control in networks; the classical solution, as found in say TCP or Ethernet, is for sources to reduce their rate. However, a passive logger cannot expect the sources to cooperate, especially when the sources are attackers. Thus, the Carousel scalable logger can be viewed as a form of randomized admission control where a random group of sources is admitted and logged in each phase. Another useful interpretation of our work is that while a Bloom filter of size M cannot usefully remove duplicates in a population of $N >> M$, the Carousel algorithm provides a way of recycling a small Bloom filter in a principled fashion to weed out duplicates in a very large population.

Acknowledgments

We thank the data center group of CSE Department at UCSD for the hardware setup.

Michael Mitzenmacher is supported in part by NSF grants CCF-0915922 and CNS-0721491, and from grants from Cisco Systems Inc., Yahoo! Inc., and Google Inc. Vinh The Lam and George Varghese are supported in part by grants from NSF and Cisco Systems Inc.

References

[1] Burton Bloom. Space/time trade-offs in hash coding with allowable errors. In *Communications of the ACM*, 1970.

[2] A. Borodin and R. El-Yaniv. Online computation and competitive analysis. In *Cambridge University Press*, 1998.

[3] A. Broder and M. Mitzenmacher. Network applications of Bloom filters: A survey. *Internet Math*, 1(4), 2003.

[4] S. Dharmapurikar and V. Paxson. Robust tcp stream reassembly in the presence of adversaries. In *14th USENIX Security Symposium*, pages 5–5, 2005.

[5] S. Hogg. Security at 10 Gbps: http://www.networkworld.com/community/node/39071. In *Network World*, 2009.

[6] B Lampson. Alto: A personal computer. In *Computer Structures: Principles and Examples*, 1979.

[7] F. Li and et al. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3), 2000.

[8] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[9] Arash Partow. General purpose hash functions: http://www.partow.net/programming/hashfunctions/.

[10] M. Raab and A. Steger. Balls into bins: a simple and tight analysis. In *Workshop on Randomization and Approximation Techniques in Computer Science*, 1998.

[11] A. Ross. The coupon subset collection problem. In *Journal of Applied Probability*, 2001.

[12] R. Smith and et al. Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. *SIGCOMM Comput. Commun. Rev.*, 38(4):207–218, 2008.

[13] Snort. Snort ids: http://www.snort.org.

[14] W. Stadje. The collector's problem with group drawings. In *Advances Applied Probability*, 1990.

[15] G. Varghese, J. Fingerhut, and F. Bonomi. Detecting evasion attacks at high speeds without reassembly. *SIGCOMM*, 36(4), 2006.

[16] M. Vutukuru, H. Balakrishnan, and V. Paxson. Efficient and robust tcp stream normalization. In *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 96–110, 2008.

[17] C. Zou, W. Gong, and D. Towsley. Code red worm propagation modeling and analysis. In *CCS '02*, pages 138–147. ACM, 2002.

SplitScreen: Enabling Efficient, Distributed Malware Detection

Sang Kil Cha[†], Iulian Moraru[‡], Jiyong Jang[†], John Truelove[*], David Brumley[‡], David G. Andersen[‡]
Carnegie Mellon University, Pittsburgh, PA
[†] *{sangkilc, jiyongj}@cmu.edu,* [‡] *{imoraru, dbrumley, dga}@cs.cmu.edu*
[*]*jtruelove@ll.mit.edu*

Abstract

We present the design and implementation of a novel anti-malware system called SplitScreen. SplitScreen performs an additional screening step prior to the signature matching phase found in existing approaches. The screening step filters out most non-infected files (90%) and also identifies malware signatures that are not of interest (99%). The screening step significantly improves end-to-end performance because safe files are quickly identified and are not processed further, and malware files can subsequently be scanned using only the signatures that are necessary. Our approach naturally leads to a network-based anti-malware solution in which clients only receive signatures they needed, not every malware signature ever created as with current approaches. We have implemented SplitScreen as an extension to ClamAV [13], the most popular open source anti-malware software. For the current number of signatures, our implementation is 2× faster and requires 2× less memory than the original ClamAV. These gaps widen as the number of signatures grows.

1 Introduction

The amount of malicious software (malware)—viruses, worms, Trojan horses, and the like—is exploding. As the amount of malware grows, so does the number of *signatures* used by anti-malware products (also called anti-viruses) to detect known malware. In 2008, Symantec created over 1.6 million new signatures, versus a still-boggling six hundred thousand new signatures in 2007 [2]. The ClamAV open-source anti-malware system similarly shows exponential growth in signatures, as shown in Figure 1. Unfortunately, this growth, fueled by easy-to-use malware toolkits that automatically create hundreds of unique variants [1, 20], is creating difficult system and network scaling problems for current signature-based malware defenses.

There are three scaling challenges. First, the sheer number of malware signatures that must be distributed to end-hosts is huge. For example, the ClamAV open-source product currently serves more than 120 TB of signatures per day [14]. Second, current anti-malware systems keep all signatures pinned in main memory. Reducing the size of the pinned-in-memory component is important to ensure operation on older systems and resource constrained devices such as netbooks, PDAs or smartphones, and also to reduce the impact that malware scanning has on other applications running concurrently on the same system. Third, the matching algorithms typically employed have poor cache utilization, resulting in a substantial slowdown when the signature database outgrows the L2 and L3 caches.

We propose *SplitScreen,* an anti-malware architecture designed to address the above challenges. Our design is inspired by two studies we performed. First, we found that the distribution of malware in the wild is extremely biased. For example, only 0.34% of all signatures in ClamAV were needed to detect all malware that passed through our University's e-mail gateways over a 4 month period (§5.2). Of course, for safety, we cannot simply remove the unmatched signatures since a client must be able to match anything in the signature database. Second, the performance of current approaches is bottlenecked by matching regular expression signatures in general, and by cache-misses due to that scanning in particular. Since, in existing schemes, the number of cache-misses grows rapidly with the total number of signatures, the efficiency of existing approaches will significantly degrade as the number of signatures continues to grow. Others have made similar observations [10].

At a high level, SplitScreen divides scanning into two steps. First, all files are scanned using a small, cache-optimized data structure we call a *feed-forward Bloom filter* (FFBF) [18]. The FFBF implements an approximate pattern-matching algorithm that has one-sided error: it will properly identify all malicious files, but may also identify some safe files as malicious. The FFBF outputs: (1) a set of suspect matched files, and (2) a subset of signatures from the signature database needed to confirm that suspect files are indeed malicious. SplitScreen then rescans the suspect matched files using the subset of signatures using an exact pattern matching algorithm.

The SplitScreen architecture naturally leads to a demand-driven, network-based architecture where clients download the larger exact signatures only when needed in step 2 (SplitScreen still accelerates traditional single-host scanning when running the client and the

server on the same host). For example, SplitScreen requires 55.4 MB of memory to hold the current ≈ 533,000 ClamAV signatures. ClamAV, for the same signatures, requires 116 MB of main memory. At 3 million signatures, SplitScreen can use the same amount of memory (55.4 MB), but ClamAV requires 534 MB. Given the 0.34% hit rate in our study, SplitScreen would download only 10,200 signatures for step 2 (vs. 3 million). Our end-to-end analysis shows that, overall, SplitScreen requires less than 10% of the storage space of existing schemes, with only 10% of the network volume (§5). We believe these improvements to be important for two reasons: (1) SplitScreen can be used to implement malware detection on devices with limited storage (e.g., residential gateways, mobile and embedded devices), and (2) it allows for fast signature updates, which is important when counteracting new, fast spreading malware. In addition, our architecture preserves clients' privacy better than prior network-based approaches [19].

SplitScreen addresses the memory scaling challenge because its data structures grow much more slowly than in existing approaches (with approximately 11 bytes per signature for SplitScreen compared to more than 170 bytes per signature for ClamAV). Combined with a cache-efficient algorithm, this leads to better throughput as the number of signatures grows, and represents the major advantage of our approach when compared to previous work that employed simple Bloom filters to speed-up malware detection (§5.9 presents a detailed comparison with HashAV [10]). SplitScreen addresses the signature distribution challenges because users only download the (small) subset of signatures needed for step 2. SplitScreen addresses constrained computational devices because the entire signature database need not fit in memory as with existing approaches, as well as having better throughput on lower-end processors.

Our evaluation shows that SplitScreen is an effective anti-malware architecture. In particular, we show:

- **Malware scanning at twice the speed with half the memory**: By adding a cache-efficient prescreening phase, SplitScreen improves throughput by more than 2× while simultaneously requiring less than half the total memory. These numbers will improve as the number of signatures increases.
- **Scalability**: SplitScreen can handle a very large increase in the number of malware signatures with only small decreases in performance (35% decrease in speed for 6× more signatures §5.4).
- **Distributed anti-malware**: We developed a novel distributed anti-malware system that allows clients to perform fast and memory-inexpensive scans, while keeping the network traffic very low during both normal operation and signature updates. Furthermore, clients maintain their privacy by sending only information about malware possibly present on their systems.
- **Resource-constrained devices**: SplitScreen can be applied to mobile devices (e.g., smartphones[1]), older computers, netbooks, and similar devices. We evaluated SplitScreen on a low-power device similar to an iPhone 3GS. In our experiments, SplitScreen worked properly even with 3 million signatures, while ClamAV crashed due to lack of resources at 2 million signatures.
- **Real-World Implementation**: We have implemented our approach in ClamAV, an open-source anti-malware defense system. Our implementation is available at http://security.ece.cmu.edu. We will make the malware data sets used in this paper available to other researchers upon request.

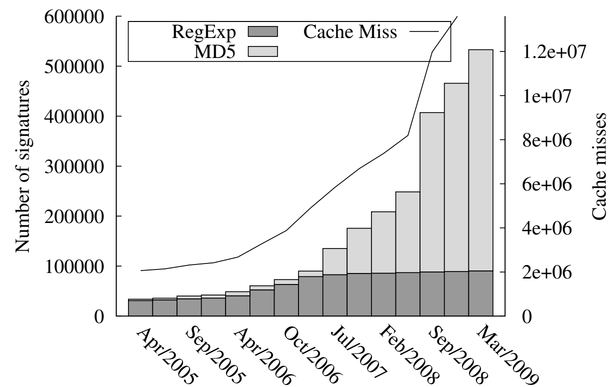


Figure 1: Number of signatures and cache misses in ClamAV from April 2005 to March 2009.

2 Background

2.1 Signature-based Virus Scanning

Signature-based anti-malware defenses are currently the most widely used solutions. While not the only approach (e.g., recent proposals for behavior-based detection such as [15]), there are two important reasons to continue improving signature-based methods. First, they remain technically viable today, and form the bedrock of the two billion dollar anti-malware industry. More fundamentally, signature-based techniques are likely to remain an important component of anti-malware defenses, even as those defenses incorporate additional mechanisms.

In the remainder of this section we describe signature-based malware scanning, using ClamAV [13] as a spe-

[1]Smartphones have many connectivity options, and are able to run an increasingly wide range of applications (sometimes on open platforms). We therefore expect that they will be subjected to the same threats as traditional computers, and they will require the same security mechanisms.

cific example. ClamAV is the most popular open-source anti-malware solution, and already incorporates significant optimizations to speed up matching and decrease memory consumption. We believe ClamAV to be representative of current malware scanning algorithms, and use it as a baseline from which to measure improvements due to our techniques.

During initialization, ClamAV reads a signed signature database from disk. The database contains two types of signatures: whole file or segment MD5 signatures and byte-pattern signatures written in a custom language with regular expression-like syntax (although they need not have wildcards) which we refer to as regular expression signatures (regexs). Figure 1 shows the distribution of MD5 and regular expression signatures in ClamAV over time. Currently 84% of all signatures are MD5 signatures, and 16% are regular expressions. In our experiments, however, 95% of the total scanning time is spent matching the regex signatures.

When scanning, ClamAV first performs several preprocessing steps (e.g., attempting to unpack and uncompress files), and then checks each input file sequentially against the signature database. It compares the MD5 of the file with MD5s in the signature database, and checks whether the file contents match any of the regular expressions in the signature database. If either check matches a known signature, the file is deemed to be malware.

ClamAV's regular expression matching engine has been significantly optimized over its lifetime. ClamAV now uses two matching algorithms [16]: Aho-Corasick [3] (AC) and Wu-Manber [23] (WM).[2] The slower AC is used for regular expression signatures that contain wildcard characters, while the faster WM handles fixed string signatures.

The AC algorithm builds a trie-like structure from the set of regular expression signatures. Matching a file with the regular expression signatures corresponds to walking nodes in the trie, where transitions between nodes are determined by details of the AC algorithm not relevant here. Successfully walking the trie from the root to a leaf node corresponds to successfully matching a signature, while an unsuccessful walk corresponds to not matching any signature. A central problem is that a trie constructed from a large number of signatures (as in our problem setting) will not fit in cache. Walks of such tries will typically visit nodes in a semi-random fashion, causing many cache misses.

The Wu-Manber [23] algorithm for multiple fixed patterns is a generalization of the single-pattern Boyer-Moore [6] algorithm. Matching using Wu-Manber entails hash table lookups, where a failed lookup means the input does not match a signature. In our setting, ClamAV uses a sliding window over the input file, where the bytes in window are matched against signatures by using a hash table lookup. Again, if the hash table does not fit in cache, each lookup can cause a cache miss. Thus, there is a higher probability of cache misses as the size of the signature database grows.

[2] ClamAV developers refer to this algorithm as extended Boyer-Moore.

2.2 Bloom Filters

The techniques we present in this paper make extensive use of Bloom filters [5]. Consider a set S. A Bloom filter is a data structure used to implement set membership tests of S quickly. Bloom filters membership tests may have one-sided errors. A false positive occurs when the outcome of the test is $x \in S$ when x is not really a member of S. Bloom filters will never incorrectly report $x \notin S$ when x really is in S.

Initialization. Bloom filter initialization takes the set S as input. A Bloom filter uses a bit array with m bits, and k hash functions to be applied to the items in S. The hashes produce integers with values between 1 and m, that are used as indices in the bit array: the k hash functions are applied to each element in S, and the bits indexed by the resulting values are set to 1 (thus, for each element in S, there will be a maximum of k bits set in the bit array—fewer if there are collisions between the hashes).

Membership test. When doing a set membership test, the tested element is hashed using the k functions. If the filter bits indexed by the resulting values are all set, the element is considered a member of the set. If at least one bit is 0, the element is definitely not part of the set.

Important parameters. The number of hash functions used and the size of the bit array determine the false positive rate of the Bloom filter. If S has $|S|$ elements, the asymptotic false positive probability of a test is $\left(1-e^{-k|S|/m}\right)^k$ [7]. For a fixed m, $k = ln2 \times |S|/m$ minimizes this probability. In practice however, k is often chosen smaller than optimum for speed considerations: a smaller k means computing a smaller number of hash functions and doing fewer accesses to the bit array. In addition, the hashing functions used affect performance, and when non-uniform, can also increase the false positive rate.

Scanning text. Text can be efficiently scanned for multiple patterns using Bloom filters in the Rabin-Karp [11] algorithm. The patterns, all of which must be of the same length w, represent the set used to initialize the Bloom filter. The text is scanned by sliding a window of length w and checking rolling hashes of its content, at every position, against the Bloom filter. Exact matching requires every Bloom filter hit to be confirmed by running a verification step to weed out Bloom filter false positives (e.g., using a subsequent exact pattern matching algorithm).

3 Design

SplitScreen is inspired by several observations. First, the number of malware programs is likely to continue to grow, and thus the scalability of an anti-malware system is a primary concern. Second, malware is not confined to high-end systems; we need solutions that protect slower systems such as smartphones, old computers, netbooks, and similar systems. Third, signature-based approaches are by far the most widely-used in practice, so improvements to signature-based algorithms are likely to be widely applicable. Finally, in current signature-based systems all users receive all signatures whether they (ultimately) need them or not, which is inefficient[3].

3.1 Design Overview

At a high level, an anti-malware defense has a set of signatures Σ and a set of files F. For concreteness, in this section we focus on regular expression signatures commonly found in anti-malware systems—so we use Σ to denote a set of regular expressions. We extend our approach to MD5 signatures in §3.5.1. The goal of the system is to determine the (possibly empty) subset $F_{malware} \subseteq F$ of files that match at least one signature $\sigma \in \Sigma$.

SplitScreen is an anti-malware system, but its approach differs from existing systems because it does not perform exact pattern matching on every file in F. Instead SplitScreen employs a cache-efficient data structure called a feed-forward Bloom filter (FFBF) [18] that we created for doing approximate pattern matching. We use it in conjunction with the Rabin-Karp text search algorithm (see §2.2). The crux of the system is that the cache-efficient first pass has extremely high throughput. The cache-efficient algorithm is approximate in the sense that the FFBF scan returns a set of suspect files $F_{suspect}$ that is a superset of malware identified by exact pattern matching, i.e., $F_{malware} \subseteq F_{suspect} \subseteq F$. In the second step we perform exact pattern matching on $F_{suspect}$ and return exactly the set $F_{malware}$. Figure 2 illustrates this strategy. The files in $F_{suspect} \setminus F_{malware}$ represent the false positives that we refer to in various sections of this paper, and they are caused by 1) Bloom filter false positives (recall that Bloom filters have one-sided error) and 2) the fact that we can only look for fixed-size fragments of signatures and not entire signatures in the first step (the FFBF scan), as a consequence of how Rabin-Karp operates.

[3] To put things in perspective, suppose there is a new Windows virus, and that the 1 billion computers with Microsoft Windows [4] are all running anti-malware software. A typical signature is at least 16 bytes (e.g., the size of an MD5). If each computer receives a copy of the signature, then that one virus has cost 15,258 MB of disk space worldwide to store the signature.

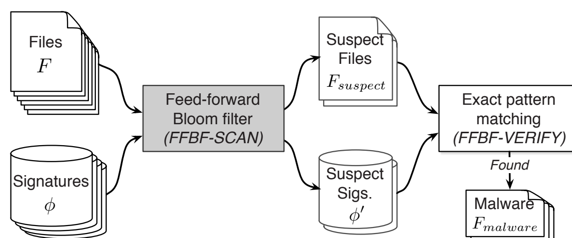


Figure 2: The SplitScreen scanning architecture.

3.2 High-Level Algorithm

The scanning algorithm used by SplitScreen consists of four processing steps called FFBF-INIT, FFBF-SCAN, FFBF-HIT, and FFBF-VERIFY, which behave as follows:

FFBF-INIT$(\Sigma) \rightarrow \phi$ takes as input the set of signatures Σ and outputs a bit-vector ϕ which we call the *all-patterns bit vector*. FFBF-SCAN will use this bit-vector to construct an FFBF to scan files.

FFBF-SCAN$(\phi, F) \rightarrow (\phi', F_{suspect})$ constructs an FFBF from ϕ and then scans each file $f \in F$ using the FFBF. The algorithm outputs the tuple $(\phi', F_{suspect})$ where $F_{suspect} \subseteq F$ is the list of files that were matched by ϕ, and ϕ' is a bit vector that identifies the signatures actually matched by $F_{suspect}$. We call ϕ' the *matched-patterns bit vector*.

FFBF-HIT$(\phi', \Sigma) \rightarrow \Sigma'$ takes in the matched-patterns bit vector ϕ' and outputs the set of regexp signatures $\Sigma' \subseteq \Sigma$ that were matched during FFBF-SCAN.

FFBF-VERIFY$(\Sigma', F_{suspect}) \rightarrow F_{malware}$ takes in a set of regular expression signatures Σ', a set of files $F_{suspect}$, and outputs the set of files $F_{malware} \subseteq F_{suspect}$ matching Σ'.

The crux of the SplitScreen algorithm can be expressed as:

$$\text{SCAN}(\Sigma, F) = \\ \text{let } (\phi', F_{suspect}) = \text{FFBF-SCAN}(\text{FFBF-INIT}(\Sigma), F) \text{ in} \\ \text{FFBF-VERIFY}(\text{FFBF-HIT}(\phi', \Sigma), F_{suspect})$$

Let R denote the existing regular expression pattern matching algorithm, e.g., R is ClamAV. SplitScreen achieves the following properties:

Correctness. SCAN will return the same set of files as identified by R, i.e., $\text{SCAN}(\Sigma, F) = R(\Sigma, F)$.

Higher Throughput. SCAN runs faster than R. In particular, we want the time for FFBF-SCAN plus FFBF-VERIFY plus FFBF-HIT to be less than the time to execute R. (Since FFBF-INIT is an initialization step performed only once per set of signatures, we do not consider it for throughput. We similarly discount in R the time to initialize any data structures in its algorithm.)

Less Memory. The amount of memory needed by SCAN is less than R. In particular, we want $\max(|\phi| + |\phi'|, |\Sigma'|) \ll |\Sigma|$ (the bit vectors are not required to be in memory during FFBF-VERIFY). We expect that the common case is that most signatures are never matched, e.g., the average user does not have hundreds of thousands or millions of unique malware programs on their computer. Thus $|\Sigma'| \ll |\Sigma|$, so the total memory overhead will be significantly smaller. In the worst case, where every signature is matched, $\Sigma' = \Sigma$ and SplitScreen's memory overhead is the same as existing systems's.

Scales to More Signatures. Since the all-patterns bit vector ϕ takes a fraction of the space needed by typical exact pattern matching data structures, the system scales to a larger number of signatures.

Network-based System. Our approach naturally leads to a distributed implementation where we keep the full set of signatures Σ on a server, and distribute ϕ to clients. Clients use ϕ to construct an FFBF and scan their files locally. After FFBF-SCAN returns, the client sends ϕ' to a server to perform FFBF-HIT, gets back the set of signatures Σ' actually needed to confirm malware is present. The client runs FFBF-VERIFY locally.

Privacy. In previous network-based approaches such as CloudAV [19], a client sends every file to a server (the cloud) for scanning. Thus, the server can see all of the client's files. In our setting, the client never sends a file across the network. Instead, the client sends ϕ', which can be thought of as a list of possible viruses on their system. We believe this is a better privacy tradeoff. Furthermore, clients can attain deniability as explained in §3.4. Note our architecture can be used to realize the existing anti-malware paradigm where the client simply asks for all signatures. Such a client would still retain the improved throughput during scanning by using our FFBF-based algorithms.

3.3 Bloom-Based Building Blocks

Bloom filters can have false positives, so a hit must be confirmed by an exact pattern matching algorithm (hence the need for FFBF-VERIFY). Our first Bloom filter enhancement reduces the number of signatures needed for verification, while the second accelerates the Bloom filter scan itself.

3.3.1 Feed-Forward Bloom Filters

An FFBF consists of two bit vectors. The *all-patterns bit vector* is a standard Bloom filter initialized as described in §3.5.1. In our setting, the set of items is Σ. The *matched-patterns bit vector* is initialized to 0.

As with an ordinary Bloom filter, a candidate item is

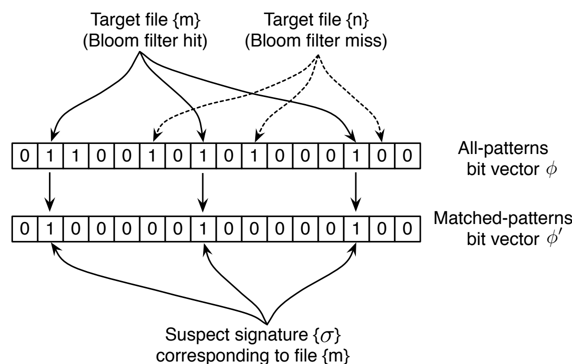


Figure 3: Building the *matched-patterns bit vector* as part of the feed-forward Bloom filter algorithm.

hashed and the corresponding bits are tested against the all-patterns bit vector. If all the hashed bits are set in the all-patterns bit vector, the item is output as a FFBF match. When a match occurs, the FFBF will additionally set each bit used to check the all-patterns bit vector to 1 in the matched-patterns bit vector. In essence, the matched-patterns bit vector records which entries were found in the Bloom filter. This process is shown in Figure 3.

After all input items have been scanned through the FFBF, the matched-patterns bit vector is a Bloom filter representing the patterns that were matched. The user of an FFBF can generate a list of potentially matching patterns by running the input pattern set against the matched-patterns Bloom filter to identify which items were actually tested. Like any other Bloom filter output, the output pattern subset may contain false positives.

In SplitScreen, ϕ is the all-patterns bit vector, and ϕ' is the matched-patterns bit vector created by FFBF-SCAN. Thus, ϕ' identifies (a superset of) signatures that would have matched using exact pattern matching. FFBF-HIT uses ϕ' to determine the set of signatures needed for FFBF-VERIFY.

3.3.2 Cache-Partitioned Bloom Filters

While a Bloom filter alone is more compact than other data structures traditionally used in pattern matching algorithms like Aho-Corasick or Wu-Manber, it is not otherwise more cache-friendly: it performs random access within a large vector. If this vector does not fit entirely in cache, the accesses will cause cache misses which will degrade performance substantially.

SplitScreen uses our cache-friendly partitioned bloom filter design [18], which splits the input bit vector into two parts. The first is sized to be entirely cache-resident, and the first s hash functions map only into this section of the vector. The second is created using virtual memory super-pages (when available) and is sized to be as *large* as possible without causing TLB misses. The FFBF pre-

vents cache pollution by using non-cached reads into the second bloom filter. The mechanisms for automatically determining the size of these partitions and the number of hash functions are described in our technical report [18].

The key to this design is that it is optimized for bloom-filter *misses*. Recall that a Bloom filter hit requires matching each hash function against a "1" in the bit vector. As a result, most misses will be detected after the first or second test, with an exponentially decreasing chance of requiring more and more tests.

The combination of a bloom-filter representation and a cache-friendly implementation provide a substantial speedup on modern architectures, as we show in §5.

3.4 SplitScreen Distributed Anti-Malware

In the SplitScreen distributed model, the input files are located on the clients, while the signatures are located on a server. The system works as follows:

1. The server generates the all-patterns bit vector for the most recent malware signatures and transmits it to the client. It will be periodically updated to contain the latest malware bit patterns, just as existing approaches must be updated.
2. The client performs the pre-screening phase using the feed-forward Bloom filter, generates the matched-patterns bit vector, compresses it and transmits it to the server.
3. The server uses the matched-patterns bit vector to filter the signatures database and sends the full definitions (1% of the signatures) to the client.
4. The client performs exact matching with the suspect files from the first phase and the suspect signatures received from the server.

In this system, SplitScreen clients maintain only the all-patterns bit vectors ϕ (there will be two bit vectors corresponding to two FFBFs, one for each type of signature). Instead of replicating the large signature database at each host, the database is stored only at the server and clients only get the signatures they are likely to need. This makes updates inexpensive: the server updates its local signature database and then sends differential all-patterns bit vector updates[4] to the clients.

Since the clients don't have to use the entire set of signatures for scanning, they also need less in-core memory (important for multi-task systems), and have smaller load times.

SplitScreen does not expose as much private data as earlier distributed anti-malware systems [19], because the contents of clients' files are never sent over the network, instead clients only send compact representations

[4] An all-patterns bit vector update is a sparse—so highly compressible—bit vector that is overlaid on top of the old bit vector.

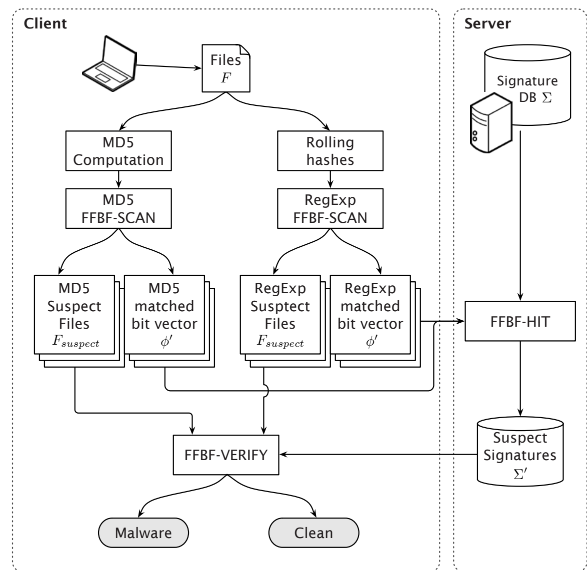


Figure 4: Data flow for distributed SplitScreen.

(bit vectors) of short hashes (under 32 bits) of small (usually under 20 bytes long) parts of undisclosed files and hashes of MD5 signatures of files. Clients concerned about deniability could set additional (randomly chosen) bits in their matched-patterns bit vectors in exchange for increased network traffic.

3.5 Design Details

3.5.1 Files and Signatures Screening

As explained in §2.1, ClamAV uses two types of signatures: regexp signatures and MD5 signatures. We handle each with its own FFBF.

Pattern signatures. The SplitScreen server extracts a fragment of length *w* from every signature (the way *w* is chosen is discussed in §5.8, while handling signatures smaller than *w* bytes and signatures containing wildcards is presented in §3.5.3 and §3.5.2). These fragments will be hashed and inserted into the FFBF. When performing FFBF scanning, a window of the same size (*w*) is slid through the examined files, and its content at every position is tested against the filter. The hash functions we use in our FFBF implementation are based on hashing by cyclic polynomials [8] which we found to be effective and relatively inexpensive. To reduce computation further, we use the idea of Kirsch and Mitzenmacher [12] and compute only two independent hash functions, deriving all the others as linear combinations of the first two.

MD5 signatures. ClamAV computes the MD5 hash of each scanned file (or its sections) and searches for it in a hash table of malware MD5 signatures. SplitScreen replaces the hash table with an FFBF to save memory. The elements inserted into the filter are the MD5 signatures

themselves, while the candidate elements tested against the filter are the MD5 hashes computed for the scanned files. Since the MD5 signatures are uniform hash values, the hash functions used for the FFBF are straightforward: given a 16-byte MD5 signature $b_1 b_2 \ldots b_{16}$, we compute the 4-byte hash values as linear combinations of $h_1 = b_1 \ldots b_4 \oplus b_5 \ldots b_8$ and $h_2 = b_9 \ldots b_{12} \oplus b_{13} \ldots b_{16}$.

3.5.2 Signatures with Wildcards

A small fraction (1.5% in ClamAV) of regular expression signatures contain wildcards, but SplitScreen's Rabin-Karp-based FFBF algorithm operates with fixed strings. Simply expanding the regular expressions does not work. For example, the expression

$$3c666f726d3e\{1-200\}3c696e707574$$

(where "{1-200}" matches any sequence no longer than 200 bytes) generates 256^{200} different byte sequences. It is impractical to put all of them into the Bloom filter.

Instead, SplitScreen extracts the invariant fragments (fixed byte subsequences) of a wildcard-containing signature and selects one of these fragments to put in the FFBF (see §3.5.4 for more details about fragment selection).

3.5.3 Short Signatures

If a regular expression signature does not contain a fixed fragment at least as long as the window size, the signature cannot be added to the feed-forward Bloom filter. Decreasing the window size to the length of the shortest signature in the database would raise the Bloom filter scan false positive rate to an unacceptable level, because the probability of a random sequence of bytes being found in any given file increases exponentially as the sequence shortens.

SplitScreen therefore performs a separate, exact pattern matching step for short signatures concurrently with the FFBF scanning. Short signatures are infrequent (they represent less than 0.4% of ClamAV's signature set for our default choice for the window size—12 bytes), so this extra step does not significantly reduce performance. The SplitScreen server builds the *short signature set* when constructing the Bloom filters. Whenever a SplitScreen client requires Bloom filter updates, the SplitScreen server sends it this short signature set too.

3.5.4 Selecting Fragments using Document Frequency

While malware signatures are highly specific, the fixed-length substrings that SplitScreen uses may not be. For example, suppose that the window size is 16 bytes. Almost every binary file contains 16 consecutive "0x00" bytes. Since we want to keep as few files as possible for the exact-matching phase, we should be careful not to include such a pattern into the Bloom filter.

Σ_i = set of signatures
σ = input signature ($\sigma \in \Sigma$)
w = fixed window size
γ = length w fixed byte sequence (w-gram) in σ
$DF(\gamma)$ = the document frequency of w-gram γ
——— *outputs* ———
ϕ_i = FFBF signatures
Σ_{short} = set of short signatures

for all $\sigma \in \Sigma_{md5}$, put σ into ϕ_{md5}
for all σ in $\Sigma_{fixed} \cup \Sigma_{wild}$
  if $|\sigma| \geq w$
    for all fixed byte w-grams γ in σ
      if $DF(\gamma) = 0$
        put γ into ϕ_{regexp}; GOTO next σ
  //either shorter than w or no zero DF
  put σ into Σ_{short}

Figure 5: Final FFBF-INIT algorithm.

We use the *document frequency (DF)* of signature fragments in clean binary files to determine if a signature fragment is likely to match safe files. The *DF* of a signature fragment represents the number of documents containing this fragment. A high DF indicates that the corresponding signature fragment is common and may generate many false positives.

We compute the DF value for each window-sized signature fragment in clean binary samples. For each signature, we insert into the filter the first fragment with a DF value of zero (i.e., the fragment did not occur in any of the clean binary files). The signatures that have no zero DF fragments are added to the short signature set.

We summarize our signature processing algorithm in Figure 5. The SplitScreen server runs this algorithm for every signature, and creates two Bloom filters—one for MD5 signatures, and one for the regular expression signatures—as well as the set of short signatures.

3.5.5 Important Parameters

We summarize in this section the important parameters that affect the performance of our system, focusing on the tradeoffs involved in choosing those parameters.

Bit vector size. The size of the bit vectors trades scan speed for memory use. Larger bit vectors (specifically, larger non-cache-resident parts) result in fewer Bloom filter false positives, improving performance up to the point where TLB misses become a problem (see §3.3.2).

Sliding window size. The wider the sliding window used to scan files during FFBF-SCAN, the less chance there is of a false positive (see §5.8). This makes FFBF-VERIFY run faster (because there will be fewer files to check). However, the wider the sliding window, the more signatures that must be added to the short signature set. Since we look for short signatures in every input file,

a large number of short signatures will reduce performance.

Number of Bloom filter hash functions. The number of hash functions used in the FFBF algorithm (the k parameter in §2.2) is a parameter for which an optimum value can be computed when taking into account the characteristics of the targeted hardware (e.g. the size of the caches, the latencies in accessing different levels of the memory hierarchy) as described in [18]. Empirically, we found that two hash functions each for the cache-resident part and the non-cache-resident part of the FFBF works well for a wide range of hardware systems.

4 Implementation

We have implemented SplitScreen as an extension of the ClamAV open source anti-malware platform, version 0.94.2. Our code is available at `http://security.ece.cmu.edu`. The changes comprised approximately 8K lines of C code. The server application used in our distributed anti-malware system required 5K lines of code. SplitScreen servers and SplitScreen clients communicate with each other via TCP network sockets.

The SplitScreen client works like a typical anti-malware scanner; it takes in a set of files, a signature database (ϕ in SplitScreen), and outputs which files are malware along with any additional metadata such as the malware name. We modified the existing `libclamav` library to have a two-phase scanning process using FF-BFs.

The SplitScreen server generates ϕ from the default ClamAV signatures using the algorithm shown in Figure 5. Note that SplitScreen can implement traditional single-host anti-malware by simply running the client and server on the same host. We use run-length encoding to compress the bit vectors and signatures sent between client and server.

5 Evaluation

In this section we first detail our experimental setup, and then briefly summarize the malware measurements that confirm our hypothesis that most of the volume of malware can be detected using a few signatures. We then present an overall performance comparison of SplitScreen and ClamAV, followed by detailed measurements to understand why SplitScreen performs well, how it scales with increasing numbers of regexp and MD5 signatures, and how its memory use compares with ClamAV. We then evaluate SplitScreen's performance on resource constrained devices and its performance in a network-based use model.

5.1 Evaluation Setup

Unless otherwise specified, our experiments were conducted on an Intel 2.4 GHz Core 2 Quad with 4 GB of RAM and a 8 MB split L2 cache using a 12-byte window size (see §3). When comparing SplitScreen against ClamAV, we exclude data structure initialization time in ClamAV, but count the time for FFBF_INIT in SplitScreen. Thus, our measurements are conservative because they reflect the best possible setting for ClamAV, and the worst possible setting for SplitScreen. Unless otherwise specified, we report the average over 10 runs.

Scanned files. Unless otherwise specified, all measurements reflect scanning 344 MB of 100% clean files. We use clean files because they are the common case, and exercise most code branches. (§5.7 shows performance for varying amounts of malware.) The clean files come from a fresh install of Microsoft Windows XP plus typical utilities such as MS Office 2007 and MS Visual Studio 2007.

Signature sets. We use two sets of signatures for the evaluation. If unspecified, we focus on the current ClamAV signature set (main v.50 and daily v.9154 from March 2009), which contained 530K signatures. We use four additional historical snapshots from the ClamAV source code repository. To measure how SplitScreen will improve as the number of signatures continues to grow, we generated additional regex and MD5 signatures ("projected" in our graphs) in the same relative proportion as the March signature set. The synthetic regexs were generated by randomly permuting fixed strings in the March snapshot, while the synthetic MD5s are random 16-byte strings.

5.2 Malware Measurements

Given a set of signatures Σ, we are interested in knowing how many individual signatures Σ' are matched in typical scenarios, i.e., $|\Sigma'|$ vs. $|\Sigma|$. We hypothesized that most signatures are rarely matched ($|\Sigma'| \ll |\Sigma|$), e.g., most signatures correspond to malware variants that are never widely distribution.

One typical use of anti-malware products is to filter out malware from email. We scanned Carnegie Mellon University's email service from May 1st to August 29th of 2009 with ClamAV. 1,392,786 malware instances were detected out of 19,443,381 total emails, thus about 7% of all email contained malware by volume. The total number of unique signatures matched was 1,825, which is about 0.34% of the total signatures—see figure 6.

Another typical use of anti-malware products is to scan files on disk. We acquired 393 GB of malware from various sites, removed duplicate files based upon MD5, and removed files not recognized by ClamAV using the v.9661 daily and v.51 main signature database. The total number of signatures in ClamAV was 607,988, and the

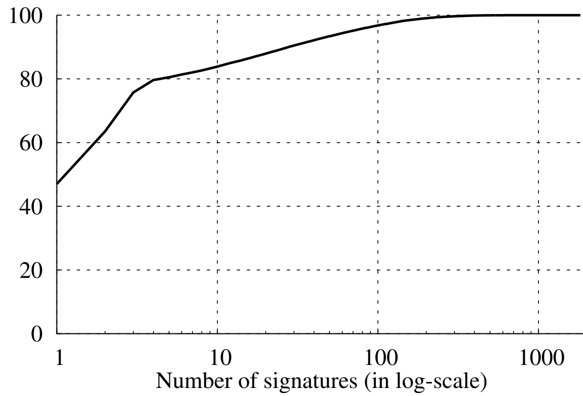


Figure 6: The overall amount of malware detected (y axis) vs. the total number of malware signatures needed (x axis). For example, about 1000 signatures are needed to detect virtually all malware.

total number of unique malware files was 960,766 (about 221 GB). ClamAV reported out of the 960,766 unique files that there were 128,992 unique malware variants. Thus, about 21.2% of signatures were matched.

We conclude that indeed most signatures correspond to rare malware, while only a few signatures are typically needed to match malware found in day-to-day operations.

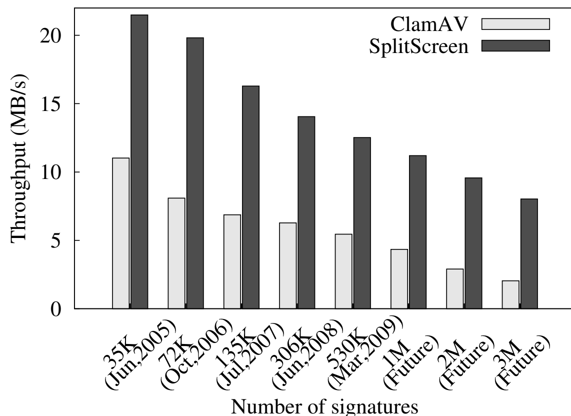


Figure 7: Performance of SplitScreen and ClamAV using historical and projected ClamAV signature sets.

5.3 SplitScreen Throughput

We ran SplitScreen using both historical and projected signature sets from ClamAV, and compared its performance to ClamAV on the same signature set. Figure 7 shows our results. SplitScreen consistently improves throughput by at least 2× on previous and existing signatures, and the throughput improvement factor increases with the number of signatures.

Understanding throughput: Cache misses. We hypothesized that a primary bottleneck in ClamAV was L2 cache misses in regular expression matching. Figure 8 shows ClamAV's throughput and memory use as the number of regular expression signatures grows from zero to roughly 125,000, with *no* MD5 signatures. In contrast, increasing the number of MD5 signatures linearly increases the total memory required by ClamAV, but has almost no effect on its throughput. With no regexp signatures, ClamAV scanned nearly 50 MB/sec, regardless of the number of MD5 signatures.

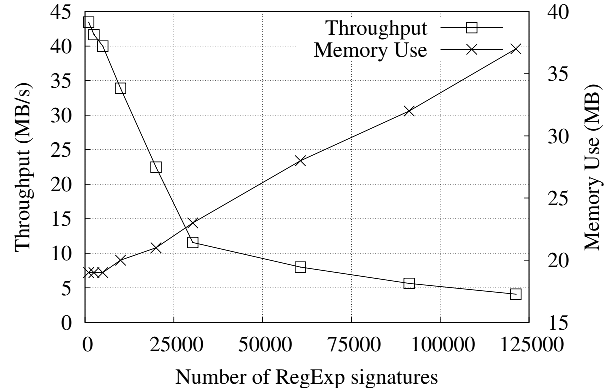


Figure 8: ClamAV scanning throughput and memory consumption as the number of regular expression signatures increases.

Figure 9 compares the absolute number of L2 cache misses for ClamAV and SplitScreen as the (total) number of signatures increases. The dramatic increase in L2 cache misses for ClamAV suggest that this is, indeed, a major source of its performance degradation. In contrast, the number of cache misses for SplitScreen is much lower, helping to explain its improved scanning performance. These results indicate that increasing the number of regex signatures increases the number of cache misses, decreases throughput, and thus is the primary throughput bottleneck in ClamAV.

5.4 SplitScreen Scalability and Performance Breakdown

How well does SplitScreen scale? We measured three scaling dimensions: 1) how throughput is affected as the number of regular expression signatures grows, 2) how FFBF size affects performance and memory use, and 3) where SplitScreen spends time as the number of signatures increases.

Throughput. Figure 10 shows SplitScreen's throughput as the number of signatures grows from 500K (approximately what is in ClamAV now) to 3 million. At 500K signatures, SplitScreen performs about 2.25 times better than ClamAV. **At 3 million signatures, SplitScreen performs 4.5 times better**. The 4.5×

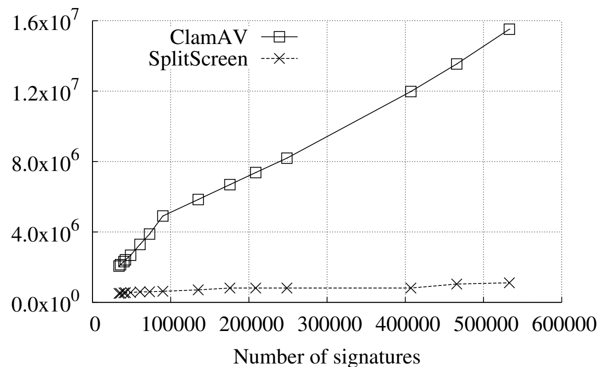


Figure 9: Cache misses.

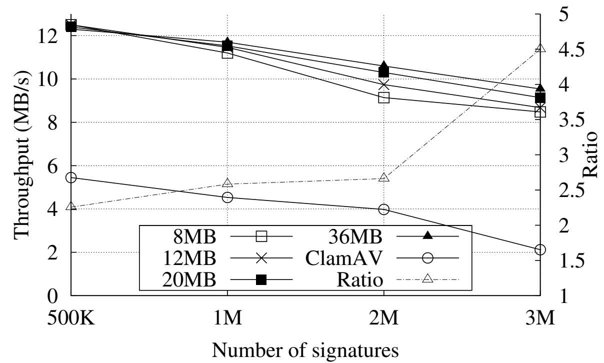


Figure 10: Performance for different size feed-forward Bloom filters, keeping the cache-resident portion constant.

throughput increase is given with a 32 MB FFBF. These measurements are all an average over 10 runs. The worst of these runs is the first when the file system cache is cold, when SplitScreen was only 3× faster than ClamAV (graph omitted due to space).

FFBF Size. We also experimented with smaller FFBF's of size 8, 12, 20, and 36 MB, as shown in Figure 10. The larger the FFBF, the smaller the false positive ratio, thus the greater the performance. We saw no additional performance gain by increasing the FFBF beyond 36 MB.

# sigs	FFBF-SCAN +Short Sigs.	FFBF-HIT + Traffic	FFBF-VERIFY
500K	27.2 (94.7%)	0.7 (2.6%)	0.8 (2.7%)
1M	27.4 (92.4%)	0.9 (3.0%)	1.4 (4.6%)
2M	26.5 (76.0%)	1.3 (3.7%)	7.1 (20.3%)
3M	24.2 (58.3%)	1.7 (4.1%)	15.6 (37.6%)

Table 1: Time spent per step by SplitScreen to scan 1.55 GB of files (in seconds and by percentage).

Per-Step Breakdown. Table 1 shows the breakdown of time spent per phase. We do not show FFBF-INIT which was always $< 0.01\%$ of total time. As noted earlier, we omit ClamAV initialization time in order to provide conservative comparisons.

We make draw several conclusions from our experiments. First, SplitScreen's performance advantage continues to grow as the number of regexp signatures increases. Second, the time required by the first phase of scanning in SplitScreen holds steady, but the exact matching phase begins to take more and more time. This occurs because we held the size of the FFBF constant. When we pack more signatures into the same size FFBF, the bit vector becomes more densely populated, thus increasing the probability of a false positive due to hash collisions. Such false positives result in more signatures to check during FFBF-VERIFY. Thus, while the overall scan time is relatively small, increasing the SplitScreen FFBF size will help in the future, i.e., we can take advantage of the larger caches the future may bring. Note that the size increases to the FFBF need be nowhere near as large as with ClamAV, e.g., a few megabytes for SplitScreen vs. a few hundred megabytes for ClamAV.

5.5 SplitScreen on Constrained Devices

Figure 11 compares the memory required by SplitScreen and ClamAV for FFBF-SCAN. 533,183 signatures in ClamAV consumed about 116 MB of memory. SplitScreen requires only 55.4 MB, of which 40 MB are dedicated to FFBFs. Our FFBF was designed to minimize false positives due to hash collisions but not adversely affect performance due to TLB misses (§3.3.2). At 3 million signatures, ClamAV consumed over 500 MB of memory, while SplitScreen still performed well with a 40 MB FFBF.

We then tested SplitScreen's performance with four increasingly more limited systems. We compare SplitScreen and ClamAV using the current signature set on: a 2009 desktop computer (Intel 2.4 GHz Core 2 Quad, 4 GB RAM, 8 MB L2 cache); a 2008 Apple laptop (Intel 2.4 GHz Core 2 Duo, 2 GB RAM, 3 MB L2 cache); a 2005 desktop (Intel Pentium D 2.8 GHz, 4 GB RAM, 2 MB L2 Cache); and a Alix3c2 (AMD Geode 500 Mhz, 256 MB RAM, 128 KB L2 Cache) that we use as a proxy for mobile/handheld devices.[5]

Figure 12 shows these results. On the desktop systems and laptop, SplitScreen performs roughly 2x better than ClamAV. On the embedded system, SplitScreen performs 30% better than the baseline ClamAV. The modest performance gain was a result of the very small L2 cache on the embedded system.

However, our experiments indicate a more fundamental limitation with ClamAV on the memory-constrained AMD Geode. When we ran using the 2 million signature dataset, ClamAV exhausted the available system memory and crashed. In contrast, SplitScreen successfully oper-

[5]The AMD Geode has hardware capabilities similar to the iPhone 3GS, which has a 600 MHz ARM processor with 128 MB of RAM.

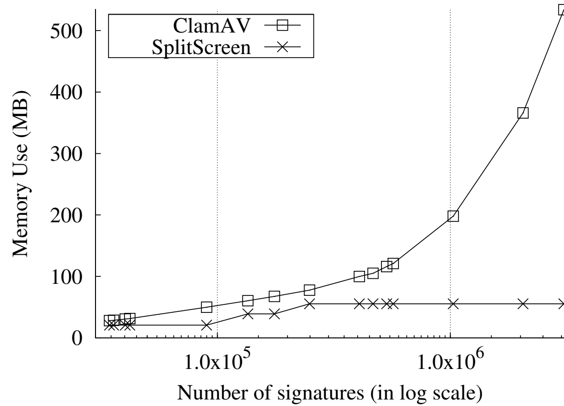


Figure 11: Memory use of SplitScreen and ClamAV.

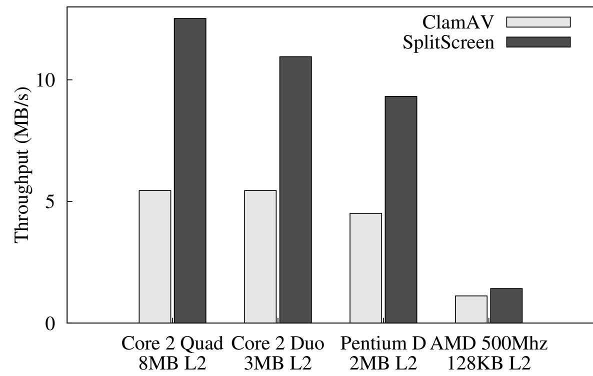


Figure 12: Performance for four different systems (differing CPU, cache, and memory size).

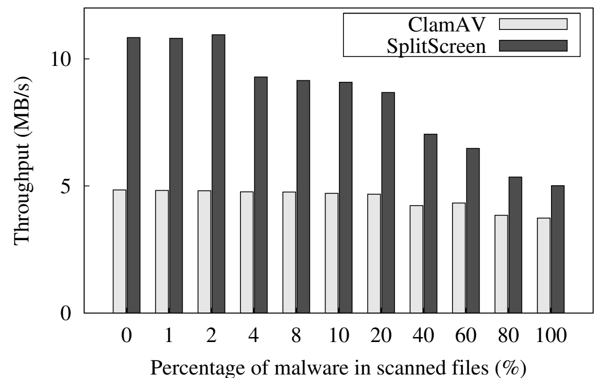


Figure 13: Throughput as % of malware increases (using total scan time including verification).

ated using even the 3 million signature dataset. These results suggest that SplitScreen is a more effective architecture for memory-constrained devices.

5.6 SplitScreen Network Performance

In the network-based setting there are three data transfers between server and client: 1) the initial bit vector ϕ (the all-patterns bit vector) generated by FFBF-INIT sent from the server to the client; 2) the bit vector ϕ' (the matched-patterns bit vector) for signatures matched by FFBF-SCAN sent by the client to the server; and 3) the set of signatures Σ' needed for FFBF-VERIFY sent by the server to the client.

Recall that SplitScreen compresses the (likely-sparse) bit vectors before transmission. The compressed size of ϕ' depends upon the signatures matched and the FFBF false positive rate. Table 2 shows the network traffic and false-positive rates in different cases. The size of both ϕ' and Σ' remains small for these files, requiring significantly less network traffic than transferring the entire signature set.

Table 3 shows the size of the all-patterns bit vector ϕ, which must be transmitted periodically to clients, for increasing (gzipped) ClamAV database sizes. SplitScreen requires about 10% the network bandwidth to distribute the initial signatures to clients.

Overall, the volume of network traffic for SplitScreen ($|\phi| + |\phi'| + |\Sigma'|$) is between 10%-13% of that used by ClamAV on a fresh scan. On subsequent scans SplitScreen will go out and fetch new ϕ' and Σ' if new signatures are matched (e.g., the ϕ' of a new scan has different bits set than previous scans). However, since $|\Sigma'| \ll |\Sigma|$, the total lifetime traffic is still expected to be very small.

5.7 Malware Scanning

How does the amount of malware affect scan throughput? We created a 100 MB corpus using different ratios of malware and clean PE files. Figure 13 shows that SplitScreen's performance advantage slowly decreases as the percentage of malware increases, because it must re-scan a larger amount of the input files using the exact signatures.

5.8 Additional SplitScreen Parameters

In addition to the FFBF size (§5.4), we measured the effect of different hash window sizes and the effectiveness of using document frequency to select good tokens for regular expression signatures.

Fixed string selection and document frequency. The better the fixed string selection, the lower the false positive rate will be, and thus the better SplitScreen performs. We use the document frequency (DF) of known good programs to eliminate fixed strings that would cause false positives. Our experiments were conducted using the known clean binaries as described in §5.1. We found the performance increase in Figure 13 was in part due to DF removing substrings that match clean files. We did a subsequent test with 344 MB of PE files from our data set. Without document frequency, we had a 22% false pos-

Target File Types	Size of target files	Number of target files	$|\phi'|$ (Bytes)	$|\Sigma'|$ (Bytes)	Total traffic (Bytes)	False-positive rate
Randomly generated	200 MB	1,000	80	405	485	0.50%
Randomly generated	2 GB	10,000	224	223	447	0.14%
Clean PE files	340 MB	1,957	1,829	15,082	16,911	4.19%
Clean ELF files	157 MB	1,319	180	11,766	13,338	9.26%
100% Malware	170 MB	534	17,100	160,828	177,928	N/A
100% Malware	1.1 GB	5277	61,748	648,962	710,710	N/A

Table 2: Network traffic for SplitScreen using 530K signatures.

# signatures	ClamAV CVD (MB)	FFBF + Short Sigs (MB)
130K	9.9	0.77
245K	13.5	1.2
530K	20.8	2.0

Table 3: Signature size initially sent to clients.

Window Size	Avg. F-P	Max. F-P	# Short Sigs
8 bytes	17.3	18.9	1169
10 bytes	11.6	14.3	1350
12 bytes	8.56	9.36	1624
14 bytes	6.70	7.77	2004
16 bytes	5.23	6.31	3203

Table 4: False positive rates for different window sizes. The average and maximum FP rates are from the 10-fold cross validation of DF on 1.55 GB of clean binaries.

itive rate and a throughput of 10 MB/s. With document frequency, we had a 0.9% false positive rate and 12 MB/s throughput. We also performed 10-fold cross validation to confirm that document frequency is beneficial, with the average and max false positive rate per window size shown in Table 4.

Window size. A shorter hash window results in fewer short regexp signatures, but increases the false positive rate. The window represents the number of bytes from each signature used for FFBF scanning. For example, a window of 1 byte would mean a file would only have to match 1 byte of a signature during FFBF-SCAN. (The system ensures correctness via FFBF-VERIFY.)

Using an eight-byte window, hash collisions caused a 3.98% of files to be mis-identified as malware in FFBF-SCAN that later had to be weeded out during FFBF-VERIFY. With a sixteen-byte window, the false positive rate was only 0.46%. The throughput for an 8 and 16 byte window was 9.44 MB/s and 8.67 MB/s, respectively. Our results indicate a window size of 12 seems optimal as a balance between the short signature set size, the false positive rate, and the scan rate.

5.9 Comparison with HashAV

The work most closely related to ours is HashAV [10]. HashAV uses Bloom filters as a first pass to reduce the number of files scanned by the regular expression algorithms. Although there are many significant differences between SplitScreen and HashAV (see §7), HashAV serves as a good reference for the difference between a typical Bloom scan and our FFBF-based techniques.

To enable a direct comparison, we made several modifications to each system. We modified SplitScreen to ignore file types and perform only the raw scanning supported by HashAV. We disabled MD5 signature computation and scanning in SplitScreen to match HashAV's behavior. We updated HashAV to scan multiple files instead of only one. Finally, we changed the evaluation to include only the file types that HashAV supported. *It is important to note that the numbers in this section are not directly comparable to those in previous sections.* HashAV did not support the complex regexp patterns that most frequently show up in SplitScreen's small signatures set, so the performance improvement of SplitScreen over ClamAV appears larger in this evaluation that it does in previous sections.

Figure 14 shows that with 100K signatures, SplitScreen performs about 9× better than HashAV, which in turn outperforms ClamAV by a factor of two. SplitScreen's performance does not degrade with an increasing number of signatures, while HashAV's performance does. One reason is SplitScreen is more cache friendly; with large signature sets HashAV's default Bloom filter does not fit in cache, and the resulting cache misses significantly degrade performance. If HashAV decreased the size of their filter, then there would be many false positives due to hash collisions. Further, HashAV does not perform verification using the small signature set as done by SplitScreen. As a result, the data structure for exact pattern matching during HashAV verification will be much larger than during verification with SplitScreen.

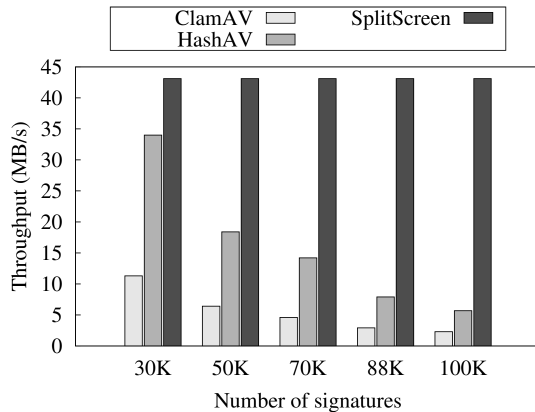


Figure 14: HashAV and SplitScreen scan throughput.

6 Discussion

We see the SplitScreen distributed model providing benefits in several scenarios, beyond the basic speedup provided by our approach. As shown in §5.6, a SplitScreen client requires 10× less data than a ClamAV client before it can start detecting malware. Furthermore, sending a new signature takes 8 bytes for SplitScreen (remember from §3.5.1 that all the FFBF bits corresponding to a signature are generated from just two independent 32-bit hashes) and 20 to 350 bytes on ClamAV. These factors make SplitScreen more effective in responding to new malware because there is less pressure on update servers, and clients get updates faster. The other advantage to dynamically downloading signatures is that SplitScreen can be installed on devices with limited storage space, like residential gateways or mobile devices.

In the SplitScreen distributed anti-malware model, the server plays an active role in the scanning process: it extracts relevant signatures from the signature database for every scan that generates suspect files on a client. Running on an Intel 2.4 GHz Core 2 Quad machine, the unoptimized server can sustain up to 14 requests per second (note that every request corresponded to a scan of 1.5 GB of binary files, so the numbers of suspect files and signatures were relatively high). As such, a single server can handle the virus scanning load of a set of clients scanning 21 GB/sec of data. While this suffices for a proof-of-concept, we believe there is substantial room to optimize the server's performance in future work: (1) Clients can cache signatures from the server by adding them to their short signatures set; (2) the server can use an indexing mechanism to more rapidly retrieve the necessary signatures based upon the bits set in the matched-patterns bit vector; (3) conventional or, perhaps, peer-to-peer replication techniques can be easily used to replicate the server, whose current implementation is CPU intensive but does not require particularly large amounts of disk or memory. These improvements are complementary to our core problem of efficient malware scanning, and we leave them as future work.

7 Related Work

CloudAV [19] applies cloud computing to anti-virus scanning. It exploits 'N-version protection' to detect malware in the cloud network with higher accuracy. Its scope is limited, however, to controlled environments such as enterprises and schools to avoid dealing with privacy. Each client in CloudAV sends files to a central server for analysis, while in SplitScreen, clients send only their matched-patterns bit vector.

Pattern matching, including using Bloom filters, has been extensively studied in and outside of the malware detection context. Several efforts have targeted network intrusion detection systems such as Snort, which must operate at extremely high speed, but that have a smaller and simpler signature set [21]. Bloom filters are a commonly-proposed technique for hardware accelerated deep packet inspection [9].

HashAV proposed using Bloom filters to speed up the Wu-Manber implementation used in ClamAV [10]. They show the importance of taking into account the CPU caches when designing exact pattern matching algorithms. However, their system does not address all aspects of an anti-malware solution, including MD5 signatures, signatures shorter than the window size, cache-friendly Bloom filters when the data size exceeds cache size, and reducing the number of signatures in the subsequent verification step. Furthermore, the SplitScreen FFBF-based approach scales much better for increases in the number of signatures.

A solution for signature-based malware detection in resource constrained mobile devices had previously been presented in [22]. Similarly to SplitScreen, it used signature fragment selection to accelerate the scanning, but could only handle fixed byte signatures, and was less memory efficient than SplitScreen.

The "Oyster" ClamAV extensions [17] replaced ClamAV's Aho-Corasick trie with a multi-level trie to improve its scalability, improving throughput, but did not change its fundamental cache performance or reduce the number of signatures that files must be scanned against.

8 Conclusion

SplitScreen's two-phase scanning enables fast and memory-efficient malware detection that can be decomposed into a client/server process that reduces the amount of storage on, and communication to, clients by an order of magnitude. The key aspects that make this design work are the observation that most malware signatures are never matched—but must still be detectable—combined with the feed-forward Bloom filter that re-

duces the problem of malware detection to scanning a much smaller set of files against a much smaller set of signatures. Our evaluation of SplitScreen, implemented as an extension of ClamAV, shows that it improves scanning throughput using today's signature sets by over 2×, using half the memory. The speedup and memory savings of SplitScreen improve further as the number of signatures increases. Finally, the efficient distributed execution made possible using SplitScreen holds the potential to enable scalable malware detection on a wide range of low-end consumer and handheld devices.

Acknowledgements

We would like to thank Pei Cao and Ozgun Erdogan for helpful discussions and feedback, as well as for making the source code to HashAV available. We would also like to thank Siddarth Adukia, the anonymous reviews and our shepherd for their helpful comments. This work was supported in part by gifts from Network Appliance, Google, and Intel Corporation, by grant CNS-0619525 from the National Science Foundation, and by CyLab at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office. The views expressed herein are those of the authors and do not necessarily represent the views of our sponsors.

References

[1] F-secure: Silent growth of malware accelerates. http://www.f-secure.com/en_EMEA/security/security-lab/latest-threats/security-threat-summaries/2008-2.html.

[2] Symantec global internet security threat report. http://www.symantec.com/about/news/release/article.jsp?prid=20090413_01.

[3] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Comm. of the ACM*, 18: 333–340, 1975.

[4] S. Ballmer. http://www.microsoft.com/msft/speech/FY07/BallmerFAM2007.mspx, 2007.

[5] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Comm. of the ACM*, 13:422–426, 1970.

[6] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Comm. of the ACM*, 20:762–772, 1977.

[7] A. Broder and M. Mitzenmacher. Network Applications of Bloom Filters: A Survey. In *Internet Mathematics*, pages 636–646, 2002.

[8] J. D. Cohen. Recursive hashing functions for n-grams. *ACM Transactions on Information Systems*, 15(3):291–320, 1997.

[9] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep Packet Inspection Using Parallel Bloom Filters. *IEEE Micro*, 24:52–61, January 2004.

[10] O. Erdogan and P. Cao. Hash-AV: fast virus signature scanning by cache-resident filters. *International Journal of Security and Networks*, 2:50, 2007.

[11] R. M. Karp and M. O. Rabin. Efficient Randomized Pattern-Matching Algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[12] A. Kirsch and M. Mitzenmacher. Less hashing, same performance: Building a better Bloom filter. *Random Structures & Algorithms*, 33(2):187–218, 2008.

[13] T. Kojm. Clamav. URL http://www.clamav.net.

[14] T. Kojm. Introduction to ClamAV. http://www.clamav.net/doc/webinars/Webinar-TK-2008-06-11.pdf, 2008.

[15] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *Proc. of the 18th USENIX Security Symposium*, 2009.

[16] P.-c. Lin, Z.-x. Li, Y.-d. Lin, Y.-c. Lai, and F. Lin. Profiling and accelerating string matching algorithms in three network content security applications. *IEEE Comm. Surveys & Tutorials*, 8:24–37, April 2006.

[17] Y. Miretskiy, A. Das, C. P. Wright, and E. Zadok. Avfs: An on-access anti-virus file system. In *Proc. of the 13th USENIX Security Symposium*, 2004.

[18] I. Moraru and D. G. Andersen. Fast Cache for Your Text: Accelerating Exact Pattern Matching with Feed-Forward Bloom Filters. Technical Report CMU-CS-09-159, Carnegie Mellon University, 2009.

[19] J. Oberheide, E. Cooke, and F. Jahanian. CloudAV: N-Version Antivirus in the Network Cloud. In *Proc. of the 17th USENIX Security Symposium*, 2008.

[20] G. Ollmann. The evolution of commercial malware development kits and colour-by-numbers custom malware. *Computer Fraud & Security*, 2008(9):4 – 7, 2008.

[21] H. Song, T. Sproull, M. Attig, and J. Lockwood. Snort offloader: a reconfigurable hardware NIDS filter. *International Conference on Field Programmable Logic and Applications, 2005.*, pages 493–498, 2005.

[22] D. Venugopal and G. Hu. Efficient signature based malware detection on mobile devices. *Mobile Information Systems*, 4(1):33–49, 2008.

[23] S. Wu and U. Manber. A fast algorithm for multi-pattern searching. Technical Report TR-94-17, University of Arizona, 1994.

Behavioral Clustering of HTTP-Based Malware and Signature Generation Using Malicious Network Traces

Roberto Perdisci[a,b], *Wenke Lee*[a], *and Nick Feamster*[a]
[a]*College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA*
[b]*Damballa, Inc. Atlanta, GA 30308, USA*
perdisci@gtisc.gatech.edu, {wenke, feamster}@cc.gatech.edu

Abstract

We present a novel *network-level* behavioral malware clustering system. We focus on analyzing the structural similarities among malicious HTTP traffic traces generated by executing HTTP-based malware. Our work is motivated by the need to provide quality input to algorithms that automatically generate network signatures. Accordingly, we define similarity metrics among HTTP traces and develop our system so that the resulting clusters can yield high-quality malware signatures.

We implemented a proof-of-concept version of our network-level malware clustering system and performed experiments with more than 25,000 distinct malware samples. Results from our evaluation, which includes real-world deployment, confirm the effectiveness of the proposed clustering system and show that our approach can aid the process of automatically extracting network signatures for detecting HTTP traffic generated by malware-compromised machines.

1 Introduction

The battle against malicious software (a.k.a. *malware*) is becoming more difficult. Today's malware writers commonly use executable packing [16] and other code obfuscation techniques to generate a large number of polymorphic variants of the same malware. As a consequence, anti-viruses (AVs) have a hard time keeping their signature database up-to-date, and their AV scanners often have many false negatives [26].

Although it is easy to create many polymorphic variants of a given malware sample, different variants of the same malware will exhibit similar malicious activities, when executed. Behavioral malware clustering groups malware variants according to similarities in their malicious behavior. This process is particularly useful because once a number of different variants of the same malware have been identified and grouped together, it is easier to write a generic *behavioral signature* that can be used to detect future malware variants with low false positives and false negatives.

Network-level signatures have some attractive properties compared to system-level signatures. For example, enforcing system-level behavioral signatures often requires the use of virtualized environments and expensive dynamic analysis [21, 34]. On the other hand, network-level signatures are usually easier to deploy because we can take advantage of existing network monitoring infrastructures (e.g., intrusion detection systems and alert monitoring tools), and monitor a large number of machines without introducing overhead at the end hosts.

The vast majority of malware needs a network connection in order to perpetrate their malicious activities (e.g., sending spam, exfiltrating private data, downloading malware updates, etc.). In this paper, we focus on *network-level* behavioral clustering of HTTP-based malware, namely, malware that uses the HTTP protocol as its main means of communicating with the attacker or perpetrating their malicious intents.

HTTP-based malware is becoming more prevalent. For example, according to [20] the majority of spam botnets use HTTP to communicate with their command and control (C&C) server. Also, from our own malware database, we found that among the malware samples that show network activities, about 75% of them generate some HTTP traffic. In addition, there is evidence that Web-based "reusable" kits (or platforms) for remote command of malware, and in particular botnets, are available for sale on the Internet [14] (e.g., the C&C Web kit for *Zeus* bots can be currently purchased for about $700 [8]).

Given a large dataset of malware samples and the malicious HTTP traffic they generate, our network-level behavioral clustering system aims at unveiling similarities (or relationships) among malware samples that may not be captured by current *system-level* behavioral clustering systems [9, 10], thus offering a new point of view and valuable information to malware analysts. Unlike pre-

vious work on behavioral malware clustering, our work is motivated by the need to provide quality input to algorithms that automatically generate network signatures. Accordingly, we define similarity metrics among HTTP traffic traces and develop our clustering system so that the resulting clusters can yield high quality malware signatures. Namely, after clustering is completed, the HTTP traffic generated by malware samples in the same cluster can be processed by an automatic signature generation tool, in order to extract network signatures that model the HTTP behavior of all the malware variants in that cluster. An Intrusion Detection System (IDS) located at the edge of a network can in turn deploy such network signatures to detect malware-related outbound HTTP requests.

The main contributions of this paper are as follows:

- We propose a novel network-level behavioral malware clustering system based on the analysis of structural similarities among malicious HTTP traffic traces generated by different malware samples.
- We introduce a new automated method for analyzing the results of behavioral malware clustering based on a comparison with family names assigned to the malware samples by multiple AVs.
- We show that the proposed system enables accurate and efficient automatic generation of network-level malware signatures, which can complement traditional AVs and other defense techniques.
- We implemented a proof-of-concept version of our malware clustering system and performed experiments with more than 25,000 malware samples. Results from our evaluation, which includes real-world deployment, confirm the effectiveness of the proposed clustering system.

2 Related Work

System-level behavioral malware clustering has been recently studied in [9, 10]. In particular, Bayer et al. [10] proposed a scalable malware clustering algorithm based on malware behavior expressed in terms of detailed system events. However, the network information they use is limited to high-level features such as the names of downloaded files, the type of protocol, and the domain name of the server. Our work is different because we focus on the malicious HTTP traffic traces generated by executing different malware samples. We extract detailed information from the network traces, such as the number and type of HTTP queries, the length and structural similarities among URLs, the length of data sent and received from the HTTP server, etc. Compared with Bayer et al. [10], we do not consider the specific TCP port and domain names used by the malware. We aim to group together malware variants that may contact different web servers (e.g., because they are controlled by a different attacker), and may or may not use an HTTP proxy (whereby the TCP port used may vary), but have strong similarities in terms of the structure and sequence of the HTTP queries they perform (e.g., because they rely on the same C&C Web kit). Also, we develop our behavioral clustering algorithm so that the results can be used to automatically generate network signatures for detecting malicious network activities, as opposed to system-level signatures.

Automatic generation of network signatures has been explored in various previous work [23, 24, 29, 32, 33]. Most of these studies focused mainly on *worm fingerprinting*. Different approaches have been proposed to deal with generating signatures from a dataset of network flows related to the propagation of different worms. In particular, Polygraph [24] applies clustering techniques to try to separate worm flows belonging to different worms, before generating the signatures. However, Polygraph's clustering algorithm is *greedy* and becomes prohibitively expensive when dealing with the high number of malicious flows generated by a large dataset of different types of malware, as we will discuss in Section 6.2. Since behavioral malware clustering aims at efficiently clustering large datasets of different malware samples (including *bots*, *adware*, *spyware*, etc., beside *Worms*), the clustering approaches proposed for worm fingerprinting are not suitable for this task. Compared with [24] and other previous work on worm fingerprinting, we focus on clustering of different types of HTTP-based malware (not only worms) in an efficient manner.

BotMiner [15], an anomaly-based botnet detection system, applies clustering of network flows to detect the presence of bot-compromised machines within enterprise networks. BotMiner uses high-level statistics for clustering network flows, and is limited to detecting botnets. On the other hand, in this paper we focus on the behavioral clustering of generic malware samples based on structural similarities among their HTTP traffic traces, and on modeling the network behavior of the discovered malware families by extracting network-level malware detection signatures.

3 HTTP-Based Behavioral Clustering

The objective of our system is to find groups of malware that interact with the Web in a similar way, learn a *network behavior* model for each group (or family) of malware, and then use such models to detect the presence of malware-compromised machines in a monitored network. Towards this end, we first perform behavioral clustering of malware samples by finding structural similarities between the sequences of HTTP requests generated as a consequence of infection. Namely, given a dataset of malware samples $\mathcal{M} = \{m^{(i)}\}_{i=1..N}$, we execute each sample $m^{(i)}$ in a controlled environment similar to BotLab [20] for a time T, and we store its HTTP

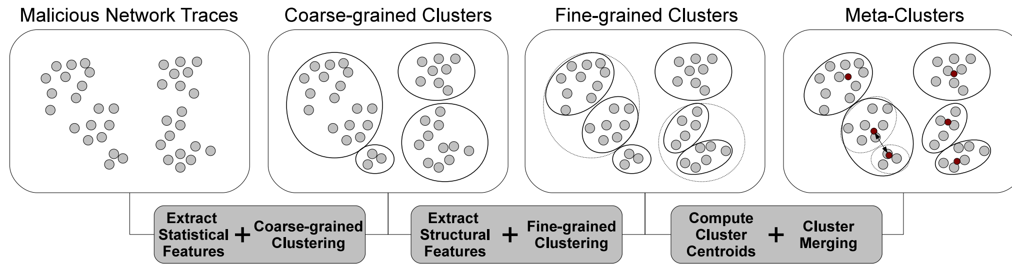


Figure 1: Overview of our HTTP-based behavioral malware clustering system.

traffic trace $\mathsf{H}(m^{(i)})$. We then partition $\mathcal{M}$ into clusters according to a definition of structural similarity among the HTTP traffic traces $\mathsf{H}(m^{(i)}), i = 1, .., N$.

3.1 System Overview

To attain high-quality clusters and decrease the computational cost of clustering, we adopt a multistep cluster-refinement process, as shown in Figure 1:

- *Coarse-grained Clustering*: In this phase, we cluster malware samples based on simple statistical features extracted from their malicious HTTP traffic. We measure features such as the total number of HTTP requests the malware generated, the number of GET and POST requests, the average length of the URLs, etc. Therefore, computing the distance between pairs of malware samples reduces to computing the distance between (short) vectors of numbers, which can be done efficiently.

- *Fine-grained Clustering*: After splitting the collected malware set into relatively large (coarse-grain) clusters, we further split each cluster into smaller groups. To this end, we consider each coarse-grained cluster as a separate malware set, measure the structural similarity between the HTTP traffic generated by each sample in a cluster, and apply fine-grained clustering. This allows us to separate malware that have similar statistical traffic characteristics (thus causing them to fall in the same coarse-grained cluster), but that present different structures of their HTTP queries. Measuring the structural similarity between pairs of HTTP traffic traces is relatively expensive. Since each coarse-grained cluster is much smaller than the total number of samples N, fine-grained clustering can be done more efficiently than by applying it directly on the entire malware dataset.

- *Cluster Merging*: The fine-grained clustering tends to produce "tight" clusters of malware that have very similar network behavior. However, one of our objectives is to derive *generic* behavior models that can be used to detect the network behavior of a large number of current and future malware samples. To achieve this goal, after fine-grained clustering we perform a further refinement step in which we try to merge together clusters of malware that have similar enough *HTTP behavior*, but that have been split by the fine-grained clustering process. In practice, given a set of fine-grained malware clusters, for each of them we define a *cluster centroid* as a set of network signatures that "summarize" the HTTP traffic generated by the malware samples in a cluster. We then measure the similarity between pairs of cluster centroids, and merge fine-grained clusters whose centroids are close to each other.

The combination of coarse-grained and fine-grained clustering allows us to decrease the computational cost of the clustering process, compared to using only fine-grained clustering. Furthermore, the cluster merging process allows us to attain more generic network-level malware signatures, thus increasing the malware detection rate (see Section 6.2). These observations motivate the use of our three-step clustering process.

In all the three phases of our clustering system, we apply single-linkage hierarchical clustering [19]. The main motivations for this choice are the fact that the hierarchical clustering algorithm is able to find clusters of arbitrary shapes, and can work on arbitrary metric spaces (i.e., it is not limited to distance in the Euclidean space). We ran pilot experiments using other clustering algorithms (e.g., X-means [27] for the coarse-grained clustering, and complete-linkage hierarchical clustering [19]). The single-linkage hierarchical clustering performed the best, according to our analysis.

The hierarchical clustering algorithm takes a matrix of pair-wise distances among objects as input and produces a *dendrogram*, i.e., a tree-like data structure where the leaves represent the original objects, and the length of the edges represent the distance between clusters [18]. Choosing the best clustering involves a *cluster validity* analysis to find the dendrogram cut that produces the most *compact and well separated* clusters. In order to automatically find the best dendrogram cut we apply the Davies-Bouldin (DB) cluster validity index [17]. We now describe our clustering system more in detail.

3.2 Coarse-grained Clustering

The goal of coarse-grained clustering is to find simple *statistical* similarities in the way different malware samples interact with the Web. Let $\mathcal{M} = \{m^{(i)}\}_{i=1..N}$ be a set of malware samples, and $\mathsf{H}(m^{(i)})$ be the HTTP traffic trace obtained by executing a malware $m^{(i)} \in \mathcal{M}$ for a given time T. We translate each trace $\mathsf{H}(m^{(i)})$ into a pattern vector $v^{(i)}$ containing the following statistical features to model how each malware uses the Web:

1. Total number of HTTP requests
2. Number of GET requests
3. Number of POST requests
4. Average length of the URLs
5. Average number of parameters in the request
6. Average amount of data sent by POST requests
7. Average response length

Because the range of different features in the pattern vectors are quite different, we first standardize the dataset so that the features will have mean equal to zero and variance equal to one, and then we apply the Euclidian distance. We partition the set $\mathcal{M}$ into coarse-grained clusters by applying the single-linkage hierarchical clustering algorithm and DB index [17] cluster validity analysis.

3.3 Fine-grained Clustering

In the fine-grained clustering step, we consider the *structural* similarity among sequences of HTTP requests (as opposed to the statistical similarity used for coarse-grained clustering). Our objective is to group together malware that interact with Web applications in a similar way. For example, we want to group together bots that rely on the same Web-based C&C application. Our approach is based on the observation that two different malware samples that rely on the same Web server application will query URLs structured in a similar way, and in a similar sequence. In order to capture these similarities, we first define a measure of distance between two HTTP requests r_k and r_h generated by two different malware samples. Consider Figure 2, where m, p, n, and v, represent different parts of an HTTP request:

- m represents the *request method* (e.g., GET, POST, HEADER, etc.). We define a distance function $d_m(r_k, r_h)$ that is equal to 0 if the requests r_k, and r_h both use the same method (e.g, both are GET requests), otherwise it is equal to 1.
- p stands for *page*, namely the first part of the URL that includes the path and page name, but does not include the parameters. We define $d_p(r_k, r_h)$ to be equal to the normalized Levenshtein distance[1] between the strings related to the path and pages that appear in the two requests r_k and r_h.
- n represents the set of parameter names (i.e., $n = \{id, version, cc\}$ in the example in Figure 2). We define $d_n(r_k, r_h)$ as the Jaccard distance[2] between the sets of parameters names in the two requests.
- v is the set of parameter values. We define $d_v(r_k, r_h)$ to be equal to the normalized Levenshtein distance between strings obtained by concatenating the parameter values (e.g., `0011.0US`).

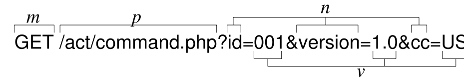


Figure 2: Structure of an HTTP request used in fine-grained clustering. m=Method; p=Page; n=Parameter Names; v=Parameter Values.

We define the overall distance between two HTTP requests as

$$\begin{aligned} d_r(r_k, r_h) = & w_m \cdot d_m(r_k, r_h) + w_p \cdot d_p(r_k, r_h) \\ & + w_n \cdot d_n(r_k, r_h) + w_v \cdot d_v(r_k, r_h) \end{aligned} \quad (1)$$

where the factors $w_x, x \in \{m, p, n, v\}$ are predefined weights (the actual value assigned to the weights w_x are discussed in Section 6) that give more importance to the distance between the request methods and pages, for example, and less weight to the distance between parameter values. We then define the *fine-grain* distance between two malware samples as the average minimum distance between sequences of HTTP requests from the two samples, and apply the single-linkage hierarchical clustering algorithm and the DB cluster validity index [17] to split each coarse-grained cluster into fine-grained clusters (we only split coarse-grained clusters whose diameter is larger than a predefined threshold $\theta = 0.1$).

3.4 Cluster Merging

Fine-grained clustering tends to produce tight clusters, which yield specific malware signatures. However, our objective is to derive generic malware signatures which can be used to detect as many future malware variants as possible, while maintaining a very low false positive rate. Towards this end, we apply a further refinement step in which we merge together fine-grained clusters of malware variants that behave similarly enough, in terms of the HTTP traffic they generate. For each fine-grained malware cluster we compute a cluster *centroid*, which summarizes the HTTP requests performed by the malware samples in a cluster, and then we define a measure of distance among centroids (and therefore among clusters). The cluster merging phase is a *meta-clustering* step in which we find groups of malware clusters that are very

[1]The normalized Levenshtein distance between two strings s_1 and s_2 (also known as *edit* distance) is equal to the minimum number of character operations (insert, delete, or replace) needed to transform one string into the other, divided by $\max(length(s1), length(s2))$.

[2]The Jaccard distance between two sets A and B is defined as $J(A,B) = 1 - \frac{|A \cap B|}{|A \cup B|}$

```
        t1                  t2                  t3          t4
a)   GET /.*/command\.php\?id=1\..*&version=.*&cc=.*

b)      GET //command.php?id=1.&version=&cc=
```

Figure 3: Example of network signature (a), and its plain text version (b).

close to each other, and we merge them to form bigger clusters.

Cluster Centroids Let $\mathbf{C}_i$ be a cluster of malware samples, and $\mathbf{H}_i = \{\mathsf{H}(m_k^{(i)})\}_{k=1..c_i}$ the related set of HTTP traffic traces obtained by executing each malware sample in $\mathbf{C}_i$. We define the centroid of $\mathbf{C}_i$ as a set $\mathbf{S}_i = \{s_j\}_{j=1..l_i}$ of network signatures. Each signature s_j is extracted from a *pool* p_j of HTTP requests selected from the traffic traces in $\mathbf{H}_i$. We first describe the algorithm used for creating the set of HTTP request pools, and then we describe how the signatures are extracted from the obtained pools.

To create a set $\mathbf{P}_i$ of request pools, we first randomly select one of the malware samples in cluster $\mathbf{C}_i$ to be our *centroid seed*. Assume we pick $m_h^{(i)}$ for this purpose. We then consider the set of HTTP requests in the HTTP traffic trace $\mathsf{H}(m_h^{(i)}) = \{r_j\}_{j=1..l_i}$. We initialize the pool set $\mathbf{P}_i$ by putting each request r_j in a different (until now empty) pool p_j. Now, using the definition of distance between HTTP requests in Equation 1, for each request $r_j \in \mathsf{H}(m_h^{(i)})$ we find the closest request $r'_k \in \mathsf{H}(m_g^{(i)})$ from another malware sample $m_g^{(i)} \in \mathbf{C}_i$, and we add r'_k to the pool p_j. We repeat this for all the malware $m_g^{(i)} \in \mathbf{C}_i, g \neq h$. After this process is complete, and pool p_j has been filled with HTTP requests, we reiterate the same process to construct pool $p_{j' \neq j}$ starting from request $r_{j'} \in \mathsf{H}(m_h^{(i)})$, until all pools $p_j, j = 1, .., l_i$ have been filled.

After the pools have been filled with HTTP requests, we extract a signature s_j from each pool $p_j \in \mathbf{P}_i$ using the *Token-Subsequences* algorithm implemented in [24] (Token-Subsequences signatures can be easily translated into Snort signatures). Since the signature generation algorithm itself is not a contribution of this paper, we refer the reader to [24] for more details on how the Token Subsequences signatures are generated. Here it is sufficient to notice that a Token Subsequences signature is an ordered list of invariant tokens, i.e, substrings that are in common to all the requests in a request pool p. Therefore, a signature s_j can be written as a regular expression of the kind `t1.*t2.*...*tn`, where the `t`'s are *invariant tokens* that are common to all the requests in the pool p_j. We consider only the first part of each HTTP request for signature generation purposes, namely, the request method and URL (see Figure 3a).

Meta-Clustering After a centroid has been computed for each fine-grained cluster, we can compute the distance between pairs of centroids $d(\mathbf{S}_i, \mathbf{S}_j)$,. We first define the distance between pairs of signatures, and then we extend this definition to consider sets of signatures. Let s_i be a signature, and s'_j be a *plain text* concatenation of the invariant tokens in signature s_j. For example, `t1t2t3` is a plain text version of the signature `t1.*t2.*t3` (see Figure 3 for a concrete example). We define the distance between two signatures as

$$d(s_i, s_j) = \frac{agrep(s_i, s'_j)}{length(s'_i)} \in [0, 1] \qquad (2)$$

where $agrep(s_i, s'_j)$ is a function that performs approximate matching of regular expression [31] of the signature s_i on the string s'_j, and returns the number of matching errors. In practice, $d(s_i, s_j)$ is equal to zero when s_i perfectly "covers" (i.e., is more generic than) s_j, and tends to one when signatures s_i and s_j are more and more different.

Given the above definition of distance between signatures, we define the distance between two centroids (i.e., two clusters) as the minimum average distance between two sets of signatures[3]. It is worth noting that when computing the distance between two centroids, we only consider those signatures s_k for which $length(s'_k) \geqslant \lambda$. Here s'_k is again the *plain text* version of s_k, $length(s'_k)$ is the length of the string s'_k, and λ is a predefined length threshold. The threshold λ is chosen to avoid applying the $agrep$ function on short, and therefore likely too generic, signatures that would match most HTTP requests (e.g., s_k = `GET /.*`), thus artificially skewing the distance value towards zero.

We then apply again the hierarchical clustering algorithm in combination with the DB validity index [17] to find groups of clusters (or meta-clusters) that are close to each other and should therefore be merged.

4 Network Signatures

The cluster-merging step described in Section 3.4 represents the last phase of our behavioral clustering process, and its output represents the final partitioning of the original malware set $\mathcal{M} = \{m^{(i)}\}_{i=1..N}$ into groups of malware that share similar HTTP behavior. Now, for each of the final output clusters $\mathbf{C}'_i, i = 1, .., c$, we can compute an "updated" centroid signature set $\mathbf{S}'_i$ using the same algorithm described in Section 3.4 for computing cluster centroids. The signature set $\mathbf{S}'_i$ can then be deployed into an IDS at the edge of a network in order to detect malicious HTTP requests, which are a symptom of malware infection.

It is important to notice that some malware samples may contact legitimate websites for malicious purposes.

[3]Formally
$d(\mathbf{S}_i, \mathbf{S}_j) = \min\left\{\frac{1}{l_i}\sum_i \min_j\{d(s_i, s_j)\}, \frac{1}{l_j}\sum_j \min_i\{d(s_j, s_i)\}\right\}$

For example, some botnets use `facebook` or `twitter` for C&C [3]. To decrease the possibility of false positives, one may be tempted to prefilter all the HTTP requests sent by malware samples against well known, legitimate websites before generating the network signatures. However, prefiltering all the HTTP requests against these websites may not be a good idea because we may discard HTTP requests that, although "served" by legitimate websites, are specific to certain malware families and whose related network signatures may yield a high detection rate with low false positives. To solve this problem, instead of prefiltering HTTP traffic against legitimate websites, we apply a *signature pruning* process by testing the signature set $\mathbf{S}'_i$ against a large dataset of legitimate traffic and discard the signatures that generate false positives.

5 Cluster Validity Analysis

Clustering can be viewed as an *unsupervised learning* task, and analyzing the *validity* of the clustering results is intrinsically hard. Cluster validity analysis often involves the use of a subjective criterion of optimality [19], which is specific to a particular application. Therefore, no standard way exists of validating the output of a clustering procedure [19]. As discussed in Section 3, we make use of the DB validity index [17] in all the phases of our malware clustering process to automatically choose the best possible partitioning of the malware dataset. However, it is also desirable to analyze the clustering results by quantifying the level of agreement between the obtained clusters and the information about the clustered malware samples given by different AV vendors, for example.

Bayer et al. [10] proposed to use *precision* and *recall* (which are widely used in text classification problems, for example, but not as often for cluster validity analysis) to compare the results of their system-level behavioral clustering system to a *reference clustering*. Generating such reference clustering is not easy because the labels assigned by different AV scanners to variants of the same malware family are seldom consistent. This required Bayer et al. [10] to define a mapping between labels assigned by different AVs.

We propose a new approach to analyze the validity of malware clustering results, which does not require any manual mapping of AV labels. Our approach is based on a measure of the *cohesion* (or compactness) of each cluster, and the *separation* among different clusters. We measure both cohesion and separation in terms of the agreement between the labels assigned to the malware samples in a cluster by multiple AV scanners. It is worth noting, though, that since the AV labels themselves are not always consistent (as observed in [9, 10]), our measures of cluster cohesion and separation give us an indication of the validity of the clustering results, rather than being an *oracle*. However, we devised our cluster cohesion and separation indices to mitigate possible inconsistencies among AV labels.

AV Label Graphs Before describing how cluster cohesion and separation are measured, we need to introduce the notion of *AV label graph*. We introduce AV label graphs to mitigate the effects of the inconsistency of AV labels, and to map the problem of measuring the cohesion (or compactness) and separation of clusters in terms of easier-to-handle graph-based indices. We first start with an example to show how to construct the AV label graph given a cluster of malware samples. We then provide a more formal definition.

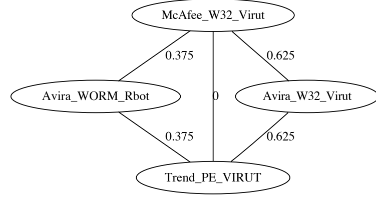
Consider the example of malware cluster in Figure 4a, which contains eight malware samples (one per line). Each line reports the MD5 hash of a malware sample, and the AV labels assigned to the sample by three different AV scanners (McAfee [4], Avira [1], and Trend Micro [7]). From this malware cluster we construct an AV label graph as follows:

1. Create a node in the graph for each distinct AV malware family label (we identify a malware family label by extracting the first AV label substring that ends with a '.' character). For example (see Figure 4b), the first malware sample is classified as belonging to the `W32/Virut` family by McAfee, `WORM/Rbot` by Avira, and `PE_VIRUT` by Trend Micro. Therefore we create three nodes in the graph called `McAfee_W32_Virut`, `Avira_WORM_Rbot`, and `Trend_PE_VIRUT` (in case a malware sample is not detected by an AV scanner, we map it to a special *null* label).
2. Once all the nodes have been created, we connect them using weighted edges. We connect two nodes with an edge only if the related two malware family labels (i.e., the name of the nodes) appear together in at least one of the lines in Figure 4a.
3. A weight equal to $1 - \frac{m}{n}$ is assigned to each edge, where m represents the number of times the two malware family labels connected by the edge have appeared on the same line in the cluster (i.e., for the same malware sample), and n is the total number of samples in the cluster ($n = 8$ in this example).

As we can see from Figure 4b, the nodes `McAfee_W32_Virut` and `Trend_PE_VIRUT` are connected by an edge with weight equal to zero because both McAfee and Trend Micro consistently classify each malware sample in the cluster as `W32/Virut` and `PE_VIRUT`, respectively (i.e., $m = n$). On the other hand, the edge between nodes `McAfee_W32_Virut` and `Avira_W32_Virut`, for example, was assigned a weight equal to 0.625 because in this case $m = 3$. We now define AV label graphs more formally.

```
65c862e240e2560245c4e25a568377ce m=W32/Virut.gen a=WORM/Rbot.50176.5 t=PE_VIRUT.D-1
662b544694c6a6235f7eb66a44bed26f m=W32/Virut.gen a=WORM/Rbot.50176.5 t=PE_VIRUT.D-2
6fec875d853898d4ca47daddfe831c43 m=W32/Virut.gen a=W32/Virut.Gen      t=PE_VIRUT.D-4
8e89e61e06362bfa29d4f72284c52a2e m=W32/Virut.gen a=W32/Virut.X        t=PE_VIRUT.XO-2
94fd3b8f56e34d9123c9d5c56a9c87a0 m=W32/Virut.gen a=WORM/Rbot.50176.5 t=PE_VIRUT.D-2
96697634e1dbdec28bb75d54cb9559a5 m=W32/Virut.gen a=W32/Virut.H        t=PE_VIRUT.NS-2
9961446cae40297df49537c5c4b3b78a m=W32/Virut.gen a=WORM/Rbot.50176.5 t=PE_VIRUT.D-2
a548fec2eb98a5516934fe6f289f3910 m=W32/Virut.gen a=WORM/Rbot.50176.5 t=PE_VIRUT.D-1
```

(a) Malware Cluster



(b) AV Label Graph

Figure 4: Example of Malware Cluster (a) and related AV Label Graph (b). Each malware sample (identified by its MD5 hash) is labeled using three different AV scanners, namely McAfee (m), Avira (a), and Trend Micro (t).

Definition 1 - ***AV Label Graph***. *An AV label graph is an undirected weighted graph. Given a malware cluster $\mathbf{C}_i = \{m_k^{(i)}\}_{k=1..c_i}$, let $\Gamma_i = \{L_1 = (l_1, .., l_v)_1, .., L_{c_i} = (l_1, .., l_v)_{c_i}\}$ be a set of label vectors, where label vector $L_h = (l_1, .., l_v)_h$ is the set of malware family labels assigned by v different AV scanners to malware $m_h^{(i)} \in \mathbf{C}_i$. The AV label graph $\mathcal{G}_i = \{V_k^{(i)}, E_{k_1,k_2}^{(i)}\}_{k=1..l}$ is constructed by adding a node $V_k^{(i)}$ for each distinct malware family label $l_k \in \Gamma_i$. Two nodes $V_{k_1}^{(i)}$ and $V_{k_2}^{(i)}$ are connected by a weighted edge $E_{k_1,k_2}^{(i)}$ only if the malware family labels l_{k_1} and l_{k_2} related to the two nodes appear at least once in the same label vector $L_h \in \Gamma_i$. Each edge $E_{k_1,k_2}^{(i)}$ is assigned a weight $w = 1 - \frac{m}{c_i}$, where m is equal to the number of label vectors $L_h \in \Gamma_i$ containing both l_{k_1} and l_{k_2}, and c_i is the number of malware samples in $\mathbf{C}_i$.*

Cluster Cohesion and Separation Now that we have defined AV label graphs, we can formally define cluster cohesion and separation in terms of AV labels.

Definition 2 - ***Cohesion Index***. *Given a cluster $\mathbf{C}_i$, let $\mathcal{G}_i = \{V_k^{(i)}, E_{k_1,k_2}^{(i)}\}_{k=1..l}$ be its AV label graph, and δ_{l_1,l_2} be the shortest path between two nodes $V_{l_1}^{(i)}$ and $V_{l_2}^{(i)}$ in $\mathcal{G}_i$. If no path exists between the two nodes, the distance δ_{l_1,l_2} is assumed to be equal to a constant "gap" $\gamma \gg \sup(w_{k_1,k_2})$, where w_{k_1,k_2} is the weight of a generic edge $E_{k_1,k_2}^{(i)} \in \mathcal{G}_i$. The cohesion index of cluster $\mathbf{C}_i$ is defined as*

$$\mathtt{C}(\mathbf{C}_i) = 1 - \frac{1}{\gamma} \frac{2}{n \cdot v(n \cdot v - 1)} \sum_{l_1 < l_2} \delta_{l_1,l_2}$$

where n is the number of malware samples in the cluster, and v is the number of different AV scanners.

According to our definition of AV label graph, $\sup(w_{k_1,k_2}) = 1$, and we set $\gamma = 10$. In practice, the cohesion index $\mathtt{C}(\mathbf{C}_i) \in (0, 1]$ will be equal to one when each AV scanner consistently assigns the same malware family label to each of the malware samples in cluster $\mathbf{C}_i$. On the other hand the cohesion index will tend to zero if each AV scanner assigns different malware family labels to each of the malware samples in the cluster. For example, the graph in Figure 4b has a cohesion index equal to 0.999. The cohesion index is very high thanks to the fact that both McAfee and Trend Micro consistently assign the same family label to all the samples in the cluster. If Avira also consistently assigned the same family label to all the samples (either always `Avira_W32_Virut` or always `Avira_W32_Rbot`), the cohesion index would be equal to one. As we can see, regardless of the inconsistency in Avira's labels, thanks to the fact that we use multiple AV scanners and we leverage the notion of AV label graphs, we can correctly consider the cluster in Figure 4a as very compact, thus confirming the validity of the behavioral clustering process.

Definition 3 - ***Separation Index***. *Given two clusters $\mathbf{C}_i$ and $\mathbf{C}_j$ and their respective label graphs $\mathcal{G}_i$ and $\mathcal{G}_j$, let $\mathbf{C}_{ij}$ be the cluster obtained by merging $\mathbf{C}_i$ and $\mathbf{C}_j$, and $\mathcal{G}_{ij}$ be its label graph. By definition, $\mathcal{G}_{ij}$ will contain all the nodes $V_k^{(i)} \in \mathcal{G}_i$ and $V_h^{(j)} \in \mathcal{G}_j$. The separation index between $\mathbf{C}_i$ and $\mathbf{C}_j$ is defined as*

$$\mathtt{S}(\mathbf{C}_i, \mathbf{C}_j) = \frac{1}{\gamma} \operatorname{avg}_{k,h}\{\Delta(V_k^{(i)}, V_h^{(j)})\}$$

where $\Delta(V_k^{(i)}, V_h^{(j)})$ is the shortest path in $\mathcal{G}_{ij}$ between nodes $V_k^{(i)}$ and $V_h^{(j)}$, and γ is the "gap" introduced in Definition 2.

In practice, the separation index takes values in the interval $[0, 1]$. $\mathtt{S}(\mathbf{C}_i, \mathbf{C}_j)$ will be equal to zero if the malware samples in clusters $\mathbf{C}_i$ and $\mathbf{C}_j$ are all consistently labeled by each AV scanner as belonging to the same malware family. Higher values of the separation index indicate that the malware samples in $\mathbf{C}_i$ and $\mathbf{C}_j$ are more and more diverse in term of malware family labels, and are perfectly separated (i.e., $\mathtt{S}(\mathbf{C}_i, \mathbf{C}_j) = 1$) when no intersection exists between the malware family labels assigned to malware samples in $\mathbf{C}_i$, and the ones assigned to malware samples in $\mathbf{C}_j$.

6 Experiments

In this section we present our experimental results.

	Malware Samples			Number of Clusters			Processing Time		
dataset	**samples**	**undetected by all AVs**	**undetected by best AV**	**coarse**	**fine**	**meta**	**coarse**	**fine**	**meta+sig**
Feb09	4,758	208 (4.4%)	327 (6.9%)	2,538	2,660	1,499	34min	22min	6h55min
Mar09	3,563	252 (7.1%)	302 (8.6%)	2,160	2,196	1,779	19min	3min	1h3min
Apr09	2,274	142 (6.2%)	175 (7.7%)	1,325	1,330	1,167	8min	5min	28min
May09	4,861	997 (20.5%)	1,127 (23.2%)	3,339	3,423	2,593	56min	8min	2h52min
Jun09	4,677	1,038 (22.2%)	1,164 (24.9%)	3,304	3,344	2,537	57min	3min	37min
Jul09	5,587	1,569 (28.1%)	1,665 (29.8%)	3,358	3,390	2,724	1h5min	5min	2h22min

Table 1: Summary of Clustering Results (column **meta+sig** includes the meta-clustering and signature extraction processing time).

6.1 HTTP-Based Behavioral Clustering

Malware Dataset Our malware dataset consists of 25,720 distinct (no duplicates) malware samples, each of which generates at least one HTTP request when executed on a victim machine. We collected our malware samples in a period of six months, from February to July 2009, from a number of different malware sources such as MWCollect [2], Malfease [5], and commercial malware feeds. Table 1 (first and second column), shows the number of distinct malware samples collected in each month. Similar to previous works that rely on an analysis of malware beahavior [9, 10, 20], we executed each sample in a controlled environment for a period $T = 5$ minutes, during which we recorded the HTTP traffic to be used for our behavioral clustering (see Section 3).

To perform cluster analysis based on AV labels, as described in Section 5, we scanned each malware sample with three commercial AV scanners, namely McAfee [4], Avira [1], and Trend Micro [7]. As we can see from Table 1 (third and fourth column), each of our datasets contains a number of malware samples which are not detected by any of our AV scanners. In addition, the number of undetected samples grew significantly during the last few months, for both the combination of the three scanners, and for the single best AV (i.e., the AV scanner that overall detected the highest number of samples). This is justified by the fact that we scanned all the binaries in August 2009 using the most recent AV signatures. Therefore, AV companies had enough time to generate signatures for most malware collected in February, for example, but evidently not enough time to generate signatures for many of the more recent malware samples. Given the rapid pace at which new malware samples are created [30], and since it may take months for AV vendors to collect a specific malware variant and generate traditional detection signatures for it, this result was somewhat expected and is in accordance with the results reported by Oberheide et al. [26].

Experimental Setup We implemented a proof-of-concept version of our behavioral clustering system (see Section 3), which consists of a little over 2,000 lines of Java code. We set the weights defined in Equation 1 (as explained in Section 3.3) to $w_m = 10$, $w_p = 8$, $w_n = 3$, and $w_v = 1$. We set the minimum signature length λ used to compute the distance between cluster centroids (see Section 3.4) to 10. To perform fine- and meta-clustering (see Section 3.4), we considered the first 10 HTTP requests generated by each malware sample during execution. We performed approximate matching of regular expressions (see *agrep* function in Section 3.4) using the TRE library [22]. All the experiments were performed on a 4-core 2.67GHz Intel Core-i7 machine with 12GB of RAM, though we never used more than 2 cores and 8GB of RAM for each experiment run.

Clustering Results We applied our behavioral clustering algorithm to the malware samples collected in each of the six months of observation. Table 1 summarizes our clustering results, and reports the number of clusters produced by each of the clustering refinement steps, i.e., coarse-grain, fine-grained, and meta-clustering (see Section 3). For example, in February 2009, we collected 4,758 distinct malware samples. The coarse-grained clustering step grouped them into 2,538 clusters, the fine-grained clustering further split some of these clusters to generate a total of 2,660 clusters, and the meta-clustering process found that some of the fine-grained clusters could be merged to produce a final number of 1,499 (meta-)clusters. Table 1 also reports the time needed to complete each step of our clustering process. The most expensive step is almost always the meta-clustering (see Section 3.4) because measuring the distance between centroids requires using the *agrep* function for approximate matching of regular expressions, which is relatively expensive to compute. However, computing the clusters for one month of HTTP-based malware takes only a few hours. The variability in clustering time is due to the different number of samples per month, and by the different amount of HTTP traffic they generated during execution. Further optimizations of our clustering system are left as future work.

Table 7 (first and second row) shows, for each month, the number of clusters and the clustering time obtained by directly applying the fine-grained clustering step alone to our malware datasets (we will explain the meaning of the last row of Table 7 later in Section 6.2). We can see from Table 1 that the combination of coarse-grained and fine-grained clustering requires a lower computation time, compared to applying fine-grained clustering by itself. For example, according to Table 1, computing the coarse-grained clusters first and then refining the results using fine-grained clustering on the *Feb09* dataset takes 56 minutes. On the other hand, according to Table 7, applying fine-grained clustering directly on *Feb09*

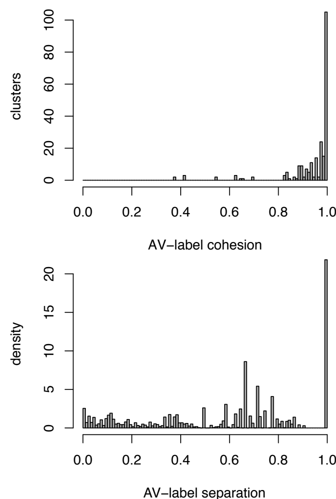


Figure 5: Distribution of cluster *cohesion* and *separation* (*Feb09*).

```
v1 = 076b81e8c6622e9c6a94426e8c2dfe33
AV Labels = Generic FakeAlert.h [McAfee];
            TR/Dropper.Gen [Avira]

HTTP Traffic

[1249356561 192.168.14.2:1037 => 94.247.2.193:80]
POST /cgi-bin/generator HTTP/1.0
Content-Length: 45
[... DATA ...]

[1249356562 192.168.14.2:1038 => 94.247.2.193:80]
POST /extra.php HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 44
[... DATA ...]

File System Operations

Delete c:\docume~1\admini~1\locals~1\temp\tmp1.tmp
Read   \\?\globalroot\systemroot\system32\msvcrt.dll
Write  c:\docume~1\admini~1\locals~1\temp\tmp1.tmp
```

(a) Variant 1

```
v2 = 7ee251d8d13ed32914a4e39740b91ae2
AV Labels = DR/PCK.Tdss.A.21 [Avira]

HTTP Traffic

[1249345674 192.168.12.2:1034 => 94.247.2.193:80]
POST /cgi-bin/generator HTTP/1.0
Content-Length: 45
[... DATA ...]

[1249345674 192.168.12.2:1038 => 94.247.2.193:80]
POST /extra.php HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 44
[... DATA …]

File System Operations

Delete c:\docume~1\admini~1\locals~1\temp\tmp4.tmp
Delete c:\docume~1\admini~1\locals~1\temp\tmp5.tmp
Write  c:\docume~1\admini~1\locals~1\temp\tmp5.tmp
Read   \\?\globalroot\systemroot\system32\advapi32.dll
Write  c:\docume~1\admini~1\locals~1\temp\tmp4.tmp
Write  c:\docume~1\admini~1\locals~1\temp\nso3.tmp\modern-header.bmp
Delete c:\docume~1\admini~1\locals~1\temp\nso3.tmp
Write  c:\docume~1\admini~1\locals~1\temp\matrix329411.exe
Read   (MALWARE_PATH)
Delete c:\docume~1\admini~1\locals~1\temp\nsc1.tmp
```

(b) Variant 2

Figure 6: Example of malware variants that generate the very same network traffic, but also generate significantly different system events.

requires more than 4 hours. Furthermore, although applying fine-grained clustering by itself requires less time than our three-step clustering approach (which includes meta-clustering) for three out of six datasets, our three-step clustering yields better signatures and a higher malware detection rate in all cases, as we discuss in Section 6.2.

To analyze the quality of the final clusters generated by our system we make use of the cluster *cohesion* and *separation* defined in Section 5. Figure 5 shows a histogram of the cohesion index values (top graph) computed for each of the clusters obtained from the *Feb09* malware dataset, and the distribution of the separation among clusters (bottom graph). Because of space limitations, we only discuss the cohesion and separation results from *Feb09*. The cohesion histogram only considers clusters that contain two or more malware samples (clusters containing only one sample have cohesion equal to 1 by definition). Ideally, we would like the value of cohesion for each cluster to be as close as possible to 1. Figure 5 confirms the effectiveness of our clustering approach. The vast majority of the behavioral clusters generated by our clustering system are very compact in terms of AV label graphs. This shows a strong agreement between our results and the malware family labels assigned to the malware samples by the AV scanners.

Figure 5 also shows the distribution of the separation between pairs of malware clusters. Ideally we would like all the pairs of clusters to be perfectly separated (i.e., with a separation index equal to 1). Figure 5 (bottom graph) shows that most pairs of clusters are relatively well separated from each other. For example, 90% of all the cluster pairs from *Feb09* have a separation index higher than 0.1. Both cluster cohesion and separation provide a comparison with the AV labels, and although our definition of cohesion and separation indexes attenuates the effect of AV label inconsistency (see Section 5), the results ultimately depend on the quality of the AV labels themselves. For example, we noticed that most pairs of clusters that have a low separation are due to the fact that their malware samples are labeled by the AV scanners as belonging to generic malware families, such as "Generic", "Downloader", or "Agent".

Overall, the distributions of the cohesion and separation indexes in Figure 5 show that most of the obtained behavioral malware clusters are very compact and fairly well separated, in terms of AV malware family labels. By combining this automated analysis with the manual analysis of those cases in which the separation index seemed to disagree with our clustering, we were able to confirm that our network-level clustering approach was indeed able to accurately cluster malware samples according to their network behavior.

6.2 Network Signatures

In this section, we discuss how our network-level behavioral malware clustering can aid the automatic generation of network signatures. The main idea is to periodically extract signatures from newly collected malware samples, and to measure the effectiveness of such signatures for detecting the malicious HTTP traffic generated by current and future malware variants.

Table 2 summarizes the results of the automatic signature generation process. For each month worth of HTTP-based malware, we considered only the malware clusters containing at least 2 samples. We do not consider signatures from singleton clusters because they are too specific and not representative of a *family* of malware. We ex-

dataset	clusters ($n > 1$)	samples	signatures	pruned sig.
Feb09	235	3,494	544	446
Mar09	290	2,074	721	627
Apr09	178	1,285	378	326
May09	457	2,725	1,013	833
Jun09	492	2,438	974	915
Jul09	567	3,430	1,367	1,225

Table 2: Automatic signature generation and pruning results (processing times for signature extraction are included in Table 1, **meta+sig** column).

	Feb09	*Mar09*	*Apr09*	*May09*	*Jun09*	*Jul09*
Sig_Feb09	85.9%	50.4%	47.8%	27.0%	21.7%	23.8%
Sig_Mar09	-	64.2%	38.1%	25.6%	23.3%	28.6%
Sig_Apr09	-	-	63.1%	26.4%	27.6%	21.6%
Sig_May09	-	-	-	59.5%	46.7%	42.5%
Sig_Jun09	-	-	-	-	58.9%	38.5%
Sig_Jul09	-	-	-	-	-	65.1%

Table 3: Signature detection rate on current and *future* malware samples (1 month training)

tracted a signature set from each of the considered clusters as explained in Section 4. For example (Table 2, row 1), for the *Feb09* malware dataset our clustering system found 235 clusters that contained at least 2 malware samples. The cumulative number of distinct samples contained in the 235 clusters was 3,494, from which the automatic signature generation process extracted a total of 544 signatures. After signature pruning (explained below) the number of signatures was reduced to 446.

To perform signature pruning (see Section 4), we proceeded as follows. We collected a dataset of legitimate traffic by sniffing the HTTP requests crossing the web-proxy of a large, well administered enterprise network with strict security policies for about 2 days, between November 25 and November 27, 2008. The collected dataset of legitimate traffic contained over $25.3 \cdot 10^6$ HTTP requests from 2,010 clients to thousands of different Web sites. We used existing automatic techniques for detecting malicious HTTP traffic and manual analysis to confirm that the collected HTTP traffic was actually as clean as possible. We split this dataset in two parts. We used the first day of traffic for signature pruning, and the second day to estimate the false positive rate of our pruned signatures (we will discuss our findings regarding false positives later in this section). To prune the 544 signatures extracted from *Feb09*, we translated the signatures in a format compatible with Snort [6], and then we used Sort's detection engine to run our signatures over the first day of legitimate traffic. We then pruned those signatures that generated any alert, thus leaving us with 446 signatures. We repeated this pruning process for all the signature sets we extracted from the other malware datasets. In the following, we will refer to the pruned set of signatures extracted from *Feb09* as *Sig_Feb09*, and similarly for the other months *Sig_Mar09*, *Sig_Apr09*, etc.

Detection Rate We measured the ability of our signatures to detect current and *future* malware samples. We measured the detection rate of our automatically generated signatures as follows. Given the signatures in the set *Sig_Feb09*, we matched them (using Snort) over the HTTP traffic traces generated by malware samples in *Feb09*, *Mar09*, *Apr09*, etc. We repeated the same process by testing the signatures extracted from a given month on the HTTP traffic generated by the malware collected in that month and in future months. We consider a malware sample to be detected if its HTTP traffic causes at least one alert to be raised. The detection results we obtained are summarized in Table 3. Take as an example the first row. The signature set *Sig_Feb09* "covers" (i.e., is able to detect) 85.9% of the malware samples collected in *Feb09*, 50.4% of the malware samples collected in *Mar09*, 47.8% of the malware samples collected in *Apr09*, and so on. Therefore, each of the signature sets we generated is able to generalize to new, never-beforseen malware samples. This is due to the fact that our network signatures aim to "summarize" the behavior of a malware family, instead of individual malware samples. As we discussed before, while malware variants from the same family can be generated at a high pace (e.g., using executable packing tools [16]), when executed they will behave similarly, and therefore can be detected by our behavioral network signatures. Naturally, as malware behavior evolves, the detection rate of our network signatures will decrease over time. Also, our approach is not able to detect "unique" malware samples, which behave differently from any of the malware groups our behavioral clustering algorithm was able to identify. Nonetheless, it is evident from Table 3 that if we periodically update our signatures with a signature set automatically extracted from the most recent malware samples, we can maintain a relatively high detection rate on current and future malware samples.

False Positives To measure the false positives generated by our network signatures we proceeded as follows. For each of the signature sets *Sig_Feb09*, *Sig_Mar09*, etc., we used Snort to match them against the second day of legitimate HTTP traffic collected as described at the beginning of this Section. Table 4 summarizes the results we obtained. The first row reports the false positive rate, measured as the total number of alerts generated by a given signature set divided by the number of HTTP requests in the legitimate dataset. The numbers between parentheses represent the absolute number of alerts raised. On the other hand, the second row reports the fraction of distinct source IP addresses that were deemed to be compromised, due to the fact that some of their HTTP traffic matched any of our signatures. The numbers between parenthesis represent the absolute number of the source IPs for which an alert was raised. The results reported in Table 4 show that our signatures generate a low false positive rate. Furthermore, matching our signatures against one entire day of legit-

	Sig_Feb09	*Sig_Mar09*	*Sig_Apr09*	*Sig_May09*	*Sig_Jun09*	*Sig_Jul09*
FP rate	0% (0)	$3 \cdot 10^{-4}$% (38)	$8 \cdot 10^{-6}$% (1)	$5 \cdot 10^{-5}$% (6)	$2 \cdot 10^{-4}$% (26)	10^{-4}% (18)
Distinct IPs	0% (0)	0.3% (6)	0.05% (1)	0.2% (4)	0.4% (9)	0.3% (7)
Processing Time	13 min	10 min	6 min	9 min	12 min	38 min

Table 4: False positives measured on one day of legitimate traffic (approximately 12M HTTP queries from 2,010 different source IPs).

imate traffic (about 12M HTTP queries from 2,010 distinct source IPs) can be done in minutes. This means that we would "keep up" with the traffic in real-time.

	Apr09	*May09*	*Jun09*	*Jul09*
Sig_Feb09-Apr09	70.8%	35.6%	36.4%	35.1%
Sig_Mar09-May09	-	61.6%	48.6%	44.7%
Sig_Apr09-Jun09	-	-	62.7%	48.6%
Sig_May09-Jul09	-	-	-	68.6%

Table 5: Signature detection rate on current and *future* malware samples (3 months training)

Other Detection Results Table 5 shows that if we combine multiple signature sets, we can further increase the detection rate on new malware samples. For example, by combining the signatures extracted from the months of *Apr09*, *May09*, and *Jun09* (this signature set is referred to as *Sig_Apr09-Jun09* in Table 5), we can increase the "coverage" of the *Jun09* malware set from 58.9% (reported in Table 3) to 62.7%. Also, by testing the signature set *Sig_Apr09-Jun09* against the malware traffic from *Jul09* we obtained a detection rate of 48.6%, which is significantly higher than the 38.5% detection rate obtained using only the *Sig_Jun09* signature set (see Table 3). In addition, matching our largest set of signatures *Sig_May09-Jul09*, consisting of 2,973 distinct Snort rules, against one entire day of legitimate traffic (about 12 million HTTP queries) took less than one hour. This shows that our behavioral clustering and subsequent signature generation approach, though not a silver bullet, is a promising complement to other malware detection techniques, such as AV scanners, and can play an important role in a *defense-in-depth* strategy. This is also reflected in the results reported in Table 6, which represent the detection rate of our network signatures with respect to malware samples that were not detected by any of the three AV scanners available to us. For example, using the signature set *Sig_Feb09* , we are able to detect 54.8% of the malware collected in *Feb09* that were not detected by the AV scanners, 52.8% of the undetected (by AVs) samples collected in *Mar09*, 29.4% of the undetected samples collected in *Apr09*, etc.

	Feb09	*Mar09*	*Apr09*	*May09*	*Jun09*	*Jul09*
Sig_Feb09	54.8%	52.8%	29.4%	6.1%	3.6%	4.0%
Sig_Mar09	-	54.1%	20.6%	5.0%	3.1%	5.4%
Sig_Apr09	-	-	41.9%	5.8%	3.8%	5.2%
Sig_May09	-	-	-	66.7%	38.8%	16.1%
Sig_Jun09	-	-	-	-	48.9%	21.8%
Sig_Jul09	-	-	-	-	-	62.9%

Table 6: Detection rate on malware undetected by all AVs.

We can see from Table 6 that, apart from the signatures *Sig_Apr09*, all the other signature sets allow us to detect between roughly 20% and 53% of future (i.e., collected in the next month, compared to when the signatures were generated) malware samples that AV scanners were not able to detect. We believe the poor performance of *Sig_Apr09* is due to the lower number of distinct malware samples that we were able to collect in *Apr09*. As a consequence, in that month we did not have a large enough number of *training* samples from which to *learn* good signatures.

6.2.1 Real-World Deployment Experience

We had the opportunity to test our network signatures in a large enterprise network consisting of several thousands nodes that run a commercial host-based AV system. We monitored this enterprise's network traffic for a period of 4 days, from August 24 to August 28, 2009. We deployed our *Sig_Jun09* and *Sig_Jul09* HTTP signatures (using Snort) to monitor the traffic towards the enterprise's web proxy. Overall, our signature set consisted of 2,140 Snort rules. We used the first 2 days of monitoring for signature-pruning (see Section 4), and the remaining 2 days to measure the number of false positives of the pruned signature set. During the pruning period, using a web interface to Snort's logs, it was fairly easy to verify that 32 of our rules were actually causing false alerts. We then pruned (i.e., disabled) those rules and kept monitoring the logs for the next 2 days. In this 2-days testing period, overall the remaining signatures generated only 12 false alerts. During our 4 days monitoring, we also found that 4 of our network signatures detected actual malware behavior generated from 46 machines. In particular, we found that 25 machines were generating HTTP queries that matched a signature we extracted from two variants of `TR/Dldr.Agent.boey` [Avira]. By analyzing the payload of the HTTP requests we actually found that these infected machines seemed to be exfiltrating (POSTing) data to a notoriously spyware-related website. In addition, we found 19 machines that appeared to be infected by *rogue AV* software, one bot-infected machine that contacted its HTTP-based C&C server, and one machine that downloaded what appeared to be an update of `PWS-Banker.gen.dh.dldr` [McAfee].

6.2.2 Comparison with other approaches.

Table 7, third row, shows the *next month* detection rate (NMDR) for signatures generated by applying fine-grained clustering alone to each of our malware datasets. For example, given the malware dataset *Feb09*, we directly applied fine-grained clustering to the related malicious traffic traces, instead of applying our three-step clustering process. Then, we extracted a set of signatures from each fine-grained cluster, and we tested the ob-

tained signature set on the HTTP traces generated by executing the malware samples from the *Mar09* dataset. We repeated this process for all the other malware datasets (notice that the NMDR for *Jul09* is not defined in table Table 7, since we did not collect malware from August 2009). By comparing the results in Table 7 with Table 3, we can see that the signatures obtained by applying our three-step clustering process always yield a higher detection rate, compared to signatures generated by applying the fine-grained clustering algorithm alone.

	Feb09	*Mar09*	*Apr09*	*May09*	*Jun09*	*Jul09*
Clusters	2,934	2,352	1,485	2,805	2,719	3,343
Time	4h4min	1h52min	1h1min	3h9min	3h18min	2h57min
NMDR	38.4%	25.7%	24.2%	46.2%	36.3%	-

Table 7: Results obtained using only fine-grained clustering, instead of the three-step clustering process (NMDR = *next month* detection rate).

We also compared our approach to [10]. We randomly selected around four thousand samples from the *Feb09* and *May09* malware datasets. Precisely, we selected 2,038 samples from *Feb09* and 1,978 samples from *May09*. We then shared these samples with the authors of [10], who kindly agreed to provide the clustering results produced by their system. The results they were able to share with us were in the form of a similarity matrix for each of the malware datasets we sent them. We then applied single linkage hierarchical clustering to each of these similarity matrices to obtain the related dendrogram. In order to find where to cut the obtained dendrogram, we used two different strategies. First, we applied the DB index [17] to automatically find the best dendrogram cut. However, the results we obtained were not satisfactory in this case, because the clusters were too "tight". Only very few clusters contained more than one sample, thus yielding very specific signatures with a low detection rate. We then decided to select the threshold manually using our domain knowledge. This manual tuning process turned out to be very time-consuming, and therefore we finally decided to simply use the similarity threshold value $t = 0.7$, which was also used in [10]. A manual analysis confirmed that with this threshold we obtained much better results, compared to using the DB index.

We then extracted network signatures from the malware clusters obtained using both our three-step network-level clustering system, our fine-grained clustering only, and [10] (in all cases, we used the HTTP traffic traces collected using our malware analysis system to extract and test the network signatures). All clustering approaches were applied to the same reduced datasets described earlier. The results of our experiments are reported in Table 8. In the first row, "*Sig_Feb09 net-clusters*" indicates the dataset of signatures extracted from the (reduced) *Feb09* dataset using our three-step network-level clustering. "*Sig_Feb09 net-fg-clusters*" represents the set of signatures extracted using our fine-grained clustering only, while "*Sig_Feb09 sys-clusters*" indicates the signatures extracted from malware clusters obtained using a system-level clustering approach similar to [10]. We then tested the obtained signature sets on the traffic traces of the entire malware datasets collected in *Feb09* and *Mar09*. We repeated a similar process to obtain and test the "*Sig_May09 net-clusters*", "*Sig_May09 net-fg-clusters*", and "*Sig_May09 sys-clusters*" using malware from the *May09* dataset. From Table 8 we can see that the signatures obtained using our three-step clustering process yield a higher detection rate in all cases, compared to using only fine-grained clustering, and to signatures obtained using a clustering approach similar to [10].

	Feb09	*Mar09*	*May09*	*Jun09*
Sig_Feb09 net-clusters	78.6%	48.9%	-	-
Sig_Feb09 net-fg-clusters	60.1%	35.1%	-	-
Sig_Feb09 sys-clusters	56.9%	33.9%	-	-
Sig_May09 net-clusters	-	-	56.0%	44.3%
Sig_May09 net-fg-clusters	-	-	50.8%	42.5%
Sig_May09 sys-clusters	-	-	32.7%	32.0%

Table 8: Malware detection rate for network signatures generated using our three-step network-level clustering (*net-clusters*), only fine-grained network-level clustering (*net-fg-clusters*), and clusters generated using [10] (*sys-clusters*).

It is worth noting that while it may be possible to tune the similarity threshold t to improve the system-level clusters, our network-level system can automatically find the optimum dendrogram cut and yield accurate network-level malware signatures.

We also applied Polygraph [24] to a subset of (only) 49 malware samples from the `Virut` family. Polygraph ran for more than 2 entire weeks without completing. It is clear that Polygraph's greedy clustering algorithm is not suitable for the problem at hand, and that without the *preprocessing* provided by our clustering system generating network signatures to detect malware-related outbound HTTP traffic would be much more expensive.

6.2.3 Qualitative Analysis

In this section, we analyze some of the reasons why system-level clustering may perform worse than network-level clustering, as shown in Table 8.

In some cases, malware variants that generate the same malicious network traffic may generate significantly different system-level events. Consider the example in Figure 6, which reports information about the system and network events generated by two malware variants v_1, and v_2 (which are part of our *Feb09* dataset). v_1 is labeled as `Generic FakeAlert.h` by McAfee and as `TR/Dropper.Gen` by Avira (Trend did not detect it), whereas v_2 is labeled as `DR/PCK.Tdss.A.21` by Avira (neither McAfee nor Trend detected this sample). When executed, the first sample runs in the background and does not display any message to the user. On the

other hand, the second sample is a Trojan that presents the user with a window pretending to be the installation software for an application called *Aquaplay*. However, regardless of whether the user chooses to complete the installation or not, the malware starts running and generating HTTP traffic. The set of operations each variant performs on the system are significantly different (because of space limitations, Figure 6 only shows filesystem events), and therefore these two samples would tend to be separated by *system-level* behavioral clustering. However, the HTTP traffic they generate is exactly the same. Both v_1 and v_2 send the same amount of data to an IP address apparently located in Latvia, using the same two POST requests. It is clear that these two malware samples are related to each other, and our network-level clustering system correctly groups them together. We speculate that this is due to the fact that some malware authors try to spread their malicious code by infecting multiple different legitimate applications (e.g., different games) with the same *bot* code, for example, and then publishing the obtained *trojans* on the Internet (e.g., via peer-to-peer networks). When executed, each trojan may behave quite differently from a system point of view, since the original *legitimate* portions of their code are different. However, the *malicious* portions of their code will contact the same C&C.

Another factor to take into account is that malware developers often reuse code written by others and customize it to fit their needs. For example, they may reuse the malicious code used to compromise a system (e.g., the *rootkit* installation code) and replace some of the malicious code modules that provide network connectivity to a C&C server (e.g., to replace an IRC-based C&C communication with code that allows the malware to contact the C&C using the HTTP protocol). In this case, while the system-level activities of different malware may be very similar (because of a common *system infection* code base), their network traffic may look very different. In this case, grouping these malware in the same cluster may yield overly generic network signatures, which are prone to false positives and will likely be filtered out by the signature pruning process. Although it is difficult to measure how widespread such malware propagation strategies are, it is evident that system-level clustering may not always yield the desired results when the final objective is to extract network signatures.

7 Limitations and Future Work

Similarly to previous work that relies on executing malware samples to perform behavioral analysis [9, 10, 20], our analysis is limited to malware samples that perform some "interesting actions" (i.e., malicious activities) during the execution time T. Unfortunately, these interesting actions (both at the system and network level) may be triggered by events [11] such as a particular date, the way the user interacts with the infected machine, etc. In such cases, techniques similar to the ones proposed in [11] may be used to identify and activate such triggers. Trigger-based malware analysis is outside the scope of this paper, and is therefore left to future work.

Because we perform an analysis of the content of HTTP requests and responses, encryption represents our main limitation. Some malware writers may decide to use the HTTPS protocol, instead of HTTP. However, it is worth noting that using HTTPS may play against the malware itself, since many networks (in particular enterprise networks) may decide to allow only HTTPS traffic to/from certified servers. While some legitimate websites operate using *self-signed* public keys (e.g., to avoid CA signing costs), these cases can be handled by progressively building a whitelist of authorized self-signed public keys. However, we acknowledge this approach may be hard to implement in networks (e.g., ISP networks) where strict security policies may not be enforced.

Our signature pruning process (see Section 4) relies on testing malware signatures against a large dataset of *legitimate* traffic. However, collecting a completely *clean* traffic dataset may be difficult in practice. In turn, performing signature pruning using a *non-clean* traffic dataset may cause some malware signatures to be erroneously filtered out, thus decreasing our detection rate. There are a number of practical steps we can follow to mitigate this problem. First, since we are mostly interested in detecting new malware behavior, we can apply our signature pruning process over a dataset of traffic collected a few months before. The assumption is that this "old" traffic will not contain traces of *future* malware behavior, and therefore the related malware signatures extracted by our system will not be filtered out. On the other hand, we expect the majority of *legitimate* HTTP traffic to be fairly "stable", since the most popular Web sites and applications do not change very rapidly. Another approach we can use is to collect traffic from many different networks, and only filter out those signatures that generate false positives in the majority of these networks. The assumption here is that the same new malware behavior may not be present in the majority of the selected networks at the same time.

Evasion attacks, such as *noise-injection* attacks [28] and other similar attacks [25], may affect the results of our clustering system and network signatures. Because we run the malware in a protected environment, it may be possible to identify what HTTP requests are actually performed to send or receive information critical for the correct functioning of the malware using *dynamic taint analysis* [13]. This may allow us to correlate network traffic with system activities performed by the malware, and to identify whether the malware is injecting ran-

domly generated/selected elements into the network traffic. However, taint analysis may be evaded [12] and misled using sophisticated noise-injection attacks. System-level malware clustering (such as [9, 10]) and signature generation algorithms may also be affected by such attacks, e.g., by creating "noisy" system events that do not serve real malicious purposes, but simply try to mislead the clustering process and the generation of a good detection model. Noise injection attacks are a challenging research problem to be addressed in future work.

8 Conclusion

In this paper, we presented a *network-level* behavioral malware clustering system that focuses on HTTP-based malware and clusters malware samples based on a notion of structural similarity between the malicious HTTP traffic they generate. Through network-level analysis, our behavioral clustering system is able to unveil similarities among malware samples that may not be captured by current system-level behavioral clustering systems. Also, we proposed a new method for the analysis of malware clustering results. The output of our clustering system can be readily used as input for algorithms that automatically generate network signatures. Our experimental results on over 25,000 malware samples confirm the effectiveness of the proposed clustering system, and show that it can aid the process of automatically extracting network signatures for detecting HTTP traffic generated by malware-compromised machines.

Acknowledgements

We thank Paolo Milani Comparetti for providing the results that allowed us to compare our system to [10], Philip Levis for his help with the preparation of the final version of this paper, and Junjie Zhang and the anonymous reviewers for their constructive comments.

This material is based upon work supported in part by the National Science Foundation under grants no. 0716570 and 0831300, the Department of Homeland Security under contract no. FA8750-08-2-0141, the Office of Naval Research under grant no. N000140911042. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, the Department of Homeland Security, or the Office of Naval Research.

References

[1] Avira Anti-Virus. http://www.avira.com.
[2] Collaborative Malware Collection and Sensing. https://alliance.mwcollect.org.
[3] Koobface. http://blog.threatexpert.com/2008/12/koobface-leaves-victims-black-spot.html.
[4] McAfee Anti-Virus. http://www.mcafee.com.
[5] Project Malfease. http://malfease.oarci.net.
[6] Snort IDS. http://www.snort.org.
[7] Trend Micro Anti-Virus. http://www.trendmicro.com.
[8] Zeus Tracker. https://zeustracker.abuse.ch/faq.php.
[9] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario. Automated classification and analysis of internet malware. In *Recent Advances in Intrusion Detection*, 2007.
[10] U. Bayer, P. Milani Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *Network and Distributed System Security Symposium*, 2009.
[11] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. *Botnet Detection*, 2008.
[12] L. Cavallaro, P. Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *International conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2008.
[13] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *International symposium on Software testing and analysis*, 2007.
[14] D. Danchev. Web based botnet command and control kit 2.0, August 2008. http://ddanchev.blogspot.com/2008/08/web-based-botnet-command-and-control.html.
[15] G. Gu, R. Perdisci, J. Zhang, and W. Lee. Botminer: clustering analysis of network traffic for protocol- and structure-independent botnet detection. In *USENIX Security*, 2008.
[16] F. Guo, P. Ferrie, and T. Chiueh. A study of the packer problem and its solutions. In *Recent Advances in Intrusion Detection*, 2008.
[17] M. Halkidi, Y. Batistakis, and M. Vazirgiannis. On clustering validation techniques. *J. Intell. Inf. Syst.*, 17(2-3):107–145, 2001.
[18] A. K. Jain and R. C. Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., 1988.
[19] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Comput. Surv.*, 31(3):264–323, 1999.
[20] J. P. John, A. Moshchuk, S. D. Gribble, and A. Krishnamurthy. Studying spamming botnets using botlab. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.
[21] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. A. Kemmerer. Behavior-based spyware detection. In *USENIX Security*, 2006.
[22] V. Laurikari. TRE. http://www.laurikari.net/tre/.
[23] Z. Liand, M. Sanghi, Y. Chen, M. Kao, and B. Chavez. Hamsa: Fast signature generation for zero-day polymorphicworms with provable attack resilience. In *IEEE Symposium on Security and Privacy*, 2006.
[24] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *IEEE Symposium on Security and Privacy*, 2005.
[25] J. Newsome, B. Karp, and D. Song. Paragraph: Thwarting signature learning by training maliciously. In *Recent Advances in Intrusion Detection (RAID)*, 2006.
[26] J. Oberheide, E. Cooke, and F. Jahanian. CloudAV: N-Version antivirus in the network cloud. In *USENIX Security*, 2008.
[27] D. Pelleg and A. W. Moore. X-means: Extending k-means with efficient estimation of the number of clusters. In *International Conference on Machine Learning*, 2000.
[28] R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. Sharif. Misleadingworm signature generators using deliberate noise injection. In *IEEE Symposium on Security and Privacy*, 2006.
[29] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *ACM/USENIX Symposium on Operating System Design and Implementation*, December 2004.
[30] Symantec. *Symantec Global Internet Security Threat Report trends for 2008*, April 2009.
[31] S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *USENIX Technical Conference*, 1992.
[32] Y. Xie, F. Yu, K. Achan, R. Panigrahy, G. Hulten, and I. Osipkov. Spamming botnets: signatures and characteristics. In *ACM SIGCOMM conference on data communication*, 2008.
[33] V. Yegneswaran, J. T. Giffin, P. Barford, and S. Jha. An architecture for generating semantics-aware signatures. In *USENIX Security Symposium*, 2005.
[34] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *ACM Conference on Computer and Communications Security*, 2007.

Glasnost: Enabling End Users to Detect Traffic Differentiation

Marcel Dischinger
MPI-SWS

Massimiliano Marcon
MPI-SWS

Saikat Guha
MPI-SWS, Microsoft Research

Krishna P. Gummadi
MPI-SWS

Ratul Mahajan
Microsoft Research

Stefan Saroiu
Microsoft Research

Abstract

Holding residential ISPs to their contractual or legal obligations of “unlimited service” or “network neutrality” is hard because their traffic management policies are opaque to end users and governmental regulatory agencies. We have built and deployed Glasnost, a system that improves network transparency by enabling ordinary Internet users to detect whether their ISPs are differentiating between flows of specific applications. We identify three key challenges in designing such a system: (a) to attract many users, the system must have low barrier of use and generate results in a timely manner, (b) the results must be robust to measurement noise and avoid false accusations of differentiation, which can adversely affect ISPs’ reputation and business, (c) the system must include mechanisms to keep it up-to-date with the continuously changing differentiation policies of ISPs worldwide. We describe how Glasnost addresses each of these challenges. Glasnost has been operational for over a year. More than 350,000 users from over 5,800 ISPs worldwide have used Glasnost to detect differentiation, validating many of our design choices. We show how data from individual Glasnost users can be aggregated to provide regulators and monitors with useful information on ISP-wide deployment of various differentiation policies.

1 Introduction

A confluence of technical, business, and political interests has made “network neutrality” a hot button issue [18, 19]. The debate revolves around whether and to what extent Internet service providers (ISPs), who own and operate data networks, should be allowed to differentiate one class of traffic from another. Many ISPs want to restrict bandwidth-hungry applications that can hurt other applications in the network. Some also want to control applications such as VoIP that reduce ISPs’ ability to profit from competing services of their own. In contrast, many content providers are against traffic differentiation because it gives the ISPs arbitrary control over the quality of service experienced by users. In parallel, regulatory bodies and politicians are trying to devise policies that balance competing concerns [20, 21].

As this debate rages, ordinary Internet users are often in the dark, even though they are directly affected. The information sources available to users today are media reports, blogs, and statements made by ISPs; such information sources are imprecise at best and incorrect at worst. As a result, much traffic differentiation occurs without their knowledge. However, when ISPs traffic management practices come to light, user outrage forces regulatory bodies to conduct public hearings on prevalent practices [20, 21].

This situation led us to build and deploy a system, called Glasnost, that enables users to detect if they are subject to traffic differentiation. We make no judgment about whether traffic differentiation should be permitted by regulatory policy. Rather, our motivation is to make any differentiation along their paths *transparent* to users.

While other recent research efforts also aim to detect traffic differentiation [27, 31], Glasnost is unique in its focus on users. Instead of providing only a broad characterization of differentiation in the Internet, our goal is to let individual users determine if they experience differentiation and quantify its impact at the time they use our system.

Our focus is on enabling individuals who are not technically savvy. This creates design constraints that are typically not present in other measurement systems. First, the bar to using the system must be low. For instance, it is undesirable to require the installation of special software on client machines, especially if such software needs privileged access. This constraint hinders our ability to collect high-fidelity data (e.g., packet traces) or to finely control packet transmissions. We must limit ourselves to coarse-grained data obtained through unprivileged client operations. Second, the results for an individual user must be accurate and simple to interpret. For example, we cannot return results that

rely on inferences derived from data aggregated across users. While such results are accurate on aggregate, they could be incorrect when applied to an individual user. Third, the system must evolve with ISP practices. Otherwise, Glasnost would gradually become unable to detect the presence of differentiation and users would stop trusting the system.

We based our Glasnost design on these constraints. The result is a system that is effective and easy to use. A user can detect differentiation by simply pointing her browser to a Web page. The browser downloads and runs a Java applet which exchanges traffic with our measurement server. The client-server nature of our architecture helps to avoid many of the operational issues with network measurements, such as traversing NATs and firewalls, or raising alarms in network intrusion detection systems. The traffic exchange is designed to accurately and quickly detect any differentiation. We also build a simple flow emulation tool that simplifies the incorporation of tests to detect new differentiation techniques that emerge in the Internet.

The diversity of ISP practices makes it challenging to detect traffic differentiation reliably. For instance, an ISP might employ differentiation only at specific times (e.g., in the evenings), or only under high loads, or only for flows that send too much traffic. These factors led us to design an on-demand system. Each time a user uses Glasnost, she performs an *individual* test that detects the presence of traffic differentiation for her Internet connection *at the time of the test*. This provides a more reliable answer to this user than extrapolating the results from other testing times or other users.

Glasnost has been operational since March 2008, enabling users to detect BitTorrent differentiation. Between March 2008 and September 2009, more than 350,000 users from over 5,800 ISPs worldwide have used the system. Several individuals and corporations volunteered to host Glasnost measurement servers on their own infrastructure in order to allow operations on an even larger scale. We believe that our design principles have directly contributed to the success of Glasnost.

In addition to the design and evaluation of Glasnost, we also present a detailed analysis of BitTorrent differentiation in the Internet. We find that about 10% of our users experience differentiation of BitTorrent traffic. We also study ISPs' BitTorrent differentiation policies in detail over a period of two months (from January to February 2009) using data from the Glasnost tests. We find, for instance, that it is more common for ISPs to differentiate against file uploads than downloads and to differentiate throughout the day rather than only during peak hours.

2 Traffic Differentiation

Traffic differentiation refers to an ISP treating the packets of one flow differently than those of another flow. Based on information published by ISPs, researchers, and equipment vendors [5, 10, 22], we characterize traffic differentiation along three dimensions.

1. Traffic differentiation based on flow types. To differentiate between flows of different types, i.e., belonging to different applications, ISPs must distinguish the packets of one flow from those of other flows. This can be done by examining one of the following:

(a) *The IP header.* The source or destination addresses can determine how an ISP treats a flow. For example, universities routinely rate-limit only traffic that's going to or coming from their student dorms.
(b) *The transport protocol header.* ISPs can use port numbers or other transport protocol identifiers to determine a flow's treatment. For example, P2P traffic is sometimes identified based on its port numbers.
(c) *The packet payload.* ISPs can use deep-packet inspection (DPI) to identify the application generating a packet. For example, ISPs look for P2P protocol messages in packet payload to rate-limit the traffic of P2P applications, such as BitTorrent.

2. Traffic differentiation independent of flow type. In addition to features of a flow itself, an ISP may use other criteria to determine whether to differentiate. Some of these include:

(a) *Time of day.* An ISP may differentiate only during peak hours.
(b) *Network load.* An ISP may differentiate on a link only when the network load on that link is high.
(c) *User behavior.* An ISP may differentiate only against users with heavy bandwidth usage.

3. Traffic manipulation mechanisms. There are a number of ways in which an ISP can treat one class of packets differently.

(a) *Blocking.* One form of differentiation is to terminate a flow, either by blocking its packets or by injecting a connection termination message (e.g., sending a TCP FIN or TCP RST packet).
(b) *Deprioritizing.* Routers can use multiple priority queues when forwarding packets. ISPs can use this mechanism to assign differentiated flows to lower priority queues and to limit the throughput of certain classes.

(c) *Packet dropping.* Packets of a flow can be dropped either using a fixed or variable drop rate.

(d) *Modifying TCP advertised window size.* ISPs can lower the advertised window size of a TCP flow, prompting a sender to slow down.

(e) *Application-level mechanisms.* ISPs can control an application's behavior by modifying its protocol messages. For example, transparent proxies [28] can redirect HTTP or P2P flows to alternate content servers.

What kinds of traffic differentiation does Glasnost detect?
Our current implementation of Glasnost detects traffic differentiation that is triggered by transport protocol headers (e.g., ports) or packet payload. These triggers are more common than IP headers [1, 5].

We designed Glasnost to be an on-demand system. Each time a user uses Glasnost, we detect traffic differentiation between flows of the user at the time of the test. While Glasnost has not been designed detect traffic shaping that affects all flows of a user, e.g., based on time of day or network load or user behavior, it is possible to infer such shaping policies by aggregating and comparing the results of Glasnost tests conducted at different times of the day by different users on different networks.

Instead of inferring differentiation based on a particular manipulation mechanism, Glasnost detects the presence of differentiation based on its impact on application performance.

3 Design Principles

In the process of developing Glasnost we identified several key design principles. Although in Glasnost our focus is traffic differentiation, the design principles we identified are more general and apply to many measurement systems that want to attract a large number of users. In this section, we discuss these principles in detail and argue why they are generally useful when designing measurement systems for Internet users at large.

Our goal was to build a system that lets ordinary Internet users determine if they are affected by traffic differentiation. Because of its focus on end users and the nature of its measurements, Glasnost must satisfy certain design requirements that are typically not present in other measurement systems. We distill these requirements into three design principles. These principles dictate that the system must be easy to use so that it can serve any Internet user, its inferences must be robust and simple to interpret, and it must be extensible to allow detection of new network policies as they evolve.

We explain these principles in detail below and also describe the consequences they have on the design of Glasnost. These consequences motivate certain design choices and rule out many others.

Principle #1: Low barrier of use
Attracting a large number of users to a measurement system requires having a low barrier of use. Although this challenge appears obvious, solving it is the key to success. As we discuss later, it complicated the design of other aspects of the system. But at each step we resisted the temptation to compromise in the interest of other desirables such as efficiency and higher-fidelity data.

Design consequences. There are four design consequences of this principle. First, because most users are not technically savvy, the interface must be simple and intuitive. Second, we cannot require users to install new software or perform administrative tasks. Many network measurement techniques require installing drivers (e.g., the WinPcap library for Windows) or running privileged code (e.g., raw sockets) on users' machines. Such code can provide detailed, low-level data (e.g., packet traces) that simplifies the measurement task. But in our experience, users are often unwilling to use systems with such requirements. For example, one of our earlier attempts required users to run code with administrator privileges on their machines and to leave a port open in their firewalls and NATs. These obstacles greatly limited adoption; we attracted fewer than fifty users. Third, because many users have little patience, the system must complete its measurements quickly. Fourth, to incentivize users to use the system in the first place, the system should display per-user results immediately after completing the measurements.

In order to satisfy above the requirements, our current client-side implementation uses a small-size Java applet (21 KBytes) that users download off our webpage. The applet exchanges traffic with our servers, which we then analyze to detect differentiation (we explain the nature of this traffic below). The test runs for about 6 minutes. Immediately after the test is finished Glasnost whether the user is affected by traffic differentiation.

Our quick and simple test methodology is inspired by non-research-oriented web sites for broadband speed tests [2] and represents a departure from other research systems. For instance, Scriptroute [25] requires users to write their own measurement scripts, and thus its use has been limited to researchers and other experts.

Principle #2: Measurement accountability
Because the system is designed for ordinary users, it is essential that the measurements are accurate and that the results cannot be misinterpreted. For instance, consider the results of an experiment to infer path capacities in

the Internet. Since the measurements can be affected by transient noise, researchers will know that the answer computed along an individual path cannot be trusted but the answers can be aggregated to provide an accurate estimate of path capacity. But an ordinary user that is interested in the capacity of her own path might not be in a position to make that distinction.

When detecting traffic differentiation, accurate interpretation of results is critical due to the controversial nature of traffic management in the Internet: there is still a heated debate whether it is legal for an ISP to employ traffic management. In addition, if people were to falsely interpret results as their ISP performing traffic differentiation when in fact it is not, the system would quickly lose credibility. In fact, in the past there have been instances when some widely publicized studies have mistakenly accused ISPs of using policies they never deployed [26, 29].

Design consequences. Maintaining measurement accountability has three design consequences. First, the test to detect differentiation should, to the extent possible, marginalize any factors that add uncertainty. The performance of an Internet flow can be affected by many confounding factors. This includes the operating system, especially its networking stack and its configuration. Additionally, directly using application client software is problematic as it does not give full control over the measurement traffic. Short-term throughput of such "natural" flows can differ because of differences in packet sizes and burstiness. Finally, we have to consider transient noise, as, e.g., caused by background traffic.

With passive measurement tools, it is often not easy to isolate these factors. These tools must take into account for a large number of confounding factors in their inference. The complexity of this analysis can lead to inaccurate results. In contrast, active measurements can be designed to avoid most confounding factors. Having full control over the traffic that is sent to measure performance simplifies the analysis. Further, active measurements allow to run all measurements between the same pair of hosts, removing factors like OS and networking stack. The only remaining confounding factor is transient noise, which can be dealt with using simple techniques such as repeating measurements multiple times.

Second, because not all uncertainty can be removed from the inference, the result presented to the user must be conservative, with a near-zero false positive rate. In the context of traffic differentiation, a false positive means that the system falsely claims that the user is experiencing traffic differentiation. Minimizing false positives is challenging because it results in an increase in the false negative rate. This trade-off is inherent.

Because of the concerns above, our testing primitive is based on comparing the throughput of a pair of flows. One flow in the pair belongs to the potential victim application. The second is a reference flow that belongs to a different application. The flows are identical except for the trigger that we want to test for differentiation, such as port number or payload. The flows are generated back-to-back and multiple pairs are run to reduce and calibrate the effect of noise.

Third, we must be prepared to provide the data and the evidence behind our inferences when requested. We retain the data of all measurements in which Glasnost detects traffic differentiation. We treat this data as evidence. If we are challenged to justify our findings, the stored data will help us explain on what basis Glasnost declared that an ISP is using traffic differentiation.

Principle #3: Easy to evolve

To remain relevant, a system that wants to detect traffic differentiation must be able to evolve as ISPs evolve their traffic management policies. For example, in Fall 2008, Comcast blocked BitTorrent uploads for some of its customers [10]. Several months later, they started replacing this practice with less severe forms of differentiation [7]. In fact, our recent measurements indicate that BitTorrent traffic blocking is rare today unlike in 2008. A system with a fixed set of capabilities will have a limited shelf life in such an evolving environment.

Design consequences. This principle mandates incorporation of mechanisms that help the system evolve with the network. Network evolution may be incidental or adversarial. In an incidental evolution, ISPs might target new applications in the future or use new traffic manipulation mechanisms. A detection system should be extensible, to add tests that detect traffic differentiation against popular new applications or based on new shaping techniques. Glasnost enables advanced users to submit packet-level traces of applications that they suspect are being targeted by their ISPs. User suspicion is powerful; it was how many of the currently known ISP differentiation behaviors came to light. We do not expect all users to be able to submit traces but there are many enthusiastic users that are capable of collecting (with our help if needed) and sharing traces. Glasnost then makes it easy to use these network traces to construct new detection tests. These tests help us keep pace with new traffic differentiation techniques and applications that may be targeted.

Adversarialy, ISPs could begin whitelisting traffic from measurement servers in an attempt to evade detection. A successful system must be aware of this problem and find ways to minimize whitelisting. Our solution was to make our server code publicly available. Anyone can setup Glasnost on a well-provisioned server and

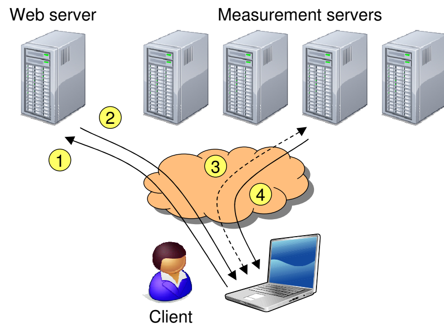


Figure 1: The Glasnost system. (1) The client contacts the Glasnost webpage. (2) The webpage returns the address of a measurement server. (3) The client connects to the measurement server and loads a Java applet. The applet then starts to emulate a sequence of flows. (4) After the test is done, the collected data is analyzed and a results page is displayed to the client.

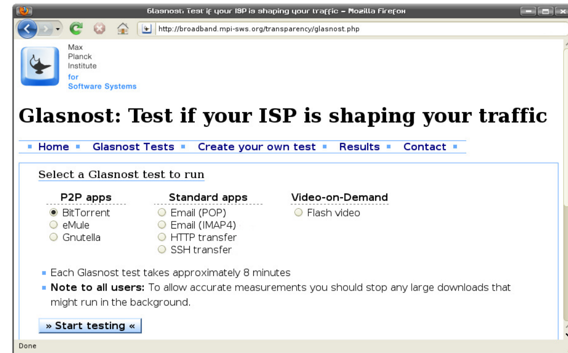


Figure 2: The Glasnost web interface.

other users can start measuring to new servers. Making our code publicly available allowed other Glasnost servers to appear on the Internet, which makes it hard for ISPs to evade detection. However, this method is not foolproof; a determined ISP may choose to stay up-to-date with the list of Glasnost servers. We doubt that many ISPs would be willing to invest significant effort in evading detection. As much as an ISP would like to conceal its traffic management practices from the public, denying those practices or making blatant attempts to hide them is risky. Such behavior, if detected, would attract intense scrutiny from telecom regulators and would severely damage the ISP's reputation. For example, when Comcast's BitTorrent blocking practices were revealed to the public [1], Comcast was fined by the FCC and was subjected to highly critical media coverage.

4 Design of Glasnost

We now present the design of Glasnost based on the requirements outlined above.

4.1 System architecture

Glasnost is based on a client-server architecture. Clients connect to a Glasnost server to download and run various tests. Each test measures the path between the client and the server by generating flows that carry application-level data. This data is carefully constructed to detect traffic differentiation along the path.

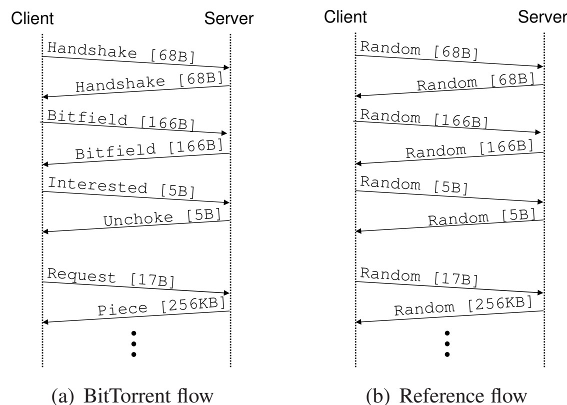
Figure 1 presents a high-level description of how clients measure their Internet paths. A client first contacts a central webpage that redirects to a Glasnost measurement server. This dynamic redirection enables load balancing across measurement servers and makes it easy to incorporate new servers by adding them to the redirection list.

After the client is redirected, the measurement server presents a simple interface to the user. As shown in Figure 2, the user selects the application traffic she would like to test and starts the test by just clicking the "Start testing" button. The client's browser downloads a Java applet that starts exchanging packets with the server. We elaborate on the Glasnost measurement tests next.

4.2 Measurement tests

The key primitive behind the Glasnost measurement tests is the *emulation* of a pair of flows that are identical except in one respect that we suspect triggers differentiation along the path. Comparing the performance of these flows helps to determine if differentiation is indeed present.

Figure 3 shows two flows designed to detect whether differentiation based on BitTorrent protocol content is present along a path. The exchange on the left corresponds to the first flow. The client opens a TCP connection to the measurement server and starts exchanging packets that implement the BitTorrent protocol: the packet payloads carry BitTorrent protocol headers and content. The exchange on the right corresponds to the second flow. The client opens another TCP connection and performs the same packet exchange, but the packets contain random bytes instead of BitTorrent headers or data. An ISP that differentiates against BitTorrent based on protocol messages would impact only the first flow. Thus, significant differences in the flows' performance is likely to be caused by the differences in their payloads and lets us detect whether differentiation is present along the path. Transient noise can also lead to differences in flows' performance; we describe in the next section how we handle noise.



(a) BitTorrent flow (b) Reference flow

Figure 3: A pair of flows used in Glasnost tests. The two flows are identical in all aspects other than their packet payloads, which allows us to detect differentiation that targets flows based on their packet contents.

During the test, the measurement server records a packet-level trace of all emulated flows and the client applet records ancillary information including exceptions caused by network errors. Once the transfers end, the client uploads the recorded information to the server. The server analyzes this information together with the traces collected on the server-side and shows the findings to the client.

Glasnost's emulation methodology leads to measurement robustness. As Figure 3 shows, application-level data is the only difference between the two emulated flows. The two flows traverse the same network path and have the same network-level characteristics, such as port numbers, packet sizes, etc. In contrast, passive measurement, a different technique, may have many factors differ across these measured flows. Correctly accounting for all such differences is challenging.

Another benefit of active measurement is the ability to carefully control the measurement test. For example, we can repeat flows with different payloads or port numbers. This ability allows Glasnost to precisely identify the specific factors that trigger differentiation.

In the next section, we describe our measurement test in more detail and how we make it robust to transient noise. We describe how we make the system easy to evolve using a trace replay based tool for constructing measurement tests in Section 6.

5 Robust Detection of Differentiation

As described earlier, Glasnost emulates a pair of flows and determines the presence of traffic differentiation by comparing their performance. When comparing the performance of a pair of flows, we must ensure that their difference is indeed due to the differences in their content and not due to some changes in the test environment. Our measurement tests are constructed in a way that eliminates all major confounding factors except one – transient noise due to interference from cross-traffic (background traffic) along the measurement path. In this section, we discuss techniques to robustly detect traffic differentiation in the face of transient noise.

The primary challenge in this task stems from the fact that the noise can vary at small time-scales. Thus, two flows can be affected differently even if run back-to-back. As one egregious example, we found that the throughput of two back-to-back flows differed by a factor of three even though the flows were identical. A simplistic detection method will mistakenly detect differentiation in this case. It might appear that the differential impact of noise could be reduced by running the flows simultaneously. But we find that setup to be even worse because of self-interference among the two flows.

Our basic strategy for robust detection is to run each flow type multiple times. We use the variance in the performance of the flows of the same type to identify paths that are too noisy to enable reliable detection. For the remaining paths, we can then detect differentiation by comparing the flows of different types. We first describe how we apply this strategy when tests are run long enough that we do not have to worry about having too little data. As we found that many users are too impatient to run long tests, we adapted our strategy to tests that run for a shorter duration.

We describe our method using throughput as the measure of flow performance[1], since it is of prime interest to many applications and is the target of many ISPs looking to reduce their network load. Because of TCP dynamics, throughput is directly affected by any differentiation that impacts flow latency or loss.

5.1 Filtering tests affected by noise

To detect the level of transient noise, we repeat the runs of the two flow types multiple times back-to-back. Unlike active ISP differentiation, transient noise does not discriminate based on flow content; it would not affect multiple runs of the same flow type and thus can be detected by comparing their performance.

To understand transient noise patterns and the extent to which they affect flow throughput, we configured our Glasnost deployment to run a BitTorrent flow and a reference flow with random bytes, five times each. The runs of the two flow types were interspersed and each flow lasted for 60 seconds to allow sufficient time for TCP to achieve stable throughput. Over a period of one

[1] Our method can be extended to other measures of performance such as jitter.

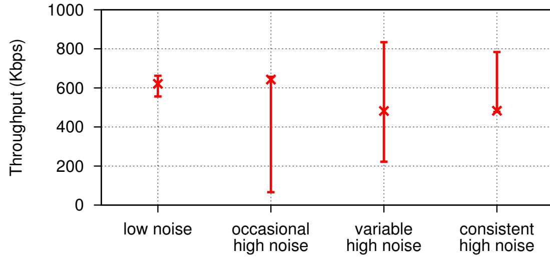


Figure 4: The four classes of noise we observed in our analysis. The graph shows the minimum, median, and maximum throughputs observed in example tests affected by each class of noise.

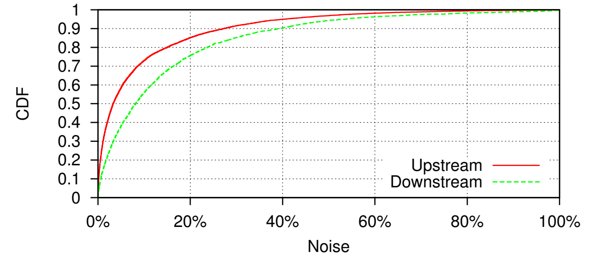


Figure 5: Noise observed in our 3,705 sample dataset. 85.2% of upstream flows and 75.7% of downstream flows have less than 20% of noise. Noise is measured as the difference between maximum and median throughput calculated as a percentage of maximum throughput.

month, we collected measurements of 3,705 residential broadband hosts, 2,871 in the upstream and 834 in the downstream direction.

We compared the throughput obtained by the five runs of each flow type with each other. Our analysis of the maximum, median, and minimum throughput reveals the four distinct patterns shown in Figure 4, corresponding to four different cross-traffic levels:

1. **Consistently low cross-traffic:** all throughput measurements belonging to the same flow type fall within a narrow range (i.e., min is close to max).
2. **Mostly low but occasionally high cross-traffic:** a majority of throughput measurements are clustered around the maximum but a few points are farther away (i.e., max and min are far apart but median is close to max).
3. **Highly variable cross-traffic:** the throughput measurements are scattered over a wide range (i.e., max and min are far apart and median is far apart from both).
4. **Mostly high but occasionally low cross-traffic:** a majority of throughput measurements are clustered around the minimum but a few measurements are farther away (i.e., max and min are far apart but median is close to min).

Our categorization of the level of cross-traffic in each case is based on two key observations about the nature and impact of cross-traffic. First, cross-traffic only lowers throughput and never improves it. Thus, when a majority of throughput measurements are close to min but far apart from max (as in category 4 above), it is more likely that the noise-free throughput is closer to max than min.

Second, cross-traffic is unlikely to be consistently high over a long period of time. In theory, measurements in category 1 above could be explained by consistently high cross-traffic. But, this would require the cross-traffic to remain high and consistent (without changing) over the duration of the entire experiment, which is ten minutes. We believe that this is unlikely.

For robust detection of differentiation, we discard all tests where a majority of flows are affected by high noise (i.e., categories 3 and 4). For these tests, we cannot determine whether the difference in throughput is caused by differentiation or transient noise. We analyze only the remaining tests, for which a majority of runs experience low noise (i.e., categories 1 and 2).

To help determine which tests belong to the predominantly low noise category, we plot the difference between maximum and median throughput as a percentage of maximum throughput in Figure 5. We found that for a large majority of tests (85.2% of upstream tests and 75.7% of downstream tests) the median throughput is within 20% of the maximum. The difference between median and maximum throughput is considerably larger for the remaining flows. We thus use the 20% difference between median and max throughputs as a threshold to discard tests that are significantly affected by noise. Next, we describe how we detect traffic differentiation within the remaining tests.

5.2 Detecting differentiation in low-noise tests

To detect traffic differentiation among tests that are identified as low noise, we compare the maximum througput of each flow type. Our decision to use the maximum is based on the observations that (a) in low-noise cases, most measurements lie close to the maximum throughput and (b) because noise tends to lower throughput, the maximum throughput is a good approximation for what the flows would achieve without cross-traffic.

We infer that the two flow types are being treated differently if the maximum throughput of one differs from

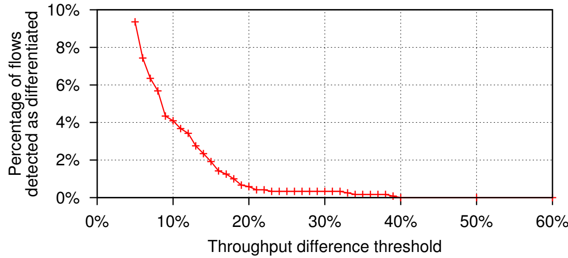


Figure 6: Selecting a good throughput difference threshold. Thresholds smaller than 20% tend to produce a significant number of false positives.

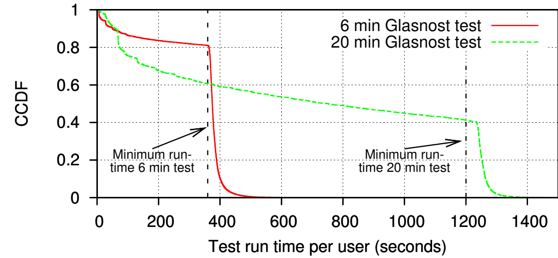


Figure 7: Duration users run the Glasnost test. Longer duration tests are aborted by a larger fraction of users.

that of the other by more than a threshold δ. Selecting a good δ involves a trade-off. With high values, we cannot detect differentiation unless the impact on throughput is high. For instance, with δ=50%, we would only detection differentiation that halves the flow throughput. Thus, high values raise the false negative rate. On the other hand, with low values of δ (say 5%), we risk false positives, i.e., declaring that ISPs are employing traffic differentiation while they actually do not.

To understand how the false positive rate varies with δ, we selected 302 test runs from users from ISPs that we know do not differentiate. Figure 6 plots the percentage of tests that are falsely marked as being differentiated for different threshold values. The plot shows an interesting trend; the false positive rate drops steeply until δ reaches 20%. Beyond this threshold, there are a handful of hosts (0.58%) that pass our noise tests but are still falsely marked as differentiated. To avoid any false positives, we would need to raise the threshold to 40%, which increases the false negative rate.

We thus set δ to 20%. With this value we maintain a low false positive rate (under 0.6%), but we fail to detect differentiation that reduces a flow's throughput by less than 20%. We consider this an acceptable trade-off.

5.3 User impatience with long tests

As described above, we configured Glasnost to run a pair of one-minute-long flows five times, resulting in a total test time of 10 minutes. The tests we originally deployed also detected whether the differentiation was based on port number or payload, extending the test duration to 20 minutes. While this test configuration enables us to detect differentiation with high confidence, we noticed that a considerable fraction of users were aborting the tests before completion.

Figure 7 shows how long users keep their Glasnost test running. The plot for 20 minute long tests shows an alarming decline in the percentage of users as the test progresses. Only 40% of the users stay till the end and nearly 50% aborted their tests within the first 10 minutes. The sudden drop near the 20 minute point corresponds to successfully completed tests.

Our results show that users are impatient. Most are not willing to use tests that take more than a few minutes. To confirm this, we reconfigured Glasnost to use shorter-duration tests. We reduced the number of times we repeat each flow type to two (from five), and we decreased the duration of each flow to 20 seconds (from 60 seconds), which is still sufficient for TCP to exit slow-start and achieve stable throughput. We bundled the tests for both upstream and downstream directions, and the resulting test takes 5.33 or roughly 6 minutes.

Figure 7 shows also how long users keep the 6 minute Glasnost test running. More than 80% of the users stay till the end, confirming that shorter tests on the order of a few minutes are more effective at retaining users.

5.4 Detecting differentiation with short tests

Short duration tests are challenging for detecting differentiation robustly because they gather few measurement samples. To estimate the impact of this reduction in data on detection accuracy, we consider data from the longer tests for which we have a result, i.e., for which we know whether or not the ISP is differentiating. We prune the data to include only what would be gathered by the short test and run our analysis on the pruned data. We compare the results from this shorter test data with those obtained before.

We find that nearly 25% of the long tests that we were able to successfully analyze before, were discarded as too noisy after pruning. We find the false positive rate (i.e., cases when the long test found no traffic differentiation but the short test did) to be 2.8% and the false negative rate (i.e., cases when the long test found traffic differentiation but the short test did not) to be 0.9%.

We also find that we can achieve a four-fold reduction in false positive rate, to 0.7% (which is comparable

to the false positive rate of long tests), by raising the δ threshold from 20% to 50%. While this increases the false negative rate to 1.7%, we consider it an acceptable trade-off.

6 Facilitating New Test Construction

Manually implementing Glasnost tests for a new application is a laborious and error prone task. It requires detailed knowledge of the application's protocols and their common implementations. This creates a high barrier for new test construction, making it difficult to keep pace with the evolution of ISPs' policies.

In this section, we present a tool called `trace-emulate` that simplifies the construction of new tests by automating most of the process. We also present a validation of the tests constructed by `trace-emulate` using the open source DPI engine of a commercial traffic shaper [22].

Our `trace-emulate` tool automatically generates a new Glasnost test from the packet-level trace of an application. It extracts the essential characteristics of the application flows. These include packet sizes and payloads as well as the order of packets with protocol messages and the inter-packet timing.

The test configuration that `trace-emulate` outputs is then used by the Glasnost Java applet to run the test. When run against the server, the applet exchanges two flows. The first flow has the same characteristics as the original trace. For example, assume that in the original trace the client performed the following operations: (1) sent packet A, (2) received packet B, (3) sent packet C after t seconds. These operations occur in the same order and relative times in the generate flow. In some cases, simultaneously preserving packet ordering and inter-packet timing is impossible. Such cases arise when an endpoint is waiting to receive a packet that gets delayed in the network. We make the endpoint (client or the server) wait until the packet is received before continuing the emulation, even though it increases the inter-packet time. Our decision to preserve ordering at the expense of timing is motivated by the observation that ISPs often use the sequence of protocol messages to identify applications, rather than their relative timing. The second flow exchanged by the applet is a reference flow with the same characteristics but uses different payloads and ports. The user uploading the trace can set the ports to specific values, e.g., the application's default port; otherwise, random ports are used.

Our experiments confirm that the replaying method of `trace-emulate` produces the same packet sizes, payloads, and ordering as the original trace. We omit detailed results.

Validating tests generated by trace-emulate. We validate that `trace-emulate` captures the essential characteristics that an ISP might use to identify an application flow in practice. While ISPs can, in theory, use arbitrarily complex mechanisms, in practice they are limited to using mechanisms that can scale to at least multiple Gbps. We are therefore interested in validating `trace-emulate` against *practical* detection mechanisms used by ISPs.

As one might imagine, ISPs use traffic classification solutions from third-party vendors such as Sandvine, BlueCoat, and Arbor Networks; most ISPs do not build their own system. Fortunately, pressure from privacy watchdogs compelled one of these vendors – Ipoque – to release the code it uses to inspect user traffic [22]. This release gives the research community, for the first time, access to production code that ISPs use to detect the application that a user is running.

The Ipoque code allows us to realistically validate `trace-emulate`. By inspecting the code we discover what applications are detected. We run the application and check whether the Ipoque detector detects the application from the packet flow. We then use `trace-emulate` to generate a Glasnost test for that application. We run the test and check whether the result is the same. If Ipoque detects our emulated flow as the target application, then we have successfully captured the essential characteristics that are necessary for detection by a commercial traffic classifier.

Ipoque's detector can identify traffic from more than 90 widely-used applications broadly classified as peer-to-peer, video streaming, instant messaging, online gaming, and other applications (email, web, etc.). It took us less than two hours to generate Glasnost tests for 10 representative applications in all five of the above categories. This included eMule, Gnutella, and BitTorrent (all P2P); YouTube (streaming video); World of Warcraft (online game); IRC (instant messaging); and HTTP, FTP and IMAP. For eMule and Gnutella, Ipoque separately identifies their control and data connections; consequently, we used `trace-emulate` to generate the corresponding two tests. That we were able to generate all tests in a matter of hours is a testament to the simplicity of `trace-emulate`.

In every single case, Ipoque identified the test generated by `trace-emulate` as the target application. To the extent Ipoque is representative of other similar vendors, we can claim that `trace-emulate` captures the essential flow characteristics for applications that do not encrypt traffic. However, without knowledge of how Ipoque detects applications from encrypted traffic, we cannot make any claims in that regard.

To convince ourselves that the Ipoque result holds in the real-world, we further validated `trace-emulate`

Application	Port-based	Content-based
BitTorrent	6881, down	down
eMule data	4662, down	down
Gnutella control	6346, down+up	down+up
Gnutella data	6346, down+up	down
HTTP	no	no
IMAP	no	no
SSH	no	no

Table 1: Results from running new Glasnost tests on a host connected via Kabel Deutschland. We identified instances of port-based and content-based traffic differentiation both the downstream (down) and upstream (up) directions.

against Kabel Deutschland, the biggest cable ISP in Germany. Kabel Deutschland targets P2P filesharing applications between 6pm and midnight [12]; their chosen vendor for traffic shaping equipment is unknown. In any event, since we know their policy, validating `trace-emulate` is straightforward. We run tests we generated for BitTorrent, eMule, Gnutella, HTTP, IMAP, and SSH from a Kabel Deutschland user, and check if Glasnost detects traffic differentiation.

Glasnost detected traffic differentiation for *each* of the P2P applications, and *none* of the non-P2P applications. In fact, by running the tests in both directions (downstream and upstream) and using different ports (default application port, random port), we were able to refine the policy published by Kabel Deutschland. Table 1 shows that P2P traffic is differentiated regardless of the port number used (i.e., based on the packet content). Next, we ran the HTTP, IMAP, and SSH tests on the ports typically used by the three P2P applications and found the flows achieved significant lower throughput. Running the same tests on random ports resulted in normal throughput. This is precisely what one might expect if Kabel Deutschland additionally uses port-based detection, which naturally has false-positives. Regardless of whether Kabel Deutschland sought to omit mention of side effects of their differentiation policy or we have identified a misconfiguration, our finding demonstrates the value of network transparency tools such as Glasnost.

7 Deployment Experiences

We deployed Glasnost publicly on the Internet on March 18th, 2008 and it has been operational ever since. It can be accessed at *http://broadband.mpi-sws.org/transparency/glasnost.php*. Initially, Glasnost was deployed on eight servers at MPI-SWS. Over the last year, the number of servers has grown to eighteen with the use of Measurement Lab (M-Lab) [16], an open platform for the deployment of Internet measurement tools to enhance network transparency. Eleven servers are in Europe, three on the west coast of the USA, and four on the east coast of the USA.

In the beginning, we chose to focus on one application as we developed our system and refined its techniques. We picked BitTorrent because it is widely suspected of being manipulated by ISPs [15]. However, our differentiation detection techniques are not specific to BitTorrent and can be applied to other applications as well. Because we have only recently deployed tests for other applications, an overwhelming majority of our data is from BitTorrent. We thus limit most of the discussion below to BitTorrent.

Details of deployed tests. In this paper, we present results for four BitTorrent tests deployed on Glasnost. These tests detect port- and content-based differentiation in the upstream as well as the downstream direction. Each test involves emulating BitTorrent and reference flows. For detecting content-based differentiation, we replace BitTorrent packet payloads with random bytes in the reference flows, while keeping other aspects identical. For detecting port-based differentiation, only the port of the reference flow is switched from a well-known BitTorrent port (e.g., 6881) to a neutral port that is not associated with any particular application (e.g., 10009). We emulate flows in both upstream and downstream directions to check for manipulation of both BitTorrent uploads and downloads. As described in Section 5, we configured Glasnost to offer a 6-minute long test to users with each flow running for 20 seconds. Also, each flow type is repeated once.

Usage. Between March 18th, 2008 and September 21st, 2009, 368,815 users[2] from 5,846 ISPs used Glasnost to test for traffic differentiation. We believe that our large user base is a result of our focus on lowering the barrier of use such that even lay users can use our system.

Figure 8 shows that our users have a wide geographical footprint. They come from North America (38%), Europe (36%), South America (11%), Asia (12%), Oceania (3%), and Africa (<1%).

Table 2 lists the top 20 access ISPs to which our users belonged. Users' IP addresses are mapped to ISPs using *whois* information from the Regional Internet Registries. We see that a large fraction of our users are from some of the largest residential ISPs in their respective countries, such as Comcast in the USA, Bell Canada in Canada, or BT in the UK.

[2]In this section, we use the terms tests, IP addresses, and users interchangeably. There are very few IP addresses from which we saw repeat tests and a vast majority of tests correspond to an unique IP address. The same end user may be associated with different IP addresses during the course of our study. By overlooking this, we may be over-counting the number of unique end users.

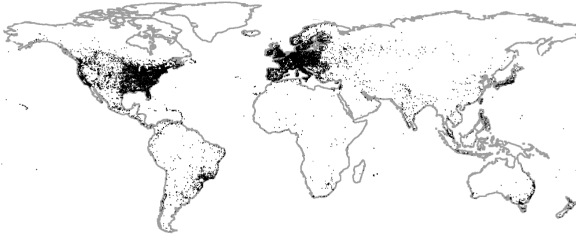


Figure 8: Location of Glasnost users.

ISP	Tests	ISP	Tests
Comcast (US)	29,464	BT (UK)	5,192
RoadRunner (US)	16,257	Chunghwa T. (TW)	5,084
AT&T (US)	10,884	Shaw (CA)	4,933
UPC (NL)	8,871	Brasil Telec. (BR)	4,862
Verizon (US)	7,611	Rogers (CA)	4,499
Cox (US)	4,194	Telefonica (BR)	4,408
Net Virtua (BR)	7,207	Telefonica (ES)	4,229
Telecom Italia (IT)	6,955	NTL (UK)	3,852
Charter (US)	3,634	Vivo (BR)	3,723
Bell Canada (CA)	5,233	GVT (BR)	3,723

Table 2: Top 20 ISPs based on the number of Glasnost tests conducted by their users.

7.1 Characterizing BitTorrent Differentiation

We now use the data collected during our deployment to characterize BitTorrent differentiation in the Internet. To our knowledge, such detailed characterization was not available before.

Figure 9 shows the percentage of users for whom we detected differentiation in at least one of the four tests that we widely deployed on Glasnost. Aside from a few weeks in the beginning when we did not have enough users, this percentage has stayed roughly constant around 10%. Thus, a non-negligible fraction of our testers are subject to differentiation.

We do not, however, claim that 10% of all Internet users experience differentiation. Glasnost users are self-selecting, and our data may be biased towards users that suspect their ISP to be differentiating against BitTorrent.

7.2 Understanding ISP behaviors

Our Glasnost deployment was so popular that we had hundreds of users from some of the largest ISPs worldwide. Aggregating results from all the users belonging to an ISP can provide an understanding of the extent to which the ISP differentiates traffic. Such ISP-wide perspectives are especially useful for policy makers and government regulators responsible for monitoring ISP behavior. Further, end users can compare the state of

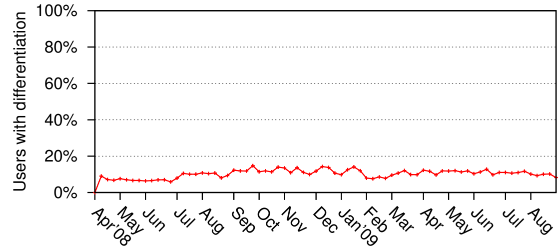


Figure 9: Percentage of tests in which we detected differentiation since March 2008.

differentiation across different ISPs to make a more informed choice when selecting their ISP.

We now turn our attention to understanding the policies of individual access ISPs. For this analysis, we map users to their access ISPs (using *whois*) and assume that the access ISP is responsible for any observed differentiation. While it is possible that the responsibility lies with a transit ISP along the path, differentiation is a more common practice amongst access ISPs [1, 5].

We limit the analysis in this section to the tests conducted in the two-month period that covers January and February 2009 because the differentiation behavior of an ISP can change over time. We select the two-month period for which we have the most data. Further, we consider only ISPs for which we have at least 100 tests in this time period. There are 140 such ISPs.

7.2.1 Basis for differentiation

Table 3 shows the list of the top-30 ISPs ranked based on the fraction of hosts that detected differentiation. Table 3 also shows how traffic is differentiated. More than half the ISPs differentiate only in the upstream direction and 7 ISPs only in the downstream direction. 20% of ISPs (e.g., Clearwire, TVCABO) differentiate in both directions. We also find that most differentiating ISPs use both content- and port-based differentiation. For only four ISPs (Free, GVT, Pipex, and Tiscali UK) do we observe an exclusive use of port-based differentiation (which is easier to evade). And only one ISP, Oi, uses content-based differentiation exclusively.

Our results show that Glasnost can shed light on how ISPs identify the traffic they differentiate.

7.2.2 Fraction of users impacted

ISPs that differentiate against BitTorrent traffic do not do so for every user. For each ISP in Table 3, Figure 10 shows the fraction of users that tested positive for differentiation. We see that in the median case only 21% of users are affected. Given our tests' low false posi-

ISP	Loc.	Upstream		Downstream	
		app	port	app	port
Bell Canada (D)	CA	×	×		
Brasil Telecom (D)	BR			×	×
BT (D)	UK	×	×		
Cablecom (C)	CH	×	×		
Canaca (D)	CA	×	×		
City Telecom (F)	HK	×	×	×	×
Clearwire (W)	US	×	×	×	×
Cogeco (C)	CA	×	×		
EastLink (C)	CA	×	×		
Free (D)	FR				×
GVT (D,F)	BR		×		
Kabel Deutschland (C)	DE	×	×	×	×
Magix (D)	SG			×	×
Oi (D)	BR			×	
ONO (C)	ES			×	×
PCCW (D)	HK	×	×	×	×
Pipex (D)	UK		×		
Rogers (C)	CA	×	×		
Shaw (C)	CA	×	×		
TekSavvy (D)	CA	×	×		
Tele2 (D)	IT	×	×		
Telenet (D)	BE		×	×	
TFN (D)	TW			×	×
Tiscali Italia (D)	IT			×	×
Tiscali UK (D)	UK		×		
TM Net (D)	MY	×	×		
TVCABO (C)	PT	×	×	×	×
UPC NL (C)	NL	×	×		
UPC Poland (C)	PL	×	×		
UPC Romania (C)	RO	×	×		

Table 3: Top 30 ISPs based on the fraction of users that are affected by traffic differentiation during January and February 2009. The table shows if the flows are differentiated based on application content (app), TCP ports, or both. The letter in parenthesis gives the type of access network the ISP runs, i.e., DSL (D), Cable (C), Fiber-To-The-Home (F), and WiMax (W).

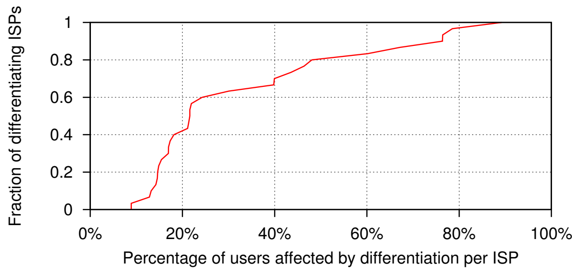


Figure 10: Typically, we detected traffic differentiation for only a fraction of an ISP's users.

tive and negative rates, this inconsistent impact within an ISP cannot be explained by inference errors alone.

Our data does not allow us to infer why only a fraction of users of an ISP experience traffic differentiation. There are many possible reasons. An ISP might choose to target only customers who generate a lot of P2P traffic, the traffic shapers might be deployed in only a portion of the ISP network, or an ISP might differentiate only during peak hours or periods of high load.

7.2.3 Dependence on time of day

One potential explanation for why only some users experience differentiation is that ISPs may differentiate only during peak hours, when the network is experiencing the greatest load. To investigate the dependence on time of day we divided our dataset into two time periods based on the local time of the user[3]. The *peak* period is 8pm–12am, and the *off-peak* period is 5–9am. These periods are strict subsets of the peak and off-peak durations for access ISPs [5, 14].

For each period we infer if an ISP differentiated traffic. Our analysis excludes ISPs that have fewer 100 measurements for either of the two time periods. This leaves us with 30 ISPs. We find that slightly more than half of these ISPs to differentiate during both peak and off-peak hours. The other ISPs, e.g., BT, Bell Canada, Kabel Deutschland, ONO, and Tiscali UK, restrict traffic differentiation to the peak period.

Our results in the last two sections show the importance of enabling end users to detect differentiation for themselves and at particular points in time. Many existing tools attempt to discover whether or not a ISP differentiates traffic [27, 30]. Since not all users of an ISP are affected by differentiation all the time, ISP-wide information alone is not sufficient for a user to determine if she experiences differentiation.

7.3 User feedback

Since our system became operational, we have received more than one hundred e-mails from users. The feedback is overwhelmingly positive, and it reveals two pieces of information. First, we find evidence of false negatives in our results. Around 6% of our emailers were skeptical when Glasnost did not discover traffic differentiation. They were convinced that their ISP differentiates, sometimes based on information their ISP publishes. If these users are right, their cases con-

[3]We used an IP-to-geolocation tool to infer the timezone of each user.

firm that our decision to minimize the false positive rate comes at the cost of false negatives. While we continue to investigate ways to reduce the false negative rate, we are pleased to report that no user has complained about the presence of a false positive.

Second, some emails requested Glasnost tests for other P2P applications such as eMule as well as non-P2P applications such as FTP, SSH, and HTTP. The constant stream of such requests motivated us to open the Glasnost platform and allow users to contribute new Glasnost tests. We describe this extension in the following section.

7.4 User-contributed Glasnost tests

It is not feasible for us to create Glasnost tests for each of the large number of applications and possible traffic differentiation policies that are of interest to users. Hence, we decided to allow users to create their own Glasnost tests using the `trace-emulate` tool that we described earlier. To create a new test, users need to capture a packet trace of their target application using `tcpdump` and then use `trace-emulate` to create a new Glasnost test from the trace. These new tests can be uploaded to our measurement servers using the Glasnost webpage. Our interface for creating new tests is targeted not at lay users, but at advanced users who have some familiarity with capturing network traces.

We have deployed this interface only recently, and we do not yet have a lot of experience with it. However, we asked a handful of our colleagues, who are doctoral students not associated with our project, to use the interface to create new Glasnost tests: they were able to create new tests quite easily.

8 Related Work

This section describes Glasnost in the context of existing work on traffic differentiation, trace replay, and measurement systems.

Traffic Differentiation. Three early studies investigated the prevalence of blocking for BitTorrent [10, 11] or for general traffic based on port numbers [4]. They found blocking to be relatively common. Our results show that gentler forms of differentiation are now much more prevalent than outright blocking.

Three recent efforts proposed techniques for detecting traffic differentiation. NetPolice [31] (previously named NVLens [30]) compares the aggregate loss rates of different flows to infer the presence of "network neutrality violations" in backbone ISPs. In contrast, Glasnost focuses on enabling individuals to detect whether they are subject to traffic differentiation.

NANO [27] uses causal inference to infer the presence of traffic performance degradation. NANO relies on a vast amount of passively collected traces from many users to infer if traversing a particular ISP leads to poorer performance for certain kinds of traffic. In contrast, Glasnost uses active measurements and a simple head-to-head comparison of two flows to quickly inform users whether they face traffic differentiation — without relying on other users. However, adding passive measurement techniques to Glasnost might enable it to detect time- or usage-dependent traffic differentiation.

DiffProbe [13] detects whether traffic differentiation based on active queue management (AQM), such as RED and weighted fair queueing, is deployed in the network path. DiffProbe complements Glasnost as it can detect differentiation that leads to small increase in latency and can identify the AQM technique used. If AQM affects application throughput, Glasnost can also detect this.

Trace replay. Monkey [6] is a TCP replay tool that takes a packet-level trace as input and generates a new trace with similar network-level properties, such as latency and bandwidth. More recent work [8] investigates ways to infer higher-level protocols from low-level packet traces. Our `trace-emulate` tool is an adaptation of such methods.

Measurement systems. Many researchers use network testbeds, such as PlanetLab [24], RON [3], and NIMI [23], to conduct measurement studies. Unlike Glasnost, these testbeds are designed explicitly for use by researchers. There are a number of tools deployed on M-Lab [16] with the goal of enhancing Internet transparency. Most of them are generic measurement tools that characterize certain features of the Internet

The DIMES project [9] is based on the SETI@home model. It uses volunteer-contributed hosts to run `traceroute` measurements that are used to map the connectivity of edge networks. The two systems, DIMES and Glasnost, offer an interesting (if unfair due to different goals) comparison of user models. DIMES relies on the ability to run arbitrary code on users' computers. It was deployed over four years ago and has about 8,000 users.

Finally, Netalyzr [17] is a web-based measurement tool that mostly focuses on the detection of networking problems. Like Glasnost, it targets lay users with an easy-to-use interface and allows them to detect, for instance, manipulation of web content by a HTTP proxy in the path or blocking of traffic on some prominent ports.

9 Conclusion

We described Glasnost, a system that we deployed more than a year ago to let ordinary users detect traffic dif-

ferentiation along their paths. More than 350,000 users from over 5,800 ISPs worldwide have used it to detect BitTorrent differentiation. We believe that our focus on making it easy for lay users to use the system and to understand its results have led to its success. Using the data gathered by Glasnost, we also presented what to our knowledge is the first detailed analysis of BitTorrent differentiation practices in the Internet. The data collected by Glasnost is available through M-Lab [16].

Over the past year, we have encountered many researchers who were skeptical about the benefits of measuring traffic differentiation. Even some of this paper's authors were initially skeptical. A common argument is that, since traffic differentiation is attracting so much attention from industry and the government, the permissible practices would soon be standardized and apparent. The skeptics might or might not be right. But the popularity of Glasnost and the positive feedback shows that many users are curious about the behavior of their Internet paths. Indeed, Glasnost's impact goes beyond traffic differentiation in our view. Its design shows one effective way to build and deploy a measurement system that satisfies such curiosities and makes the network more transparent to its users.

10 Acknowledgments

We thank Andreas Haeberlen and Alan Mislove for their contributions during the early stages of the Glasnost project. We also thank our shepherd Nick Feamster, and the anonymous reviewers for detailed feedback on this paper. Finally, we thank the people and the organizations supporting the M-Lab platform for hosting Glasnost on M-Lab servers.

References

[1] Comments of Comcast Corporation before the FCC. http://fjallfoss.fcc.gov/prod/ecfs/retrieve.cgi?native_or_pdf=pdf&id_document=6519840991.

[2] The Global Broadband Speed Test. http://www.speedtest.net.

[3] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proc. of SOSP*, 2001.

[4] R. Beverly, S. Bauer, and A. Berger. The Internet's Not a Big Truck: Toward Quantifying Network Neutrality. In *Proc. of the Passive and Active Measurement Conference (PAM)*, 2007.

[5] Canadian Radio-television and Telecommunications Commission. Review of the Internet traffic management practices of Internet service providers. http://crtc.gc.ca/PartVII/eng/2008/8646/c12_200815400.htm.

[6] Y.-C. Cheng, U. Hoelzle, N. Cardwell, S. Savage, and G. M. Voelker. Monkey See, Monkey Do: A Tool for TCP Tracing and Replaying. In *Proc. of the USENIX Technical Conference*, 2004.

[7] Comcast: Description of planned network management practices. http://downloads.comcast.net/docs/Attachment_B_Future_Practices.pdf.

[8] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz. Tupni: automatic reverse engineering of input formats. In *Proc. of CCS*, 2008.

[9] The DIMES Project. http://www.netdimes.org/.

[10] M. Dischinger, A. Mislove, A. Haeberlen, and K. P. Gummadi. Detecting BitTorrent Blocking. In *Proc. of IMC*, 2008.

[11] EFF. "Test Your ISP" Project. http://www.eff.org/testyourisp.

[12] J. Röttgers, Focus Online, 6.03.2008. Internetanbieter bremst Tauschbörsen aus. http://www.focus.de/digital/internet/kabel-deutschland_aid_264070.html.

[13] P. Kanuparthy and C. Dovrolis. DiffProbe: Detecting ISP Service Discrimination. In *Proc. of INFOCOM*, 2010.

[14] N. Laoutaris and P. Rodriguez. Good Things Come to Those Who (Can) Wait – or how to handle Delay Tolerant traffic and make peace on the Internet. In *Proc. of HotNets*, 2008.

[15] List of ISPs suspected to traffic shape BitTorrent. http://www.azureuswiki.com/index.php/Bad_ISPs.

[16] Measurement Lab. http://www.measurementlab.net.

[17] The ICSI Netalyzr. http://netalyzr.icsi.berkeley.edu.

[18] New York Times. 'Neutrality' Is New Challenge for Internet Pioneer, September 2006. http://nytimes.com/2006/09/27/technology/circuits/27neut.html.

[19] New York Times. Comcast: We're Delaying, Not Blocking, BitTorrent Traffic, October 2007. http://bits.blogs.nytimes.com/2007/10/22/comcast-were-delaying-not-blocking-bittorrent-traffic.

[20] New York Times. F.T.C. Urges Caution on Net Neutrality, June 2007. http://www.nytimes.com/2007/06/28/technology/28net.html.

[21] New York Times. F.C.C. Chairman Favors Penalty on Comcast, July 2008. http://www.nytimes.com/2008/07/11/technology/11fcc.html.

[22] OpenDPI. http://www.opendpi.org.

[23] V. Paxson, A. K. Adams, and M. Mathis. Experiences with NIMI. In *Proc. of the SAINT Workshop*, 2002.

[24] PlanetLab. http://www.planet-lab.org/.

[25] N. Spring, D. Wetherall, and T. Anderson. Scriptroute: A Public Internet Measurement Facility. In *Proc. of USITS*, 2003.

[26] Systems Research Lab, University of Colorado at Boulder. Broadband network management. http://systems.cs.colorado.edu/mediawiki/index.php/Broadband_Network_Management.

[27] M. B. Tariq, M. Motiwala, N. Feamster, and M. Ammar. Detecting Network Neutrality Violations with Causal Inference. In *Proc. of the CoNEXT Conference*, 2009.

[28] VELOCIX: New Generation Content Delivery Network. http://www.velocix.com.

[29] Vuze Network Status Monitor. http://azureus.sf.net/plugin_details.php?plugin=aznetmon.

[30] Y. Zhang, Z. M. Mao, and M. Zhang. Ascertaining the Reality of Network Neutrality Violation in Backbone ISPs. In *Proc. of ACM HotNets-VII Workshop*, 2008.

[31] Y. Zhang, Z. M. Mao, and M. Zhang. Detecting Traffic Differentiation in Backbone ISPs with NetPolice. In *Proc. of the Internet Measurement Conference (IMC)*, 2009.

EndRE: An End-System Redundancy Elimination Service for Enterprises

Bhavish Aggarwal⋆, Aditya Akella†, Ashok Anand[1]†, Athula Balachandran[1]‡,
Pushkar Chitnis⋆, Chitra Muthukrishnan[1]†, Ramachandran Ramjee⋆ and George Varghese[2]§
⋆ *Microsoft Research India;* † *University of Wisconsin-Madison;* ‡ *CMU;* §*UCSD*

Abstract

In many enterprises today, WAN optimizers are being deployed in order to eliminate redundancy in network traffic and reduce WAN access costs. In this paper, we present the design and implementation of EndRE, an alternate approach where redundancy elimination (RE) is provided as *an end system service*. Unlike middleboxes, such an approach benefits both end-to-end encrypted traffic as well as traffic on last-hop wireless links to mobile devices.

EndRE needs to be fast, adaptive and parsimonious in memory usage in order to opportunistically leverage resources on end hosts. Thus, we design a new fingerprinting scheme called SampleByte that is much faster than Rabin fingerprinting while delivering similar compression gains. Unlike Rabin fingerprinting, SampleByte can also adapt its CPU usage depending on server load. Further, we introduce optimizations to reduce server memory footprint by 33-75% compared to prior approaches. Using several terabytes of network traffic traces from 11 enterprise sites, testbed experiments and a pilot deployment, we show that EndRE delivers 26% bandwidth savings on average, processes payloads at speeds of 1.5-4Gbps, reduces end-to-end latencies by up to 30%, and translates bandwidth savings into equivalent energy savings on mobile smartphones.

1 Introduction

With the advent of globalization, networked services have a global audience, both in the consumer and enterprise spaces. For example, a large corporation today may have branch offices at dozens of cities around the globe. In such a setting, the corporation's IT admins and network planners face a dilemma. On the one hand, they could concentrate IT servers at a small number of locations. This might lower administration costs, but increase network costs and latency due to the resultant increase in WAN traffic. On the other hand, servers could be located closer to clients; however, this would increase operational costs.

This paper arises from the quest to have the best of both worlds, specifically, having the operational benefits of centralization along with the performance benefits of distribution. In recent years, protocol-independent redundancy elimination, or simply RE [20], has helped bridge the gap by making WAN communication more efficient through elimination of redundancy in traffic. Such compression is typically applied at the IP or TCP layers, for instance, using a pair of middleboxes placed at either end of a WAN link connecting a corporation's data center and a branch office. Each box caches payloads from flows that traverse the link, irrespective of the application or protocol. When one box detects chunks of data that match entries in its cache (by computing "fingerprints" of incoming data and matching them against cached data), it encodes matches using tokens. The box at the far end reconstructs original data using its own cache and the tokens. This approach has seen increasing deployment in "WAN optimizers".

Unfortunately, such middlebox-based solutions face two key drawbacks that impact their long-term usefulness: (1) Middleboxes do not cope well with end-to-end encrypted traffic and many leave such data uncompressed (e.g., [1]). Some middleboxes accommodate SSL/SSH traffic with techniques such as connection termination and sharing of encryption keys (e.g., [5]), but these weaken end-to-end semantics. (2) In-network middleboxes cannot improve performance over last-hop links in mobile devices.

As end-to-end encryption and mobile devices become increasingly prevalent, we believe that RE will be *forced out* of middleboxes and directly *into* end-host stacks. Motivated by this, we explore a new point in the design space of RE proposals — an *end-system redundancy elimination service* called EndRE. EndRE could supplement or supplant middlebox-based techniques while addressing their drawbacks. Our paper examines the costs and benefits that EndRE implies for clients and servers.

Effective end-host RE requires looking for small redundant chunks of the order of 32-64 bytes (because most enterprise transfers involve just a few packets each [16]). The standard Rabin fingerprinting algorithms (e.g., [20]) for identifying such fine scale redundancy are very expensive in terms of memory and processing especially on resource constrained clients such

[1]A part of this work was done while the authors were interns at Microsoft Research India.

[2]The author was a visiting researcher at Microsoft Research India during the course of this work.

as smartphones. Hence, we adopt a novel *asymmetric* design that systematically offloads as much of processing and memory to servers as possible, requiring clients to do no more than perform basic FIFO queue management of a small amount of memory and do simple pointer lookups to decode compressed data.

While client processing and memory are paramount, servers in EndRE need to do other things as well. This means that server CPU and memory are also crucial bottlenecks in our asymmetric design. For server processing, we propose a new fingerprinting scheme called SampleByte that is much faster than Rabin fingerprinting used in traditional RE approaches while delivering similar compression. In fact, SampleByte can be up to 10X faster, delivering compression speeds of 1.5-4Gbps. SampleByte is also *tunable* in that it has a payload sampling parameter that can be adjusted to reduce server processing if the server is busy, at the cost of reduced compression gains.

For server storage, we devise a suite of highly-optimized data structures for managing meta-data and cached payloads. For example, our Max-Match variant of EndRE (§5.2.2) requires 33% lower memory compared to [20]. Our Chunk-Match variant (§5.2.1) cuts down the aggregate memory requirements at the server by 4X compared to [20], while sacrificing a small amount of redundancy.

We conduct a thorough evaluation of EndRE. We analyze several terabytes of traffic traces from 11 different enterprise sites and show that EndRE can deliver significant bandwidth savings (26% average savings) on enterprise WAN links. We also show significant latency and energy savings from using EndRE. Using a testbed over which we replay enterprise HTTP traffic, we show that latency savings of up to 30% are possible from using EndRE, since it operates above TCP, thereby reducing the number of roundtrips needed for data transfer. Similarly, on mobile smartphones, we show that the low decoding overhead on clients can help translate bandwidth savings into significant energy savings compared to no compression. We also report results from a small-scale deployment of EndRE in our lab.

The benefits of EndRE come at the cost of memory and CPU resources on end systems. We show that a median EndRE client needs only 60MB of memory and negligible amount of CPU. At the server, since EndRE is adaptive, it can opportunistically trade-off CPU/memory for compression savings.

In summary, we make the following contributions:

(1) We present the design of EndRE, an end host based redundancy elimination service (§4).

(2) We present new asymmetric RE algorithms and optimized data structures that limit client processing and memory requirements, and reduce server memory usage by 33-75% and processing by 10X compared to [20] while delivering slightly lower bandwidth savings (§5).

(3) We present an implementation of EndRE as part of Windows Server/7/Vista as well as on Windows Mobile 6 operating systems (§6).

(4) Based on extensive analysis using several terabytes of network traffic traces from 11 enterprise sites, testbed experiments and a small-scale deployment, we quantify the benefits and costs of EndRE (§7 - §9)

2 Related Work

Over the years, enterprise networks have used a variety of mechanisms to suppress duplicate data from their network transfers. We review these mechanisms below.

Classical approaches: The simplest RE approach is to compress objects end-to-end. It is also the least effective because it does not exploit redundancy due to repeated accesses of similar content. Object caches can help in this regard, but they are unable to extract cross-object redundancies [8]. Also object caches are application-specific in nature; e.g., Web caches cannot identify duplication in other protocols. Furthermore, an increasing amount of data is dynamically generated and hence not cacheable. For example, our analysis of enterprise traces shows that a majority of Web objects are not cacheable, and deploying an HTTP proxy would only yield 5% net bandwidth savings. Delta encoding can eliminate redundancy of one Web object with respect to another [14, 12]. However, like Web caches, delta encoding is application-specific and ineffective for dynamic content.

Content-based naming: The basic idea underlying EndRE is that of *content-based naming* [15, 20], where an object is divided into chunks and indexed by computing hashes over chunks. Rabin Fingerprinting [18] is typically used to identify chunk boundaries. In file systems such as LBFS [15] and Shark [9], content-based naming is used to identify similarities across different files and across versions of the same file. Only unique chunks are transmitted between file servers and clients, resulting in lower bandwidth consumption. A similar idea is used in value-based Web caching [19], albeit between a Web server and its client. Our chunk-based EndRE design is patterned after this approach, with key modifications for efficiency (§5).

Generalizing these systems, DOT [21] proposes a "transfer service" as an interface between applications and the network. Applications pass the object they want to send to DOT. Objects are split into chunks and the sender sends chunk hashes to the receiver. The receiver maintains a cache of earlier received chunks and requests only the chunks that were not found in its cache or its neighbors' caches. Thus, DOT can leverage TBs of cache in the disks of an end host and its peers to elim-

inate redundancy. Similarly, SET [17] exploits chunk-level similarity in downloading related large files. DOT and SET use an average chunk size of 2KB or more. These approaches mainly benefit large transfers; the extra round trips that can only be amortized over the transfer lengths. In contrast, EndRE identifies redundancy across chunk sizes of 32 bytes and does not impose additional latency. It is also limited to main-memory based caches of size 1-10MB per pair of hosts (§5). Thus, EndRE and DOT complement each other.

Protocol-independent WAN optimizers. To overcome the limitations of the "classical" approaches, enterprises have moved increasingly toward protocol independent RE techniques, used in WAN optimizers. These WAN optimizers can be of two types, depending on which network layer they operate at, namely, IP layer devices [20, 3] or higher-layer devices [1, 5].

In either case, special middleboxes are deployed at either end of a WAN link to index all content exchanged across the link, and identify and remove partial redundancies on the fly. Rabin fingerprinting [18] is used to index content and compute overlap (similar to [20, 15]). Both sets of techniques are highly effective at reducing the utilization of WAN links. However, as mentioned earlier, they suffer from two key limitations, namely, lack of support for end-to-end encryption and for resource-constrained mobile devices.

3 Motivation

In exploring an end-point based RE service, one of the main issues we hope to address is whether such a service can offer bandwidth savings approaching that of WAN optimizers. To motivate the likely benefits of an end-point based RE service, we briefly review two key findings from our earlier study [8] of an IP-layer WAN optimizer [7].

First, we seek to identify the origins of redundancy. Specifically, we classify the contribution of redundant byte matches to bandwidth savings as either *intra-host* (current and matched packet in cache have identical source-destination IP addresses) or *inter-host* (current and matched packets differ in at least one of source or destination IP addresses). We were limited to a 250MB cache size given the large amount of meta-data necessary for this analysis, though we saw similar compression savings for cache sizes up to 2GB. Surprisingly, our study revealed that *over 75% of savings were from intra-host matches*. This implies that a pure end-to-end solution could potentially deliver a significant share of the savings obtained by an IP WAN optimizer, since the contribution due to inter-host matches is small. However, this finding holds good only if end systems operate with similar (large) cache sizes as middleboxes, which is impractical. This brings us to the second key finding. Examining the temporal characteristics of redundancy, we found that the redundant matches in the WAN optimizer displayed a high degree of temporal locality with *60-80% of middlebox savings arising from matches with packets in the most recent 10% of the cache*. This implies that small caches could capture a bulk of the savings of a large cache.

Taken together, these two findings suggest that an end point-based RE system with a small cache size can indeed deliver a significant portion of the savings of a WAN optimizer, thus motivating the design of EndRE.

Finally, note that, the focus of comparison in this section is between an IP-layer WAN optimizer with an in-memory cache (size is $O(GB)$) and an end-system solution. The first finding is not as surprising once we realize that the in-memory cache gets recycled frequently (on the order of tens of minutes) during peak hours on our enterprise traces, limiting the possibility for inter-host matches. A WAN optimizer typically also has a much larger on-disk cache (size is $O(TB)$) which may see a large fraction of inter-host matches; an end-system disk cache-based solution such as DOT [21] could capture analogous savings.

4 Design Goals

EndRE is designed to optimize data transfers in the direction from servers in a remote data center to clients in the enterprise, since this captures a majority of enterprise traffic. We now list five design goals for EndRE — the first two design goals are shared to some extent by prior RE approaches, but the latter three are unique to EndRE.

1. Transparent operation: For ease of deploy-ability, the EndRE service should require no changes to existing applications run within the data center or on clients.

2. Fine-grained operation: Prior work has shown that many enterprise network transfers involve just a few packets [16]. To improve end-to-end latencies and provide bandwidth savings for such short flows, EndRE must work at fine granularities, suppressing duplicate byte strings as small as 32-64B. This is similar to [20], but different from earlier proposals for file-systems [15] and Web caches [19] where the sizes of redundancies identified are 2-4KB.

3. Simple decoding at clients: EndRE's target client set includes battery- and CPU-constrained devices such as smart-phones. While working on fine granularities can help identify greater amounts of redundancy, it can also impose significant computation and decoding overhead, making the system impractical for these devices. Thus, a unique goal is to design algorithms that limit client overhead by *offloading* all compute-intensive actions to *servers*.

4. Fast and adaptive encoding at servers: EndRE is

designed to opportunistically leverage CPU resources on end hosts when they are not being used by other applications. Thus, unlike commercial WAN optimizers and prior RE approaches [20], EndRE must *adapt* its use of CPU based on server load.

5. Limited memory footprint at servers and clients: EndRE relies on data caches to perform RE. However, memory on servers and clients could be limited and may be actively used by other applications. Thus, EndRE must use as minimal memory on end-hosts as possible through the use of optimized data structures.

5 EndRE Design

In this section, we describe how EndRE's design meets the above goals.

EndRE introduces RE modules into the network stacks of clients and remote servers. Since we wish to be transparent to applications, EndRE could be implemented either at the IP-layer or at the socket layer (above TCP). As we argue in §6, we believe that socket layer is the right place to implement EndRE. Doing so offers key performance benefits over an IP-layer approach, and more importantly, shields EndRE from network-level events (e.g., packet losses and reordering), making it simpler to implement.

There are two sets of modules in EndRE, those belonging on servers and those on clients. The server-side module is responsible for identifying redundancy in network data by comparing against a cache of prior data, and encoding the redundant data with shorter meta-data. The meta-data is essentially a set of <offset, length> tuples that are computed with respect to the client-side cache. The client-side module is trivially simple: it consists of a fixed-size circular FIFO log of packets and simple logic to decode the meta-data by "de-referencing" the offsets sent by the server. Thus, most of the complexity in EndRE is mainly on the server side and we focus on that here.

Identifying and removing redundancy is typically accomplished [20, 7] by the following two steps:

• *Fingerprinting:* Selecting a few "representative regions" for the current block of data handed down by application(s). We describe four fingerprinting algorithms in §5.1 that differ in the trade-off they impose between *computational overhead* on the server and the *effectiveness* of RE.

• *Matching and Encoding:* Once the representative regions are identified, we examine two approaches for identification of redundant content in §5.2: (1) Identifying chunks of representative regions that repeat in full across data blocks, called Chunk-Match and (2) Identifying maximal matches around the representative regions that are repeated across data blocks, called Max-Match. These two approaches differ in the trade-off between the *memory overhead* imposed on the server and the *effectiveness* of RE.

Next, we describe EndRE's design in detail, starting with selection of representative regions, and moving on to matching and encoding.

5.1 Fingerprinting: Balancing Server Computation with Effectiveness

In this section, we outline four approaches for identifying the representative payload regions at the server that vary in the way they trade-off between computational overhead and the effectiveness of RE. In some of the approaches, computational overhead can be *adaptively tuned* based on server CPU load, and the effectiveness of RE varies accordingly. Although three of the four approaches were proposed earlier, the issue of their computational overhead has not received enough attention. Since this issue is paramount for EndRE, we consider it in great depth here. We also propose a new approach, SAMPLEBYTE, that combines the salient aspects of prior approaches.

We first introduce some notation and terminology to help explain the approaches. Restating from above, a "data block" or simply a "block" is a certain amount of data handed down by an application to the EndRE module at the socket layer. Each data block can range from a few bytes to tens of kilobytes in size.

Let w represent the size of the minimum redundant string (contiguous bytes) that we would like to identify. For a data block of size S bytes, $S \geq w$, a total of $S - w + 1$ strings of size w are potential candidates for finding a match. Typical values for w range from 12 to 64 bytes. Based on our findings of redundant match length distribution in [8], we choose a default value of $w = 32$ bytes to maximize the effectiveness of RE. Since $S >> w$, the number of such candidate strings is on the order of the number of bytes in the data block/cache. Since it is impractical to match/store all possible candidates, a fraction $1/p$ "representative" candidates are chosen.

Let us define *markers* as the first byte of these chosen candidate strings and *chunks* as the string of bytes between two markers. Let *fingerprints* be a pseudo-random hash of fixed w-byte strings beginning at each marker and *chunk-hashes* be hashes of the variable sized chunks. Note that two fingerprints may have overlapping bytes; however, by definition, chunks are disjoint. The different algorithms, depicted in Figure 1 and discussed below, primarily vary in the manner in which they choose the markers, from which one can derive chunks, fingerprints, and chunk-hashes. As we discuss later in §5.2, the Chunk-Match approach uses chunk-hashes while Max-Match uses fingerprints.

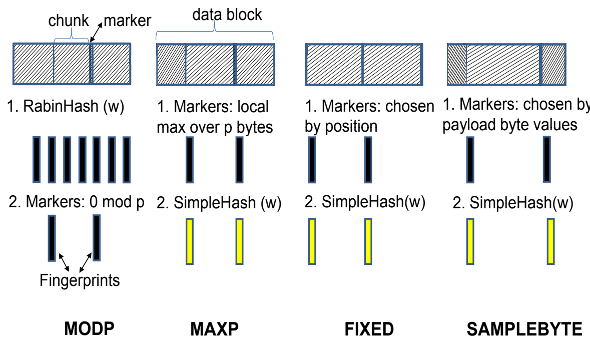


Figure 1: Fingerprinting algorithms with chunks, markers and fingerprints; chunk-hashes, not shown, can be derived from chunks

```
1  //Let w = 32; p = 32; Assume len ≥ w;
2  //RabinHash() computes RABIN hash over a w byte window
3  MODP(data, len)
4      for(i = 0; i < w − 1; i + +)
5          fingerprint = RabinHash(data[i]);
6      for(i = w − 1; i < len; i + +)
7          fingerprint = RabinHash(data[i]);
8          if (fingerprint % p == 0) //MOD
9              marker = i − w + 1;
10             store marker, fingerprint in table;
```

Figure 2: MODP Fingerprinting Algorithm

5.1.1 MODP

In the "classical" RE approaches [20, 7, 15], the set of fingerprints are chosen by first computing Rabin-Karp hash [18] over sliding windows of w contiguous bytes of the data block. A fraction $1/p$ are chosen whose fingerprint value is *0 mod p*. Choosing fingerprints in this manner has the advantage that the set of representative fingerprints for a block remains mostly the same despite small amount of insertions/deletions/reorderings since the markers/fingerprints are chosen based on content rather than position.

Note that two distinct operations — marker identification and fingerprinting — are both handled by the same hash function here. While this *appears* elegant, it has a cost. Specifically, the per block computational cost is independent of the sampling period, p (lines 4–7 in Figure 2). Thus, this approach *cannot* adapt to server CPU load conditions (e.g., by varying p). Note that, while the authors of [20] report some impact of p on processing speed, this impact is attributed to the overhead of managing meta-data (line 10). We devise techniques in §5.2 to significantly reduce the overhead of managing meta-data, thus, making fingerprint computation the main bottleneck.

```
1  //Let w = 32; p = 32; Assume len ≥ w;
2  //SAMPLETABLE[i] maps byte i to either 0 or 1
3  //Jenkinshash() computes hash over a w byte window
4  SAMPLEBYTE(data, len)
5      for(i = 0; i < len − w; i + +)
6          if (SAMPLETABLE[data[i]] == 1)
7              marker = i;
8              fingerprint = JenkinsHash(data + i);
9              store marker, fingerprint in table;
10             i = i + p/2;
```

Figure 3: SAMPLEBYTE Fingerprinting Algorithm

5.1.2 MAXP

Apart from the conflation of marker identification and fingerprinting, another shortcoming of the MODP approach is that the fingerprints/markers are chosen based on a *global* property, i.e., fingerprints have to take certain pre-determined values to be chosen. The markers for a given block may be clustered and there may be large intervals without any markers, thus, limiting redundancy identification opportunities. To guarantee that an adequate number of fingerprints/markers are chosen uniformly from each block, markers can be chosen as bytes that are *local-maxima over each region of p bytes* of the data block [8]. Once the marker byte is chosen, an efficient hash function such as Jenkins Hash [2] can be used to compute the fingerprint. By increasing p, fewer maxima-based markers need to be identified, thereby reducing CPU overhead.

5.1.3 FIXED

While markers in both MODP and MAXP are chosen based on content of the data block, the computation of Rabin hashes and local maxima can be expensive. A simpler approach is to be content-agnostic and simply select *every* p^{th} *byte as a marker*. Since markers are simply chosen by position, marker identification incurs no computational cost. Once markers are chosen, S/p fingerprints are computed using Jenkins Hash as in MAXP. While this technique is very efficient, its effectiveness in RE is not clear as it is not robust to small changes in content. While prior works in file systems (e.g., [15]), where cache sizes are large ($O(TB)$), argue against this approach, it is not clear how ineffective FIXED will be in EndRE where cache sizes are small ($O(MB)$).

5.1.4 SAMPLEBYTE

MAXP and MODP are content-based and thus robust to small changes in content, while FIXED is content-agnostic but computationally efficient. We designed SAMPLEBYTE (Figure 3) to combine the robustness of a content-based approach with the computational efficiency of FIXED. It uses a 256-entry lookup table with

a few predefined positions set. As the data block is scanned byte-by-byte (line 5), *a byte is chosen as a marker if the corresponding entry in the lookup table is set* (line 6–7). Once a marker is chosen, a fingerprint is computed using Jenkins Hash (line 8), and $p/2$ bytes of content are skipped (line 10) before the process repeats. Thus, SAMPLEBYTE is content-based, albeit based on a single byte, while retaining the content-skipping and the computational characteristics of FIXED.

One clear concern is whether such a naive marker identification approach will do badly and cause the algorithm to either over-sample or under-sample. First, note that MODP with 32-64 byte rolling hashes was originally used in file systems [15] where chunk sizes were large (2-4KB). Given that we are interested in sampling as frequent as every 32-64 bytes, sampling chunk boundaries based on 1-byte content values is not as radical as it might first seem. Also, note that if x entries of the 256-entry lookup table are randomly set (where $256/x = p$), then the expected sampling frequency is indeed $1/p$. In addition, SAMPLEBYTE skips $p/2$ bytes after each marker selection to avoid oversampling when the content bytes of data block are not uniformly distributed (e.g., when the same content byte is repeated contiguously). Finally, while a purely random selection of $256/x$ entries does indeed perform well in our traces, we use a lookup table derived based on the heuristic described below. This heuristic outperforms the random approach and we have found it to be effective after extensive testing on traces (see §8).

Since the number of unique lookup tables is large (2^{256}), we use an offline, greedy approach to generate the lookup table. Using network traces from one of the enterprise sites we study as training data (site 11 in Table 2), we first run MAXP to identify redundant content and then sort the characters in descending order of their presence in the identified redundant content. We then add these characters one at a time, setting the corresponding entries in the lookup table to 1, and stop this process when we see diminishing gains in compression. The intuition behind this approach is that characters that are more likely to be part of redundant content should have a higher probability of being selected as markers. The characters selected from our training data were 0, 32, 48, 101, 105, 115, 116, 255. While our current approach results in a static lookup table, we are looking at online dynamic adaptation of the table as part of future work.

Since SAMPLEBYTE skips $p/2$ bytes after every marker selection, the fraction of markers chosen is $\leq 2/p$, irrespective of the number of entries set in the table. By increasing p, fewer markers/fingerprints are chosen, resulting in reduced CPU overhead.

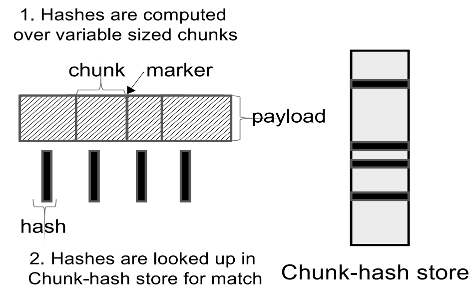


Figure 4: Chunk-Match: only chunk-hashes stored

5.2 Matching and Encoding: Optimizing Storage and Client Computation

Once markers and fingerprints are identified, identification of redundant content can be accomplished in two ways: (1) Identifying *chunks* of data that repeat in full across data blocks, called Chunk-Match, or (2) Identifying *maximal matches around fingerprints* that are repeated across data blocks, called Max-Match. Both techniques were proposed in prior work: the former in the context of file systems [15] and Web object compression [19], and the latter in the context of IP WAN optimizer [20]. However, prior proposals impose significant storage and CPU overhead.

In what follows we describe how the overhead impacts both servers and clients, and the two techniques we employ to address these overheads. The first technique is to leverage *asymmetry* between servers and clients. We propose that clients offload most of the computationally intensive operations (e.g., hash computations) and memory management tasks to the server. The second technique is to exploit the inherent *structure* within the data maintained at servers and clients to optimize memory usage.

5.2.1 Chunk-Match

This approach (Figure 4) stores hashes of the chunks in a data block in a “Chunk-hash store”. Chunk-hashes from payloads of future data blocks are looked up in the Chunk-hash store to identify if one or more chunks have been encountered earlier. Once matching chunks are identified, they are replaced by meta-data.

Although similar approaches were used in prior systems, they impose significantly higher overhead if employed directly in EndRE. For example, in LBFS [15], clients have to update their local caches with mappings between new content-chunks and corresponding content-hashes. This requires expensive SHA-1 hash computation at the client. Value-based web caching [19] avoids the cost of hash computation at the client by having the server send the hash with each chunk. However, the client still needs to store the hashes, which is a significant overhead for small chunk sizes. Also, sending

hashes over the network adds significant overhead given that the hash sizes (20 bytes) are comparable to average chunk sizes in EndRE (32-64 bytes).

EndRE optimizations: We employ two ideas to improve overhead on clients and servers.

(1) Our design carefully *offloads all storage management and computation to servers*. A client simply maintains a fixed-size circular FIFO log of data blocks. The server emulates client cache behavior on a *per-client basis*, and maintains within its Chunk-hash store a mapping of each chunk hash to the start memory addresses of the chunk in a client's log along with the length of the chunk. For each matching chunk, the server simply encodes and sends a four-byte <offset, length> tuple of the chunk in the client's cache. The client simply "de-references" the offsets sent by the server and reconstructs the compressed regions from local cache. This approach avoids the cost of storing and computing hashes at the client, as well as the overhead of sending hashes over the network, at the cost of slightly higher processing and storage at the server end.

(2) In traditional Chunk-Match approaches, the server maintains a log of the chunks locally. We observe that the server only needs to maintain an up-to-date chunk-hash store, but *it does not need to store the chunks themselves* as long as the chunk hash function is collision resistant. Thus, when a server computes chunks for a new data block and finds that some of the chunks are not at the client by looking up the chunk-hash store, it inserts mappings between the new chunk hashes and their expected locations in the client cache.

In our implementation, we use SHA-1 to compute 160 bit hashes, which has good collision-resistant properties. Let us now compute the storage requirements for Chunk-Match assuming a sampling period p of 64 bytes and a cache size of 16MB. The offset to the 16MB cache can be encoded in 24 bits and the length encoded in 8 bits assuming the maximum length of a chunk is limited to 256 bytes (recall that chunks are variable sized). Thus, server meta-data storage is 24 bytes per 64-byte chunk, comprising 4-bytes for the <offset, length> tuple and 20-bytes for SHA-1 hash. This implies that server memory requirement is about 38% of the client cache size.

5.2.2 Max-Match

A drawback of Chunk-Match is that it can only detect exact matches in the chunks computed for a data block. It could miss redundancies that, for instance, span contiguous portions of neighboring chunks or redundancies that only span portions of chunks. An alternate approach, called Max-Match, proposed for IP WAN optimizer [20, 7] and depicted in Figure 5, can identify such redundancies, albeit at a higher memory cost at the server.

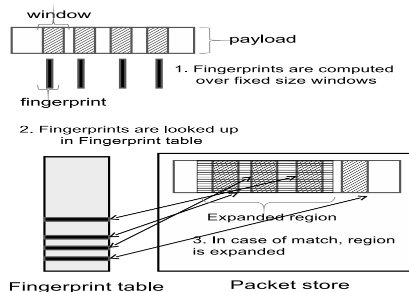


Figure 5: Max-Match: matched region is expanded

index (implicit fingerprint, 18 bits)	fingerprint remainder (8 bits)	offset (24 bits)
0	...	...
...	...	...
$2^{18}-1$	...	...

Table 1: 1MB Fingerprint store for 16MB cache

In Max-Match, fingerprints computed for a data block serve as random "hooks" into the payload around which more redundancies can be identified. The computed fingerprints for a data block are compared with a "fingerprint store" that holds fingerprints of all past data blocks. For each matching fingerprint, the corresponding matching data block is retrieved from the cache and the match region is expanded byte-by-byte in both directions to obtain the *maximal region* of redundant bytes (Figure 5). Matched regions are then encoded with <offset, length> tuples.

EndRE optimizations: We employ two simple ideas to improve the server computation and storage overhead.

First, since Max-Match relies on byte-by-byte comparison to identify matches, fingerprint collisions are not costly; any collisions will be recovered via an extra memory lookup. This allows us to significantly *limit fingerprint store maintenance overhead for all four algorithms* since fingerprint values are simply overwritten without separate bookkeeping for deletion. Further, a simple hash function that generates a few bytes of hash value as a fingerprint (e.g., Jenkins hash [2]) is sufficient.

Second, we optimize the representation of the fingerprint hash table to limit storage needs. Since the mapping is from a fingerprint to an offset value, the fingerprint itself need not be stored in the table, at least in its entirety. The index into the fingerprint table can implicitly represent part of the fingerprint and only the remaining bits, if any, of the fingerprint that are not covered by the index can be stored in the table. In the extreme case, the fingerprint table is simply a contiguous set of offsets, indexed by the fingerprint hash value.

Table 1 illustrates the fingerprint store for a cache size of 16MB and $p = 64$. In this case, the number of fingerprints to index the entire cache is simply $2^{24}/64$ or 2^{18}. Using a table size of 2^{18} implies that 18 bits of a fingerprint are implicitly stored as the index of the table. The

offset size necessary to represent the entire cache is 24 bits. Assuming we store an additional 8 bits of the fingerprint as part of the table, the entire fingerprint table can be compactly stored in a table of size $2^{18} * 4$ bytes, or 6% of the cache size. A sampling period of 32 would double this to 12% of the cache size. This leads to a significant reduction in fingerprint meta-data size compared to the 67% indexing overhead in [20] or the 50% indexing overhead in [7].

These two optimizations are not possible in the case of Chunk-Match due to the more stringent requirements on collision-resistance of chunk hashes. However, server memory requirement for Chunk-Match is only 38% of client cache size, which is still significantly lower than 106% of the cache size (cache + fingerprint store) needed for Max-Match.

6 Implementation

In this section, we discuss our implementation of EndRE. We start by discussing the benefits of implementing EndRE at the socket layer above TCP.

6.1 Performance benefits

Bandwidth: In the socket-layer approach, RE can operate at the size of socket writes which are typically larger than IP layer MTUs. While Max-Match and Chunk-Match do not benefit from these larger sized writes since they operate at a granularity of 32 bytes, the large size helps produce higher savings if a compression algorithm like GZIP is *additionally* applied, as evaluated in §9.1.
Latency: The socket-layer approach will result in fewer packets transiting between server and clients, as opposed to the IP layer approach which merely compresses packets without reducing their number. This is particularly useful in lowering completion times for short flows, as evaluated in §9.2.

6.2 End-to-end benefits

Encryption: When using socket-layer RE, payload encrypted in SSL can be compressed before encryption, providing RE benefits to protocols such as HTTPS. IP-layer RE will leave SSL traffic uncompressed.
Cache Synchronization: Recall that both Max-Match and Chunk-Match require caches to be synchronized between clients and servers. One of the advantages of implementing EndRE above TCP is that TCP ensures reliable in-order delivery, which can help with maintaining cache synchronization. However, there are still two issues that must be addressed.

First, multiple simultaneous TCP connections may be operating between a client and a server, resulting in ordering of data across connections not being preserved. To account for this, we implement a simple sequence number-based *reordering mechanism*.

Second, TCP connections may get reset in the middle of a transfer. Thus, packets written to the cache at the server end may not even reach the client, leading to cache inconsistency. One could take a *pessimistic* or *optimistic* approach to maintaining consistency in this situation. In the pessimistic approach, writes to the server cache are performed only after TCP ACKs for corresponding segments are received at the server. The server needs to monitor TCP state, detect ACKs, perform writes to its cache and notify the client to do the same. In the optimistic approach, the server writes to the cache but monitors TCP only for reset events. In case of connection reset (receipt of a TCP RST from client or a local TCP timeout), the server simply notifies the client of the last sequence number that was written to the cache for the corresponding TCP connection. It is then the client's responsibility to detect any missing packets and recover these from the server. We adopt the *optimistic approach of cache writing* for two reasons: (1) Our redundancy analysis [8] indicated that there is high temporal locality of matches; a pessimistic approach over a high bandwidth-delay product link can negatively impact compression savings; (2) The optimistic approach is easier to implement since only for reset events need to be monitored rather than every TCP ACK.

6.3 Implementation

We have implemented EndRE above TCP in Windows Server/Vista/7. Our default fingerprinting algorithm is SAMPLEBYTE with a sampling period, $p = 32$. Our packet cache is a circular buffer 1-16MB in size per pairs of IP addresses. Our fingerprint store is also allocated a bounded memory based on the values presented earlier. We use a simple resequencing buffer with a priority queue to handle re-ordering across multiple parallel TCP streams. At the client side, we maintain a fixed size circular cache and the decoding process simply involves lookups of specified data segments in the cache.

In order to enable protocol independent RE, we transparently capture application payloads on the server side and TCP payloads at the client side at the TCP stream layer, that lies between the application layer and the TCP transport layer. We achieve this by implementing a kernel level filter driver based on Windows Filtering Platform (WFP) [6]. This implementation allows seamless integration of EndRE with all application protocols that use TCP, with no modification to application binaries or protocols. We also have a management interface that can be used to restrict EndRE only to specific applications. This is achieved by predicate-based filtering in WFP, where predicates can be application IDs, source and/or destination IP addresses/ports.

Finally, we have also implemented the client-side of EndRE on mobile smartphones running the Windows

Trace Name (Site #)	Unique Client IPs	Dates (Total Days)	Size (TB)
Small Enterprise (Sites 1-2)	29-39	07/28/08 - 08/08/08 (11) 11/07/08 - 12/10/08 (33)	0.5
Medium Enterprise (Sites 3-6)	62-91	07/28/08 - 08/08/08 (11) 11/07/08 - 12/10/08 (33)	1.5
Large Enterprise (Sites 7-10)	101-210	07/28/08 - 08/08/08 (11) 11/07/08 - 12/10/08 (33)	3
Large Research Lab (Site 11, training trace)	125	06/23/08 - 07/03/08 (11)	1

Table 2: Data trace characteristics (11 sites)

Mobile 6 OS. However, since Windows Mobile 6 does not support Windows Filtering Platform, we have implemented the functionality as a user-level proxy.

7 Evaluation approach

We use a combination of trace-based and testbed evaluation to study EndRE. In particular, we quantify bandwidth savings and evaluate scalability aspects of EndRE using enterprise network traffic traces; we use a testbed to quantify processing speed and evaluate latency and energy savings. We also report results from a small pilot deployment (15 laptops) in our lab spanning 1 week.

Traces: Our trace-based evaluation is based on full packet traces collected at the WAN access link of 11 corporate enterprise locations. The key characteristics of our traces are shown in Table 2. We classify the enterprises as small, medium or large based on the number of internal host IP addresses seen (less than 50, 50-100, and 100-250, respectively) in the entire trace at each of these sites. While this classification is somewhat arbitrary, we use this division to study if the benefits depend on the size of an enterprise. Note that the total amount of traffic in each trace is approximately correlated to the number of host IP addresses, though there is a large amount of variation from day to day. Typical incoming traffic numbers for small enterprises varied from 0.3-10GB/day, for medium enterprises from 2-12GB/day and for large enterprises from 7-50GB/day. The access link capacities at these sites varied from a few Mbps to several tens of Mbps. The total size of traffic we study (including inbound/outbound traffic and headers) is about 6TB.

Testbed: Our testbed consists of a desktop server connected to a client through a router. In wireline experiments, the router is a dual-homed PC capable of emulating links of pre-specified bandwidth and latency. In wireless experiments, the router is a WiFi access point. The server is a desktop PC running Windows Server 2008. The client is a desktop PC running Windows Vista or Windows 7 in the wireline experiments, and a Samsung mobile smartphone running Windows Mobile 6 in the wireless experiments.

8 Costs

In this section, we quantify the CPU and memory costs of our implementation of EndRE. Though our evaluation focus largely on Max-Match, we also provide a brief analysis of Chunk-Match.

Max-Match	Fingerprint		InlineMatch		Admin	
$p \rightarrow$	32	512	32	512	32	512
MODP	526.7	496.7	9.6	6.8	4.8	0.6
MAXP	306.3	118.8	10.1	7.7	5.2	0.5
FIXED	69.4	14.2	7.1	4.7	4.7	0.4
SAMPLEBYTE(SB)	76.8	20.2	9.5	6.1	3.0	0.7

Table 3: CPU Time(s) for different algorithms

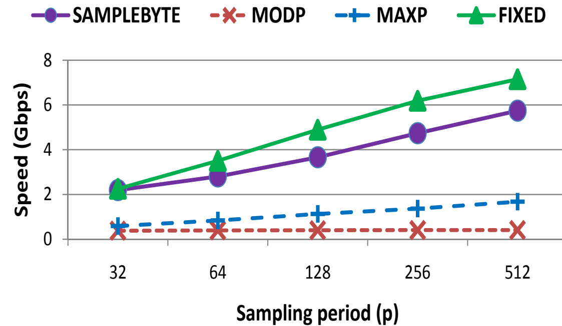


Figure 6: Max-Match processing speed

8.1 CPU Costs

Micro-benchmarks: We first focus on micro-benchmarks for different fingerprinting algorithms using Max-Match for a cache size of 10MB between a given client-server pair (we examine cache size issues in detail in §8.2). Table 3 presents a profiler-based analysis of the costs of the three key processing steps on a single large packet trace as measured on a 2GHz 64-bit Intel Xeon processor. The fingerprinting step is responsible for identifying the markers/fingerprints; the Inline-Match function is called as fingerprints are generated; and the Admin function is used for updating the fingerprint store. Of these steps, only the fingerprinting step is distinct for the algorithms, and is also the most expensive.

One can clearly see that fingerprinting is expensive for MODP and is largely independent of p. Fingerprinting for MAXP is also expensive but we see that as p is increased, the cost of fingerprinting comes down. In the case of FIXED and SAMPLEBYTE, as expected, fingerprinting cost is low, with significant reductions as p is increased.

Finally, note that the optimizations detailed earlier for updating the fingerprint store in Max-Match result in low cost for the Admin function in all the algorithms. Since matching and fingerprinting are interleaved [20], the cost of fingerprinting and matching functions, and hence total processing speed, depend on the redundancy of a particular trace. We next compute the *average* processing speed for the different algorithms over a large set of traces.

Trace analysis: Figure 6 plots the average processing speed in Gbps at the server for Max-Match for different

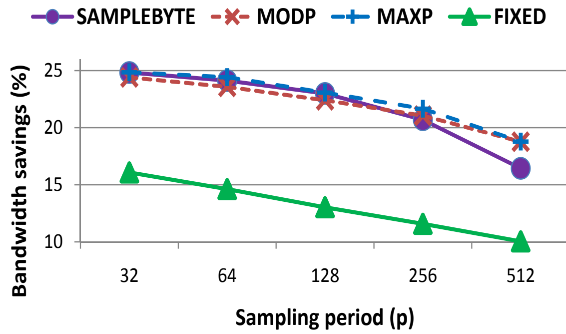


Figure 7: Max-Match bandwidth savings

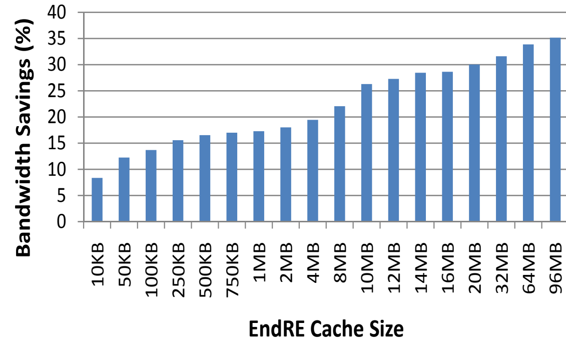


Figure 8: Cache size vs overall bandwidth savings

fingerprinting algorithms, while Figure 7 plots the average bandwidth savings. We assume a packet cache size of 10MB. We use the 11-day traces for sites 1-10 in Table 2.

We make a number of observations from these figures. First, the processing speed of MODP is about 0.4Gbps and, as discussed in §5, is largely unaffected by p. Processing speed for MAXP ranges from $0.6 - 1.7$Gbps, indicating that the CPU overhead can be decreased by increasing p. As expected, FIXED delivers the highest processing speed, ranging from $2.3 - 7.1$Gbps since it incurs no cost for marker identification. Finally, SAMPLEBYTE delivers performance close to FIXED, ranging from $2.2 - 5.8$Gbps, indicating that the cost of identification based on a single byte is low. Second, examining the compression savings, the curves for MODP, MAXP, and SAMPLEBYTE in Figure 7 closely overlap for the most part with SAMPLEBYTE under-performing the other two only when the sampling period is high (at $p = 512$, it appears that the choice of markers based on a single-byte may start to lose effectiveness). On the other hand, FIXED significantly under-performs the other three algorithms in terms of compression savings, though in absolute terms, the saving from FIXED are surprisingly high.

While the above results were based on a cache size of 10MB, typical for EndRE, a server is likely to have multiple simultaneous such connections in operation. Thus, in practice, it is unlikely to benefit from having beneficial CPU cache effects that the numbers above portray. We thus conducted experiments with large cache sizes (1-2GB) and found that processing speed indeed falls by about 30% for the algorithms. Taking this overhead into account, SAMPLEBYTE provides server processing speeds of $1.5 - 4$Gbps. To summarize, *SAMPLEBYTE provides just enough randomization for identification of chunk markers that allows it to deliver the compression savings of MODP/MAXP while being inexpensive enough to deliver processing performance, similar to FIXED, of* $1.5 - 4Gbps$.

In the case of Chunk-Match, the processing speed (not shown) is only 0.1-0.2Gbps. This is mainly due to SHA1 hash computation (§5.2.1) and the inability to use the fingerprint store optimizations of Max-Match (§5.2.2). We are examining if a cheaper hash function coupled with an additional mechanism to detect collision and recover payload through retransmissions will improve performance without impacting latency.

Client Decompression: The processing cost for decompression at the end host *client* is negligible since EndRE decoding is primarily a memory lookup in the client's cache; our decompression speed is 10Gbps. We examine the impact of this in greater detail when we evaluate end-system energy savings from EndRE in §9.3.

8.2 Memory Costs

Since EndRE requires a cache per communicating client-server pair, quantifying the memory costs at both clients and servers is critical to estimating the scalability of the EndRE system. In the next two sections, we answer the following two key questions: 1) what cache size limit do we provision for the EndRE service between a single client-server pair? 2) Given the cache size limit for one pair, what is the cumulative memory requirement at clients and servers?

8.2.1 Cache Size versus Savings

To estimate the cache size requirements of EndRE, we first need to understand the trade-off between cache sizes and bandwidth savings. For the following discussion, unless otherwise stated, by cache size, we refer to the client cache size limit for EndRE service with a given server. The server cache size can be estimated from this value depending on whether Max-Match or Chunk-Match is used (§5). Further, while one could provision different cache size limits for each client-server pair, for administrative simplicity, we assume that cache size limits are identical for all EndRE nodes.

Figure 8 presents the overall bandwidth savings versus cache size for the EndRE service using the Max-Match approach (averaged across all enterprise links). Although not shown, the trends are similar for the

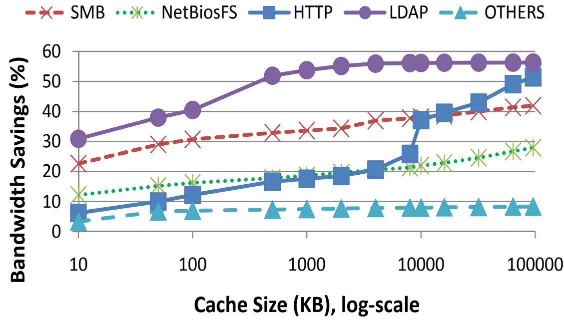
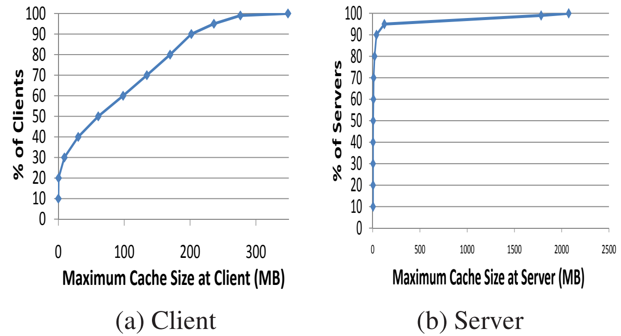


Figure 9: Cache size vs protocol bandwidth savings



(a) Client (b) Server

Figure 10: Cache scalability

Chunk-Match approach. Based on the figure, a good operating point for EndRE is at the knee of the curve corresponding to 810MB of cache, allowing for a good tradeoff between memory resource constraints and bandwidth savings.

Figure 9 plots the bandwidth savings versus cache size (in log-scale for clarity) for different protocols. For this trace set, HTTP (port 80,8080) comprised 45% of all traffic, SMB (port 445) and NetBios File sharing (port 139) together comprised 26%, LDAP (port 389) was about 2.5% and a large set of protocols, labeled as OTHERS, comprised 26.5%. While different protocols see different bandwidth savings, all protocols, except OTHERS, see savings of 20+% with LDAP seeing the highest savings of 56%. Note that OTHERS include several protocols that were encrypted (HTTPS:443, Remote Desktop:3389, SIP over SSL:5061, etc.). For this analysis, since we are estimating EndRE savings from IP-level packet traces whose payload is already encrypted, EndRE sees 0% savings. An implementation of EndRE in the socket layer would likely provide higher savings for protocols in the OTHERS category than estimated here. Finally, by examining the figure, one can see the "knee-of-the-curve" at different values of cache size for different protocols (10MB for HTTP, 4MB for SMB, 500KB for LDAP, etc.). This also confirms that the 10MB knee of Figure 8 is largely due to the 10MB knee for HTTP in Figure 9.

This analysis suggests that the cache limit could be tuned depending on the protocol(s) used between a client-server pair without significantly impacting overall bandwidth savings. Thus, *we use 10MB cache size only if HTTP traffic exists between a client-server pair, 4MB if SMB traffic exists, and a default 1MB cache size otherwise.* Finally, while this cache size limit is derived based on static analysis of the traces, we are looking at designing dynamic cache size adaptation algorithms for each client-server pair as part of future work.

8.2.2 Client and Server Memory Costs

Given the cache size limits derived in the previous section, we now address the critical question of EndRE scalability based on the *cumulative* cache needs at the client and server for all their connections. Using the entire set of network traces of ten enterprise sites (44 days, 5TB) in Table 2, we emulate the memory needs of EndRE with the above cache size limits for all clients and servers. We use a conservative memory page-out policy in the emulation: if there has been no traffic for over ten hours between a client-server pair, we assume that the respective EndRE caches at the nodes are paged to disk. For each node, we then compute the maximum in-memory cache needed for EndRE over the entire 44 days.

Figure 10(a) plots the CDF of the client's maximum EndRE memory needs for all ($\approx$ 1000) clients. We find that *the median (99 percentile) EndRE client allocates a maximum cache size of 60MB (275MB)* during its operation over the entire 44-day period. We also performed an independent study of desktop memory availability by monitoring memory availability at 1 minute intervals for 110 desktops over 1 month at one of the enterprise sites. Analyzing this data, we found that the *5, 50 and 90th percentile values of unused memory, available for use, at these enterprise desktops were 1994MB, 873MB, and 245MB, respectively*. This validates our hypothesis that desktop memory resources are typically adequately provisioned in enterprises, allowing EndRE to operate on clients without significant memory installation costs.

We now examine the size of the cache needed at the server. First, we focus on Max-Match and study the net size of the cache required across all active clients at the server. Using the same enterprise trace as above, we plot the CDF of server cache size for all the servers in the trace in Figure 10(b). From the figure, we find that the maximum cache requirement is about 2GB. If it is not feasible to add extra memory to servers, say due to cost or slot limitations, the Chunk-Match approach could be adopted instead. This would reduce the maximum cache requirement by 3X (§5).

Site	Trace Size GB	GZIP 10ms	EndRE Max-Match 10MB				EndRE Max-Match+GZIP 10MB	EndRE Chunk-Match 10MB	EndRE Max-Match + DOT 10MB	IP WAN-Opt Max-Match 2GB	IP WAN-Opt Max-Match + DOT 2GB
		% savings									
			MODP	MAXP	FIXED	SB	SB	MODP	SB	SB	SB
1	173	9	47	47	16	47	48	46	56	71	72
2	8	14	24	25	19	24	28	19	33	33	33
3	71	17	25	26	23	26	29	22	32	34	35
4	58	17	23	24	20	24	31	21	30	45	47
5	69	15	26	27	22	27	31	21	37	39	42
6	80	12	21	21	18	22	26	17	28	34	36
7	80	14	25	25	22	26	30	21	33	31	33
8	142	14	22	23	18	22	28	19	30	34	40
9	198	9	16	16	14	16	19	15	26	44	46
10	117	13	20	21	17	21	25	17	30	27	30
Avg/site	100	13	25	26	19	26	30	22	34	39	41

Table 4: Percentage bandwidth savings on incoming links to 10 enterprise sites over 11 day trace

9 Benefits

We now characterize various benefits of EndRE. We first investigate WAN bandwidth savings. We then quantify latency savings of using EndRE, especially on short transactions typical of HTTP. Finally, we quantify energy savings on mobile smartphones, contrasting EndRE with prior work on energy-aware compression [11].

9.1 Bandwidth Savings

In this section, we focus on the bandwidth savings of different RE algorithms for each of the enterprise sites, and examine the gains of augmenting EndRE with GZIP and DOT [21]. We also present bandwidth savings of an IP WAN optimizer for reference.

Table 4 compares the bandwidth savings on incoming links to ten enterprise sites for various approaches. This analysis is based on packet-level traces and while operating at packet sizes or larger buffers make little difference to the benefits of EndRE approaches, buffer size can have a significant impact on GZIP-style compression. Thus, in order to emulate the benefits of performing GZIP at the socket layer, we aggregate consecutive packet payloads for up to 10ms and use this aggregated buffer while evaluating the benefits of GZIP. For EndRE, we use cache sizes of up to 10MB. We also emulate an IP-layer middlebox-based WAN optimizer with a 2GB cache.

We observe the following: First, performing GZIP in isolation on packets aggregated for up to 10ms provides per-site savings of 13% on average. Further, there are site specific variations; in particular, GZIP performs poorly for site 1 compared to other approaches. Second, comparing the four fingerprinting algorithms (columns 3-6 in Table 4), we see that MODP, MAXP, and SAMPLEBYTE deliver similar average savings of 25-26% while FIXED under-performs. In particular, in the case of site 1, FIXED significantly under-performs the other three approaches. This again illustrates how SAMPLEBYTE captures enough content-specific characteristics to significantly outperform FIXED. Adding GZIP compression to SAMPLEBYTE improves the average savings to 30% (column 7). While the above numbers were based on Max-Match, using Chunk-Match instead reduces the savings to 22% (column 8), but this may be a reasonable alternative if server memory is a bottleneck.

We then examine savings when EndRE is augmented with DOT [21]. For this analysis, we employ a heuristic to extract object chunks from the packet traces as follows: we combine consecutive packets of the same four-tuple flow and delineate object boundaries if there is no packet within a time window(1s). In order to ensure that the DOT analysis adds redundancy not seen by EndRE, we conservatively add only inter-host redundancy obtained by DOT to the EndRE savings. We see that (third column from right) DOT improves EndRE savings by a further 6-10%, and the per-site average bandwidth savings improves to 34%. For reference, a WAN optimizer with 2GB cache provides per-site savings of 39% and if DOT is additionally applied (where redundancy of matches farther away than 2GB are only counted), the average savings goes up by only 2%. Thus, it appears that half the gap between EndRE and WAN optimizer savings comes from inter-host redundancy and the other half from the larger cache used by the WAN optimizer.

Summarizing, EndRE using the Max-Match approach with the SAMPLEBYTE algorithm provides average per-site savings of 26% and delivers two-thirds of the savings of a IP-layer WAN optimizer. When DOT is applied in conjunction, the average savings of EndRE increase to 34% and can be seen to be approaching the 41% savings of the WAN optimizer with DOT.

Pilot Deployment: We now report results from a small scale deployment. EndRE was deployed on 15 desktop/laptop clients (11 users) and one server for a period of about 1 week (09/25/09 to 10/02/09) in our lab. We also hosted a HTTP proxy at the EndRE server and users manually enabled/disabled the use of this proxy, at any given time, using a client-based software. During this period, a total of 1.7GB of HTTP traffic was delivered through the EndRE service with an average compression

RTTs	1	2	3	4	5	>5
Latency Gain	0	20%	23%	36%	20%	22%

Table 5: HTTP latency gain for different RTTs

of 31.2%. A total of 159K TCP connections were serviced with 72 peak active simultaneous TCP connections and peak throughput of 18.4Mbps (WAN link was the bottleneck). The CPU utilization at the server remained within 10% including proxy processing. The number of packet re-orderings was less than 1% even in the presence of multiple simultaneous TCP connections between client and server. We also saw a large number of TCP RSTs but, as reported in [10], these were mostly in lieu of TCP FINs and thus do not contribute to cache synchronization issues. Summarizing, even though this is a small deployment, the overall savings numbers match well with our analysis results and the ease of deployment validates the choice of implementing EndRE over TCP.

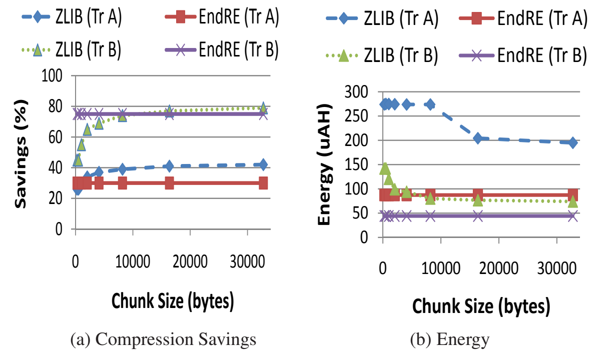
9.2 Latency Savings

In this section, we evaluate the latency gains from deploying EndRE. In general, latency gains are possible for a number of reasons. The obvious case is due to reduction of load on the bottleneck WAN access link of an enterprise. Latency gains may also arise from the choice of implementing EndRE at the socket layer above TCP. Performing RE above the TCP layer helps reduce the amount of data transferred and thus the number of TCP round-trips necessary for connection completion. In the case of large file transfers, since TCP would mostly be operating in the steady-state congestion avoidance phase, the percentage reduction in data transfer size translates into a commensurate reduction in file download latency. Thus, for large file transfers, say, using SMB or HTTP, one would expect latency gains similar to the average bandwidth gains seen earlier.

Latency gains in the case of short data transfers, typical of HTTP, are harder to estimate. This is because TCP would mostly be operating in slow-start phase and a given reduction in data transfer size could translate into a reduction of zero or more round trips depending on many factors including original data size and whether or not the reduction occurs uniformly over the data.

In order to quantify latency gains for short file transfers, we perform the following experiment. From the enterprise network traces, we extract HTTP traffic that we then categorize into a series of session files. Each session file consists of a set of timestamped operations starting with a connect, followed by a series of sends and receives (i.e., transactions), and finally a close.

The session files are then replayed on a testbed consisting of a client and a server connected by a PC-based router emulating a high bandwidth, long latency link, using the mechanism described in [13]. During the replay,



(a) Compression Savings (b) Energy

Figure 11: ZLIB vs. EndRE

strict timing is enforced at the start of each session based on the original trace; in the case of transactions, timing between the start of one transaction and the start of the next transaction is preserved as far as possible. The performance metric of interest is latency gain which is defined as the ratio of reduction in transaction time due to EndRE to transaction time without EndRE.

Table 5 shows the latency gain for HTTP for various transactions sorted by the number of round-trips in the original trace. For this trace, only 40% of HTTP transactions involved more than one round trip. For these transactions, latency gains on average ranged from 20% to 35%. These gains are comparable with the average bandwidth savings due to EndRE for this trace (∼30%), demonstrating that even short HTTP transactions see latency benefits due to RE.

9.3 Energy Savings

We study the energy and bandwidth savings achieved using EndRE on Windows Mobile smartphones and compare it against both no compression as well as prior work on energy-aware compression [11]. In [11], the authors evaluate different compression algorithms and show that ZLIB performs best in terms of energy savings on resource constrained devices for decompression. We evaluate the energy and bandwidth gains using two trace files. Traces A and B are 20MB and 15MB in size, respectively, and are based on enterprise HTTP traffic, with trace B being more compressible than trace A.

We first micro-benchmark the computational cost of decompression for ZLIB and EndRE. To do this, we load pre-compressed chunks of the traces in the mobile smartphone's memory and turn off WiFi. We then repeatedly decompress these chunks and quantify the energy cost. Figures 11(a) and (b) plot the average compression savings and energy cost of in-memory decompression for various chunk sizes, respectively. The energy measurements are obtained using a hardware-based battery power monitor [4] that is accurate to within 1mA. From these figures, we make two observations. First, as the chunk size is increased, ZLIB compression sav-

	None	ZLIB				EndRE	
	Energy uAh	Energy % savings		Byte % savings		Energy % savings	Byte % savings
	pkt	pkt	32KB	pkt	32KB	pkt	pkt
A	2038	-11	42	26	44	25	29
B	1496	-11	68	41	75	70	76

Table 6: Energy savings on a mobile smartphone

ings increase and the energy cost of decompression decreases. This implies that ZLIB is energy efficient when compressing large chunks/files. Second, the compression savings and energy costs of EndRE, as expected, are independent of chunk size. More importantly, EndRE delivers comparable compression savings as ZLIB while incurring an energy cost of 30-60% of ZLIB.

We now compare the performance of ZLIB and EndRE to the case of no compression by replaying the traces over WiFi to the mobile smartphone and performing in-memory decompression on the phone. In the case of ZLIB, we consider two cases: packet-by-packet compression and bulk compression where 32KB blocks of data are compressed at a time, the latter representing a bulk download case. After decompression, each packet is consumed in memory and not written to disk; this allows us to isolate the energy cost of communication. If the decompressed packet is written to disk or further computation is performed on the packet, the total energy consumed for all the scenarios will be correspondingly higher.

Table 6 shows energy and compression gains of using ZLIB and EndRE as compared to using no compression. We see that when ZLIB is applied on a packet-by-packet basis, even though it saves bandwidth, it results in increased energy consumption (negative energy savings). This is due to the computational overhead of ZLIB decompression. On the other hand, for larger chunk sizes, the higher compression savings coupled with lower computational overhead (Figure 11) result in good energy savings for ZLIB. In the case of EndRE, we find that the bandwidth savings directly translate into comparable energy savings for communication. This suggests that EndRE is a more energy-efficient solution for packet-by-packet compression while ZLIB, or EndRE coupled with ZLIB, work well for bulk compression.

10 Conclusion

Using extensive traces of enterprise network traffic and testbed experiments, we show that our end-host based redundancy elimination service, EndRE, provides average bandwidth gains of 26% and, in conjunction with DOT, the savings approach that provided by a WAN optimizer. Further, EndRE achieves speeds of 1.5-4Gbps, provides latency savings of up to 30% and translates bandwidth savings into comparable energy savings on mobile smartphones. In order to achieve these benefits, EndRE utilizes memory and CPU resources of end systems. For enterprise clients, we show that median memory requirements for EndRE is only 60MB. At the server end, we design mechanisms for working with reduced memory and adapting to CPU load.

Thus, we have shown that the cleaner semantics of end-to-end redundancy removal can come with considerable performance benefits and low additional costs. This makes EndRE a compelling alternative to middlebox-based approaches.

Acknowledgments. We thank our shepherd Sylvia Ratnasamy and the anonymous reviewers for their comments. Aditya Akella, Ashok Anand, and Chitra Muthukrishnan were supported in part by NSF grants CNS-0626889, CNS-0746531 and CNS-0905134, and by grants from the UW-Madison Graduate School.

References

[1] Cisco Wide Area Application Acceleration Services. `http://www.cisco.com/en/US/products/ps5680/Products_Sub_Category_Home.html`.

[2] Jenkins Hash. `http://burtleburtle.net/bob/c/lookup3.c`.

[3] Peribit Networks (Acquired by Juniper in 2005): WAN Optimization Solution. `http://www.juniper.net/`.

[4] Power Monitor, Monsoon Solutions. `http://www.msoon.com/powermonitor/powermonitor.html`.

[5] Riverbed Networks: WAN Optimization. `http://www.riverbed.com/solutions/optimize/`.

[6] Windows Filtering Platform. `http://msdn.microsoft.com/en-us/library/aa366509(V.85).aspx`.

[7] A. Anand, A. Gupta, A. Akella, S. Seshan, and S. Shenker. Packet Caches on Routers: The Implications of Universal Redundant Traffic Elimination. In *ACM SIGCOMM*, Seattle, WA, Aug. 2008.

[8] A. Anand, C. Muthukrishnan, A. Akella, and R. Ramjee. Redundant in Network Traffic: Findings and Implications. In *ACM SIGMETRICS*, Seattle, WA, June 2009.

[9] S. Annapureddy, M. J. Freedman, and D. Mazires. Shark: Scaling file servers via cooperative caching. In *NSDI*, 2005.

[10] M. Arlitt and C. Williamson. An analysis of tcp reset behavior on the internet. *ACM CCR*, 35(1), 2005.

[11] K. C. Barr and K. Asanovic. Energy-aware lossless data compression. *IEEE Transactions on Computer Systems*, 24(3):250–291, Aug 2006.

[12] F. Douglis and A. Iyengar. Application-specific delta-encoding via resemblance detection. In *USENIX*, 2003.

[13] J. Eriksson, S. Agarwal, P. Bahl, and J. Padhye. Feasibility study of mesh networks for all-wireless offices. In *MobiSys*, 2006.

[14] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for http. In *SIGCOMM*, pages 181–194, 1997.

[15] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. *SIGOPS Oper. Syst. Rev.*, 35(5), 2001.

[16] R. Pang, M. Allman, M. Bennett, J. Lee, V. Paxson, and B. Tierney. A first look at modern enterprise traffic. In *IMC*, 2005.

[17] H. Pucha, D. G. Andersen, and M. Kaminsky. Exploiting similarity for multi-source downloads using file handprints. In *NSDI*, 2007.

[18] M. Rabin. Fingerprinting by random polynomials. Technical report, Harvard University, 1981. Technical Report, TR-15-81.

[19] S. C. Rhea, K. Liang, and E. Brewer. Value-Based Web Caching. In *12th World Wide Web Conference*, 2003.

[20] N. T. Spring and D. Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *SIGCOMM*, pages 87–95, 2000.

[21] N. Tolia, M. Kaminsky, D. G. Andersen, and S. Patil. An architecture for Internet data transfer. In *NSDI*, 2006.

Cheap and Large CAMs for High Performance Data-Intensive Networked Systems

Ashok Anand[⋆]*, Chitra Muthukrishnan*[⋆]*, Steven Kappes*[⋆]*, Aditya Akella*[⋆] *and Suman Nath*[†]

[⋆]*UW-Madison,* [†]*Microsoft Research*

Abstract

We show how to build cheap and large CAMs, or *CLAMs*, using a combination of DRAM and flash memory. These are targeted at emerging data-intensive networked systems that require massive hash tables running into a hundred GB or more, with items being inserted, updated and looked up at a rapid rate. For such systems, using DRAM to maintain hash tables is quite expensive, while on-disk approaches are too slow. In contrast, CLAMs cost nearly the same as using existing on-disk approaches but offer orders of magnitude better performance. Our design leverages an efficient flash-oriented data-structure called BufferHash that significantly lowers the amortized cost of random hash insertions and updates on flash. BufferHash also supports flexible CLAM eviction policies. We prototype CLAMs using SSDs from two different vendors. We find that they can offer average insert and lookup latencies of 0.006ms and 0.06ms (for a 40% lookup success rate), respectively. We show that using our CLAM prototype significantly improves the speed and effectiveness of WAN optimizers.

1 Introduction

In recent years, a number of data-intensive networked systems have emerged where there is a need to maintain hash tables as large as tens to a few hundred gigabytes in size. Consider WAN optimizers [1, 2, 7, 8], for example, that maintain "data fingerprints" to aid in identifying and eliminating duplicate content. The fingerprints are 32-64b hashes computed over $\sim$4-8KB chunks of content. The net size of content is $\sim$10TB stored on disk [9]. Thus the hash table storing the mapping from fingerprints to on-disk addresses of data chunks could be $\geq$32GB. Just storing the fingerprints requires $\sim$16GB.

The hash tables in these content-based systems are also inserted into, looked up and updated frequently. For instance, a WAN optimizer connected to a 0.5Gbps link may require roughly 10,000 fingerprint lookups, insertions and updates each per second. Other examples of systems that employ similar large hash tables include data deduplication systems [4, 45], online backup services [5], and directory services in data-oriented network architectures [32, 37, 42]. These systems are becoming increasingly popular and being widely adopted [3].

This paper arises from the quest to design effective hash tables in these systems. The key requirement is that the mechanisms used be *cost-effective* for the functionality they support. That is, the mechanisms should offer a high number of hash table operations ($> 10K$) per second while keeping the overall cost low. We refer to mechanisms that satisfy these requirements as *CLAMs*, for cheap and large CAMs.

There are two possible approaches today for supporting the aforementioned systems. The first is to maintain hash tables in DRAM which can offer very low latencies for hash table operations. However, provisioning large amounts of DRAM is expensive. For instance, a 128GB RamSan DRAM-SSD offers 300K random IOs per second, but, it has a very high cost of ownership, including the device cost of $120K and an energy footprint of 650W [20]. In other words, it can support fewer than 2.5 *hash table operations per second per dollar*.

A much cheaper alternative is to store the hash tables on magnetic disks using database indexes, such as Berkeley-DB (or BDB) [6]. However, poor throughput of random inserts, lookups, and updates in BDB can severely undermine the effectiveness of the aforementioned systems and force them to run at low speeds. For example, a BDB-based WAN optimizer is effective for link speeds of only up to 10Mbps (§8). Note that existing fast stream databases [22, 11, 14] and wire-speed data collection systems [24, 29] are not suitable as CLAMs as they do not include any archiving and indexing schemes.

In this paper we design and evaluate an approach that is *1-2 orders of magnitude better* in terms of hash operations/sec/$ compared to *both* disk-based and DRAM-based approaches. Our approach uses a commodity two-level storage/memory hierarchy consisting of some DRAM and a much larger amount of *flash storage* (could be flash memory chips or solid state disks (SSDs)). Our design consumes most of the I/Os in the DRAM, giving low latency and high throughput I/Os compared to a flash-only design. On the other hand, using flash allows us to support a large hash table in a cheaper way than DRAM-only solutions. We choose flash over magnetic disks for its many superior properties, such as, higher I/O per second per dollar and greater reliability, as well as far superior power efficiency compared to both DRAM and magnetic disks. Newer generation of SSDs are rapidly getting bigger and cheaper. Configuring our design with 4GB of memory and 80GB of flash, for instance, costs as little as $400 using current hardware.

Despite flash's attractive I/O properties, building a CLAM using flash is challenging. In particular, since the available DRAM is limited, a large part of the hash

table must be stored in flash (unlike recent works, e.g., FAWN [13], where the hash index is fully in DRAM). Thus, hash insertion would require random I/Os, which are expensive on flash. Moreover, the granularity of a flash I/O is orders of magnitude bigger than that of an individual hash table operation in the systems we target. Thus, unless designed carefully, the CLAM could perform poorer than a traditional disk-based approach.

To address these challenges, we introduce a novel data structure, called BufferHash. BufferHash represents a careful synthesis of prior ideas along with a few novel algorithms. A key idea behind BufferHash is that instead of performing individual random insertions directly on flash, DRAM can be used to buffer multiple insertions and writes to flash can happen in a *batch*. This shares the cost of a flash I/O operation across multiple hash table operations, resulting in a better amortized cost per operation. Like a log-structured file system [39], batches are written to flash sequentially, the most efficient write pattern for flash. The idea of batching operations to amortize I/O costs has been used before in many systems [15, 28]. However using it for hash tables is novel, and it poses several challenges for flash storage.

Fast lookup: With batched writes, a given $(key, value)$ pair may reside in any prior batch, depending on the time it was written out to flash. A naive lookup algorithm would examine all batches for the key, which would incur high and potentially unacceptable flash I/O costs. To reduce the overhead of examining on-flash batches, BufferHash (i) partitions the key space to limit the lookup to one partition, instead of the entire flash (similar to how FAWN spreads lookups across multiple "wimpy nodes") [13]), and (ii) uses in-memory Bloom filters (as Hyperion [23] does) to efficiently determine a small set of batches that may contain the key.

Limited flash: In many of the streaming applications mentioned earlier, insertion of new $(key, value)$ entries into the CLAM requires creating space by evicting old keys. BufferHash uses a novel age-based internal organization that naturally supports bulk evictions of old entries in an I/O-efficient manner. BufferHash also supports other flexible eviction policies (e.g. priority-based removal) to match different application needs, albeit at additional performance cost. Existing proposals for indexing archived streaming data ignore eviction entirely.

Updates: Since flash does not support efficient update or deletion, modifying existing $(key, value)$ mappings *in situ* is expensive. To support good update latencies, we adopt a *lazy update* approach where *all* value mappings, including deleted or updated ones, are temporarily left on flash and later deleted in batch during eviction. Such lazy updates have been previously used in other contexts, such as buffer-trees [15] and lazy garbage collection in log-structured file systems [39].

Performance tuning: The unique I/O properties of flash demand careful choice of various parameters in our design of CLAMs, such as the amount of DRAM to use, and the sizes of batches and Bloom filters. Suboptimal choice of these parameters may result in poor overall CLAM performance. We model the impact of these parameters on latencies of different hash table operations and show how to select the optimal settings.

We build CLAM prototypes using SSDs from two vendors. Using extensive analysis based on a variety of workloads, we study the latencies supported in each case and compare the CLAMs against popular approaches such as using Berkeley-DB (BDB) on disk. In particular, we find that our Intel SSD-based CLAM offers an average *insert* latency of 0.006ms compared to 7ms from using BDB on disk. For a workload with 40% hit rate, the average *lookup* latency is 0.06ms for this CLAM, but 7ms for BDB. Thus, our CLAM design can yield 42 lookups/sec/$ and 420 insertions/sec/$ which is 1-2 orders of magnitude better than RamSan DRAM-SSD (2.5 hash operations/sec/$). The superior energy efficiency of flash and rapidly declining prices compared to DRAM [21] mean that the gap between our CLAM design and DRAM-based solutions is greater than indicated in our evaluation and likely to widen further. Finally, using real traces, we study the benefits of employing the CLAM prototypes in WAN optimizers. Using a CLAM, the speed of a WAN optimizer can be improved $\geq$ 10X compared to using BDB (a common choice today [2]).

Our CLAM design marks a key step in building fast and effective indexing support for high-performance content-based networked systems. We do not claim that our design is final. We speculate that there may be smarter data structures and algorithms, that perhaps leverage newer memory technologies (e.g. Phase Change Memory), offering much higher hash operations/sec/$.

2 Related Work

In this section, we describe prior work on designing data structures for flash and recent proposals for supporting data-intensive streaming networked systems.

Data structures for flash: Recent work has shown how to design efficient data structures on flash memory. Examples include MicroHash [44], a hash table and FlashDB [36], a B-Tree index. Unlike BufferHash, these data structures are designed for memory-constrained embedded devices where the design goal is to optimize energy usage and minimize memory footprint—latency is typically not a design goal. For example, a lookup operation in MicroHash may need to follow multiple pointers to locate the desired key in a chain of flash blocks and can be very slow. Other recent works on designing efficient codes for flash memory to increase its effective capacity [30, 27] are orthogonal to our work, and BufferHash

can be implemented on top of these codes.

A flash-based key-value store: Closely related to our design of CLAMs is the recent FAWN proposal [13]. FAWN-KV is a clustered key-value storage built on a large number of tiny nodes that each use embedded processors and small amounts of flash memory. There are crucial differences between our CLAM design and FAWN. First, FAWN assumes that each wimpy node can keep its hash index in DRAM. In contrast, our design targets situations where the actual hash index is bigger than available DRAM and hence part of the index needs to be stored in flash. In this sense, our design is complementary to FAWN; if the hash index in each wimpy node gets bigger than its DRAM, it can use BufferHash to organize the index. Second, being a cluster-based solution, FAWN optimizes for throughput, not for latency. As the evaluation of FAWN shows, some of the lookups can be very slow ($> 500ms$). In contrast, our design provides better worst-case latency ($< 1ms$), which is crucial for systems such as WAN optimizers. Finally, FAWN-KV does not focus on efficient eviction of indexed data.

Along similar lines is HashCache [16], a cache that can run on cheap commodity laptops. It uses an in-memory index for objects stored on disk. Our approach is complementary to HashCache, just as it is with FAWN.

DRAM-only solutions: DRAM-SSDs provide extremely fast I/Os, at the cost of high device cost and energy footprint. For example, a 128GB RamSan device can support 300K IOPS, but costs $120K$\$ and consumes 650W [20]. A cheaper alternative from Violin memory supports 200K IOPS, but still costs around $50K$\$ [40]. Our CLAM prototypes significantly outperform traditional hash tables designed in these DRAM-SSDs in terms of operations/s/$.

Large scale streaming systems: Hyperion [23] enables archival, indexing, and on-line retrieval of high-volume data streams. However, Hyperion does not suit the applications we target as it does not offer CAM-like functionality. For example, to lookup a key, Hyperion may need to examine prohibitively high volume of data resulting in a high latency. Second, it does not consider using flash storage, and hence does not aim to optimize design parameters for flash. Finally, it does not support efficient update or eviction of indexed data.

Existing data stream systems [11, 14, 22] do not support queries over archived data. StreamBase [41] supports archiving data and processing queries over past data; but the data is archived in conventional hash or B-Tree-indexed tables, both of which are slow and are suitable only for offline queries. Endace DAG [24] and CoMo [29] are designed for wire-speed data collection and archiving; but they provide no query interface. Existing DBMSs can support CAM-like functionalities. However, they are designed neither for high update and lookup rates (see [14]) nor for flash storage (see [36]).

3 Motivating Applications

In this section, we describe three networked systems that could benefit from effective mechanisms for building and maintaining large hash tables that can be written to and looked up at a very fast rate.

WAN optimization. WAN optimizers [1, 2, 8, 7] are used by enterprises and data centers to improve network utilization by looking for and suppressing redundant information in network transfers. A WAN optimizer computes fingerprints of each arriving data object and looks them up in a hash table of fingerprints found in prior content. The fingerprints are 32-64b hashes computed over ∼4-8KB data chunks. Upon finding a match, the corresponding duplicate content is removed, and the "compressed object" is transmitted to the destination, where it gets reconstructed. Fingerprints for the original object are inserted into the index to aid in future matches. The content is typically ≥10TB in net size [10]. Thus the fingerprint hash tables could be ≥32GB.

Consider a WAN optimizer connected to a heavily-loaded 500Mbps link. Roughly 10,000 content fingerprints are created per second. Depending on the implementation, three scenarios may arise during hash insertion and lookup: (1) lookups for upcoming objects are held-up until prior inserts complete, or (2) upcoming objects are transmitted without fingerprinting and lookup, or (3) insertions are aborted mid-way and upcoming objects looked up against an "incomplete index." Fast support for insertions and lookups can improve all three situations and help identify more content redundancy. In §8, we show that a BDB-based WAN optimizer can function effectively only at low speeds (10Mbps) due to BDB's poor support for random insertions and lookups, even if BDB is maintained on an Intel SSD. A CLAM-based WAN optimizer using a low-end transcend SSD that is an order of magnitude slower than an Intel SSD is highly effective even at 200-300Mbps.

Data deduplication and backup. Data deduplication [4] is the process of suppressing duplicate content from enterprise data leaving only one copy of the data to be stored for archival. Prior work suggests that data sets in de-dup systems could be roughly 8-10TB and employ 20GB indexes [4, 45].

A time-consuming activity in deduplication is merging data sets and the corresponding indexes. To merge a smaller index into a larger one, fingerprints from the latter dataset need to be looked up, and the larger index updated with any new information. We estimate that merging fingerprints into a larger index using Berkeley-DB could take as long as 2hrs. In contrast, our CLAM prototypes can help the merge finish in under 2mins. We note that a similar set of challenges arise in online backup services [5] which allow users to constantly, and in an

online fashion, update a central repository with "diffs" of the files they are editing, and to retrieve changes from any remote location on demand.

Central directory for a data-oriented network. Recent proposals argue for a new resolution infrastructure to dereference content names directly to host locations [32, 37, 42]. The names are hashes computed over chunks of content inside data objects. As new sources of data arise or as old sources leave the network, the resolution infrastructure should be updated accordingly. To support scalability, the architectures have conventionally relied on a distributed resolution mechanism based on DHTs [32, 37, 42]. However, in some deployment scenarios (e.g. a large corporation), the resolution may have to be provided by a trusted central entity. To ensure high availability and throughput for a large user-base, the centralized deployment should support fast inserts and efficient lookups of the mappings. The CLAMs we design can support such an architecture effectively.

4 Flash Storage and Hash Tables

Flash provides a non-volatile memory store with several significant benefits over typical magnetic hard disks such as fast random reads ($\ll$ 1 ms), power-efficient I/Os ($<$1 Watt), better shock resistance, etc. [33, 36]. However, because of the unique characteristics of flash storage, applications designed for flash should follow a few well-known design principles: **(P1)** Applications should avoid random writes, in-place updates, and sub-block deletions as they are significantly expensive on flash. For example, updating a single 2KB page in-place requires first erasing an entire erase block (128KB-256KB) of pages, and then writing the modified block in its entirety. As shown in [35], such operations are over two orders of magnitude slower than sequential writes, out-of-place updates, and block deletions respectively, on both flash chips and SSDs. **(P2)** Since reads and writes happen at the granularity of a flash page (or an SSD sector), an I/O of size smaller than a flash page (2KB) costs at least as much as a full-page I/O. Thus, applications should avoid small I/Os if possible. **(P3)** The high fixed initialization cost of an I/O can be amortized with a large I/O size [12]. Thus, applications should batch I/Os whenever possible. In designing flash-based CLAMs using BufferHash, we follow these design principles.

A conventional hash table on flash. Before going into the details of our BufferHash design, it might be useful to see why a conventional hash table on flash is likely to suffer from poor performance. Successive keys inserted into a hash table are likely to hash to random locations in the hash table; therefore, values written to those hashed locations will result in random writes, violating the design principle **P1** above.

Updates and deletions are immediately applied to a conventional hash table, resulting in in-place updates and sub-block deletions (since each hashed value is typically much smaller than a flash block), and violation of **P1**.

Since each hashed value is much smaller than a flash page (or an SSD sector), inserting a single key in an in-flash hash table violates principles **P2** and **P3**. Violation of these principles results in a poor performance of a conventional hash table on flash, as we demonstrate in §7.

One can try to improve the performance by buffering a part of the hash table in DRAM and keeping the remaining in flash. However, since hash operations exhibit negligible locality, such a flat partitioning has very little performance improvement. Recent research has confirmed that a memory buffer is practically useless for external hashing for a read-write mixed workload [43].

5 The BufferHash Data Structure

BufferHash is a flash-friendly data structure that supports hash table-like operations on $(key, value)$ pairs[1]. The key idea underlying BufferHash is that instead of performing individual insertions/deletions one at a time to the hash table on flash, we can perform multiple operations all at once. This way, the cost of a flash I/O operation can be shared among multiple insertions, resulting in a better amortized cost for each operation (similar to buffer trees [15] and group commits in DBMS and file systems [28]). For simplicity, we consider only insertion and lookup operations for now; we will discuss updates and deletions later.

To allow multiple insertions to be performed all at once, BufferHash operates in a lazy batched manner: it accumulates insertions in small in-memory hash tables (called *buffers*), without actually performing the insertions on flash. When a buffer fills up, all inserted items are pushed into flash in a batch. For I/O efficiency, items pushed from a buffer to flash are sequentially written as a new hash table, instead of performing expensive update to existing in-flash hash tables. Thus, at any point of time, the flash contains a large number of small hash tables. During lookup, a set of Bloom filters is used to determine which in-flash hash tables may contain the desired key, and only those tables are retrieved from flash. At a high level, the efficiency of this organization comes from batch I/O and sequential writes during insertions. Successful lookup operations may still need random page reads, however, random page reads are almost as efficient as sequential page reads in flash.

5.1 A Super Table

BufferHash consists of multiple *super tables*. Each super table has three main components: a buffer, an incarnation table, and a set of Bloom filters. These components are

[1] For clarity purposes we note that BufferHash is a data-structure while a CLAM is BufferHash applied atop DRAM and flash.

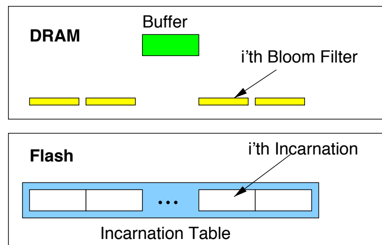


Figure 1: *A Super Table*

organized in two levels of hierarchy, as shown in Figure 1. Components in the higher level are maintained in DRAM, while those in the lower level are maintained in flash.

Buffer. This is an in-memory hash table where all newly inserted hash values are stored. The hash table can be built using existing fast algorithms such as multiple-choice hashing [18, 31]. A buffer can hold a fixed maximum number of items, determined by its size and the desired upper bound of hash collisions. When the number of items in the buffer reaches its capacity, the entire buffer is flushed to flash, after which the buffer is re-initialized for inserting new keys. The buffers flushed to flash are called *incarnations*.

Incarnation table. This is an in-flash table that contains old and flushed incarnations of the in-memory buffer. The table contains k incarnations, where k denotes the ratio of the size of the incarnation table and the buffer. The table is organized as a circular list, where a new incarnation is sequentially written at the list-head. To make space for a new incarnation, the oldest incarnation, at the tail of the circular list, is evicted from the table.

Depending on application's eviction policy, some items in an evicted incarnation may need to be retained and are re-inserted into the buffer (details in §5.1.2).

Bloom filters. Since the incarnation table contains a sequence of incarnations, the value for a given hash key may reside in any of the incarnations depending on its insertion time. A naive lookup algorithm for an item would examine all incarnations, which would require reading all incarnations from flash. To avoid this excessive I/O cost, a super table maintains a set of in-memory Bloom filters [19], one per incarnation. The Bloom filter for an incarnation is a compact signature built on the hash keys in that incarnation. To search for a particular hash key, we first test the Bloom filters for all incarnations; if any Bloom filter matches, then the corresponding incarnation is retrieved from flash and looked up for the desired key. Bloom filter-based lookups may result in false positive; thus, a match could be indicated even though there is none, resulting in unnecessary flash I/O. As the filter size increases, the false positive rate drops, resulting in lower I/O overhead. However, since the available DRAM is limited, filters cannot be too large in size. We examine the tradeoff in §6.4.

The Bloom filters are maintained as follows: When a buffer is initialized after a flush, a Bloom filter is created for it. When items are inserted into the buffer, the Bloom filter is updated with the corresponding key. When the buffer is flushed as an incarnation, the Bloom filter is saved in memory as the Bloom filter for that incarnation. Finally, when an incarnation is evicted, it's Bloom filter is discarded from memory.

5.1.1 Super Table Operations

A super table supports all standard hash table operations.

Insert. To insert a $(key, value)$ pair, the value is inserted in the hash table in the buffer. If the buffer does not have space to accommodate the key, the buffer is flushed and written as a new incarnation in the incarnation table. The incarnation table may need to evict an old incarnation to make space.

Lookup. A key is first looked up in the buffer. If found, the corresponding value is returned. Otherwise, in-flash incarnations are examined in the order of their age until the key is found. To examine an incarnation, first its Bloom filter is checked to see if the incarnation might include the key. If the Bloom filter matches, the incarnation is read from flash, and checked if it really contains the key. Note that since each incarnation is in fact a hash table, to lookup a key in an incarnation, only the relevant part of the incarnation (e.g., a flash page) can be read directly.

Update/Delete. As mentioned earlier, flash does not support small updates/deletions efficiently; hence, we support them in a lazy manner. Suppose a super table contains an item (k, v), and later, the item needs to be updated with the item (k, v'). In a traditional hash table, the item (k, v) is immediately replaced with (k, v'). If (k, v) is still in the buffer when (k, v') is inserted, we do the same. However, if (k, v) has already been written to flash, replacing (k, v) will be expensive. Hence, we simply insert (k, v') without doing anything to (k, v). Since the incarnations are examined in order of their age during lookup, if the same key is inserted with multiple updated values, the latest value (in this example, v') is returned by a lookup. Similarly, for deleting a key k, a super table does not delete the corresponding item unless it is still in the buffer; rather the deleted key is kept in a separate list (or, a small in-memory hash table), which is consulted before lookup—if the key is in the delete list, it is assumed to be deleted even though it is present in some incarnation. Lazy update wastes space on flash, as outdated items are left on flash; the space is reclaimed during incarnation eviction.

5.1.2 Incarnation Eviction

In a streaming application, BufferHash may have to evict old in-flash items to make space for new items. The de-

cision of what to evict depends on application policy.

For I/O efficiency, BufferHash evicts items in granularity of an incarnation. Since each incarnation is an independent hash table, discarding a part of it may require expensive reorganization of the table and expensive I/O to write it back to flash. To this end, BufferHash provides two basic eviction primitives. The *full discard* primitive entirely evicts the oldest incarnation. The *partial discard* primitive also evicts the oldest incarnation, but it scans through all the items in the incarnation before eviction, selects some items to be retained (based on a specified policy), and re-inserts them into the buffer. Given these two basic primitives, applications can configure BufferHash to implement different eviction policies as follows.

FIFO. The full discard primitive naturally implements the FIFO policy. Since items with similar ages (i.e., items that are flushed together from the buffer) are clustered in the same incarnation, discarding the oldest incarnation evicts the oldest items. Commercial WAN optimizers work in this fashion [8, 2].

LRU. An LRU policy can be implemented via the full discard mechanism with one additional mechanism: on every use of an item not present in the buffer, the item is re-inserted. Intuitively, a recently used item will be present in a recent incarnation, and hence it will still be present after discarding the oldest incarnation. This implementation incurs additional space overhead as the same item can be present in multiple incarnations.

Update-based eviction. With a workload with many deletes and updates, BufferHash uses the partial discard mechanism to discard items that have been deleted or updated. The former can be determined by examining the in-memory delete list, while the latter can be determined by checking the in-memory Bloom filters.

Priority-based eviction. In a priority-based policy, an item is discarded if its priority is less than a threshold (the threshold can change over time, as in [35]). It can be implemented with the partial discard primitive, where an item in the discarded incarnation is re-inserted if its current priority is above a threshold.

The FIFO policy is the most efficient, and the default policy in BufferHash. The other policies incur additional space and latency overhead due to more frequent buffer flushes and re-insertion.

Note that BufferHash may not be able to *strictly* follow an eviction policy other than FIFO if enough slow storage is not available. Suppose an item is called *live* if it is supposed to be present in the hash table under a given eviction policy (e.g., for the update-based eviction policy, the item has not been updated or deleted), and *dead* otherwise. BufferHash is supposed to evict *only* the dead items, and it does so if the flash has enough space to hold all live and unevicted dead items. On the other hand, if available flash space is limited and there are not enough dead items to evict in order to make room for newer items, BufferHash is forced to evict *live* items in a *FIFO order*. [2] We note that this sort of behavior is unavoidable in *any* storage scheme dealing with too many items to be fit in a limited amount of storage.

5.1.3 Bit-slicing with a Sliding Window

To support efficient Bloom filter lookup, we organize the Bloom filters for all incarnations within a super table in bit-sliced fashion [26]. Suppose a super table contains k incarnations, and the Bloom filter for each incarnation has m bits. We store all k Bloom filters as m k-bit slices, where the i'th slice is constructed by concatenating bit i from each of the k Bloom filters. Then, if a Bloom filter uses h hash functions, we apply them on the key x to get h bit positions in a Bloom filter, retrieve h bit slices at those positions, compute bit-wise AND of those slices. Then, the positions of 1-bits in this aggregated slice, which can be looked up from a pre-computed table, represent the incarnations that may contain the key x.

As new incarnations are added and old ones evicted, bit slices need to be updated accordingly. A naive approach would reset the left-most bits of all m bit-slices on every eviction, further increasing the cost of an eviction operation. To avoid this, we append w extra bits with every bit-slice, where w is the size of a word that can be reset to 0 with one memory operation. Within each $(k+w)$-bit-slice, a window of k bits represent the Bloom filter bits of k current incarnations, and only these bits are used during lookup. After an incarnation is evicted, the window is shifted one bit right. Since the bit falling off the window is no longer used for lookup, it can be left unchanged. When the window has shifted w bits, entire w-bit words are reset to zero at once, resulting in a small amortized cost. The window wraps around after it reaches the end of a bit-slice. For lack of space we omit the details, which can be found in [12].

5.2 Partitioned Super Tables

Maintaining a single super table is not scalable because the buffer and individual incarnations will become very large with a large available DRAM. As the entire buffer is flushed at once, the flushing operation can take a long time. Since flash I/Os are blocking operations, lookup operations that go to flash during this long flushing period will block (insertions can still happen as they go to in-memory buffer). Moreover, an entire incarnation from the incarnation table is evicted at a time, increasing the eviction cost with partial discard.

[2] With the update-based eviction policy, a live item can also be evicted if the in-memory Bloom filter incorrectly concludes that the item has been updated. However, the probability is small (equals to the false positive rate of the Bloom filters).

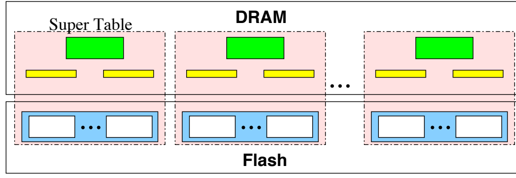


Figure 2: *A BufferHash with multiple super tables*

BufferHash avoids this problem by partitioning the hash key space and maintaining one super table for each partition (Figure 2): Suppose each hash key has $k = k_1 + k_2$ bits; then, BufferHash maintains 2^{k_1} super tables. The first k_1 bits of a key represents the index of the super table containing the key, while the last k_2 bits are used as the key within the particular super table.

Partitioning enables using small buffers in super tables, thus avoiding the problems caused by a large buffer. However, we show in §6.4 that too many partitions (i.e., very small buffers) can also adversely affect performance. We show how to choose the number of partitions for good performance. For example, we show for flash chips that the number of partitions should be such that the size of a buffer matches the flash block size.

BufferHash with multiple super tables can be implemented on a flash chip by statically partitioning it and allocating each partition to a super table. A super table writes its incarnations in its partition in a circular way—after the last block of the partition is written, the first block of the partition is erased and the corresponding incarnations are evicted. However, this approach may not be optimal for an SSD, where a Flash Translation Layer (FTL) hides the underlying flash chips. Even though writes within a single partition are sequential, writes from different super tables to different partitions may be interleaved, resulting in a performance worse than a single sequential write (see [17] for empirical results). To deal with that, BufferHash uses the entire SSD as a single circular list and writes incarnations from different super tables sequentially, in the order they are flushed to the flash. (This is in contrast to the log rotation approach of Hyperion [23] that provides FIFO semantics for each partition, instead of the entire key space.) Note that partitioning also naturally supports using multiple SSDs in parallel, by distributing partitions to different SSDs. This scheme, however, spreads the incarnations of a super table all over the SSD. To locate incarnations for a given super table, we maintain their flash addresses along with their Bloom filters and use the addresses during lookup.

6 Analysis of Costs

In this section, we first analyze the I/O costs of insertion and lookup operations in CLAMs built using BufferHash for flash-based storage, and then use the analytical results to determine optimal values of two important parameters of BufferHash. We use the notations in Table 1.

Symbol	Meaning
N	Total number of items inserted
M	Total memory size
B	Total size of buffers
b	Total size of Bloom filters
k	Number of incarnations in a super table
F	Total flash size
s	Average size taken by a hash entry
h	Number of hash functions
B'	Size of a single buffer (=B/n)
S_p	Size of a flash page/sector
S_b	Size of a flash block

Table 1: *Notations used in cost analysis.*

6.1 Insertion Cost

We now analyze the amortized and the worst case cost of an insertion operation. We assume that BufferHash is maintained on a flash chip; later we show how the results can be trivially extended to SSDs. Based on empirical results [12], we use linear cost functions for flash I/Os—reading, writing, and erasing x bits, at appropriate granularities, cost $a_r + b_r x$, $a_w + b_w x$, and $a_e + b_e x$, respectively.

Consider a workload of inserting N keys. Most insertions are consumed in buffers, and hence do not need any I/O. However, expensive flash I/O occurs when a buffer fills and is flushed to flash. Each flush operation involves three different types of I/O costs. First, each flush requires writing $n_i = \lceil B'/S_p \rceil$ pages, where B' is the size of a buffer in a super table, and S_p is the size of a flash page (or an SSD sector). This results in a write cost of

$$C_1 = a_w + b_w n_i S_p$$

Second, each flush operation requires evicting an old incarnation from the incarnation table. For simplicity, we consider full discard policy for an evicted incarnation. Note that each incarnation occupies $n_i = \lceil B'/S_p \rceil$ flash pages, and each flash block has $n_b = S_b/S_p$ pages, where S_b is the size of a flash block. If $n_i \geq n_b$, every flush will require erasing flash blocks; otherwise, only n_i/n_b fraction of the flushes will require erasing blocks. Finally, during each erase, we need to erase $\lceil n_i/n_b \rceil$ flash blocks. Putting all together, we get the erase cost of a single flush operation as

$$C_2 = \text{Min}(1, n_i/n_b)(a_e + b_e \lceil n_i/n_b \rceil S_b)$$

Finally, a flash block to be erased may contain valid pages (from other incarnations), which must be backed up before erase and copied back after erase. This can happen because flash can be erased only at the granularity of a block and an incarnation to be evicted may occupy only part of a block. In this case, $p' = (n_b - n_i) \mod n_b$ pages must be read and written during each flush. This results in a copying cost of

$$C_3 = a_r + p' b_r S_p + a_w + p' b_w S_p$$

Amortized cost. Consider insertion of N keys. If each hash entry occupies a space of s, each buffer can

hold B'/s entries, and hence buffers will be flushed to flash a total of $n_f = Ns/B'$ times. Thus, the amortized insertion cost is

$$C_{amortized} = n_f(C_1+C_2+C_3)/N = (C_1+C_2+C_3)s/B'$$

Note that the cost is independent of N and inversely proportional to the buffer size B'.

Worst case cost. An insert operation experiences the worst-case performance when the buffer for the key is full, and hence must be flushed. Thus, the worst case cost of an insert operation is

$$C_{worst} = C_1 + C_2 + C_3$$

SSD. The above analysis extends to SSDs. Since the costs C_2 and C_3 in an SSD are handled by its FTL, the overheads of erasing blocks and copying valid pages are reflected in its write cost parameters a_w and b_w. Hence, for an SSD, we can ignore the cost of C_2 and C_3. Thus, we get: $C_{amortized} = C_1 s/B'$ and $C_{worst} = C_1$.

6.2 Lookup Cost

A lookup operation in a super table involves first checking the buffer for the key, checking the Bloom filters to determine which incarnations may contain the key, and reading a flash page for each of those incarnations to actually lookup the key. Since a Bloom filter may produce false positives, some of these incarnations may not contain the key, and hence some of the I/Os may redundant.

Suppose BufferHash contains n_t super tables. Then, each super table will have $B' = B/n_t$ bits for its buffer, and $b' = b/n_t$ bits for Bloom filters. In steady state, each super table will contain $k = (F/n_t)/(B/n_t) = F/B$ incarnations. Each incarnation contains $n' = B'/s$ entries, and a Bloom filter for an incarnation will have $m' = b'/k$ bits. For a given m' and n', the false positive rate of a Bloom filter is minimized with $h = m' \ln 2/n'$ hash functions [19]. Thus, the probability that a Bloom filter will return a hit (i.e., indicating the presence of a given key) is given by $p = (1/2)^h$. For each hit, we need to read a flash page. Since there are c incarnations, the expected flash I/O cost is given by

$$\begin{aligned} C_{lookup} &= kpc_r = k(1/2)^h c_r \\ &= F/B(1/2)^{bs \ln 2/F} c_r \end{aligned}$$

where c_r is the cost of reading a single flash page from a flash chip, or a single sector from an SSD.

6.3 Discussion

The above analysis can provide insights into benefits and overheads of various BufferHash components that are not used in traditional hash tables. Consider a traditional hash table stored on an SSD; without any buffering, each insertion operation would require one random sector write. Suppose, sequentially writing a buffer of size B' is α times more expensive than randomly writing one sector of an SSD. α is typically small even for a buffer significantly bigger than a sector, mainly due to two reasons. First, sequential writes are significantly cheaper than random writes in most existing SSDs. Second, writing multiple consecutive sectors in a batch has better per sector latency. In fact, for many existing SSDs, the value of α is less than 1 even for a buffer size of $256KB$ (e.g., 0.39 and 0.36 for Samsung and MTron SSDs respectively). For Intel SSD, the gap between sequential and random writes is small; still the value of α is less than 10 due to I/O batching.

Clearly, the worst case insertion cost into a CLAM using BufferHash for flash is α times more expensive than that of a traditional hash table without buffering—a traditional hash table requires writing a random sector, while BufferHash sequentially writes the entire buffer. As discussed above, the value of α is small for existing SSDs. On the other hand, our previous analysis shows that the amortized insertion cost of BufferHash on flash is at least $\frac{B'}{\alpha s}$ times less than a traditional hash table, even if we assume random writes required by traditional hash table are as cheap as sequential writes required by BufferHash on flash. In practice, random writes are more expensive, and therefore, the amortized insertion cost when using BufferHash on flash is even more cheap than that of a traditional hash table.

Similarly, a traditional hash table on flash will need one read operation for each lookup operation, even for unsuccessful lookups. In contrast, the use of Bloom filter can significantly reduce the number of flash reads for unsuccessful lookups. More precisely, if the Bloom filters are configured to provide a false positive rate of p, use of Bloom filter can reduce the cost of an unsuccessful lookup by a factor of $1/p$. Note that the same benefit can be realized by using Bloom filters with a traditional hash table as well. Even though BufferHash maintains multiple Bloom filters over different partitions and incarnations, the total size of all Bloom filters will be the same as the size of a single Bloom filter computed over all items. This is because for a given false positive rate, the size of a Bloom filter is proportional to the number of unique items in the filter,

6.4 Parameter Tuning

Tuning BufferHash for good CLAM performance requires tuning two key parameters. First, one needs to decide how much DRAM to use, and if a large enough DRAM is available, how much of it is to allocate for buffer and how much to allocate for Bloom filters. Second, once the total size of in-memory buffers is decided, one needs to decide how many super tables to use. We use the cost analysis above to address these issues.

Optimal buffer size. Assume that the total memory size is M bits, of which B bits are allocated for (all) buffers (in all super tables) and $b = M - B$ bits are allocated for Bloom filters. Our previous analysis shows that the value of B does not directly affect insertion cost; however, it affects lookup cost. So, we would like to find the optimal value of B that minimizes the expected lookup cost.

Intuitively, the size of a buffer poses a tradeoff between the total number of incarnations and the probability of an incarnation being read from flash during lookup. As our previous analysis showed, the I/O cost is proportional to the product of the number of incarnations and the hit rate of Bloom filters. On one hand, reducing buffer size increases the number of incarnations, increasing the cost. On the other hand, increasing buffer size leaves less memory for Bloom filters, which increases its false positive rate and I/O cost.

We can use our previous analysis to find a sweet-spot. Our analysis showed that the lookup cost is given by $C = F/B \cdot (1/2)^{(M-B)s\ln 2/F} \cdot c_r$. The cost C is minimized when $dC/dB = 0$, or, equivalently $d(log_2(C))/dB = 0$. Solving this equation gives the optimal value of B as,

$$B_{opt} = \frac{F}{s(\ln 2)^2} \approx \frac{2F}{s}$$

Interestingly, this optimal value of B does not depend on M; rather, it depends only on the total size F of flash and the average space s taken by each hashed item. Thus, given some memory of size $M > B$, we should use $\approx 2F/s$ bits for buffers, and the remaining for Bloom filters. If additional memory is available, that should be used only for Bloom filters, and not for the buffers.

Total memory size. We can also determine how much total memory to use for BufferHash. Intuitively, using more memory improves lookup performance, as this allows using larger Bloom filters and lowering false positive rates. Suppose, we want to limit the I/O overhead due to false positives to C_{target}. Then, we can determine b', the required size of Bloom filters as follows.

$$\begin{aligned} C_{target} &\geq \frac{F}{B}\left(\frac{1}{2}\right)^{b's\ln 2/F} \cdot c_r \\ b' &\geq \frac{F}{s(\ln 2)^2}\ln\left(\frac{s(\ln 2)^2 c_r}{C_{target}}\right) \end{aligned}$$

Figure 3 shows required size of a Bloom filter for different expected I/O overheads. As the graph shows, the benefit of using large Bloom filter diminishes after a certain size. For example, for BufferHash with 32GB flash and 16 bytes per entry (effective size of 32 bytes per entry for 50% utilization of hashtables), allocating 1GB for all Bloom filters is sufficient to limit the expected I/O overhead C_{target} below 1ms.

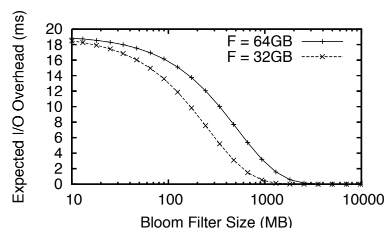


Figure 3: *Expected I/O overhead vs Bloom filter size*

Hence, in order to limit I/O overhead during lookup to C_{target}, BufferHash requires $(B_{opt} + b')$ bits of memory, of which B_{opt} is for buffers and the rest for Bloom filters.

Number of super tables. Given a fixed memory size B for all buffers, the number of super tables determines the size B' of a buffer within a super table. As our analysis shows, B' does not affect the lookup cost; rather, it affects the amortized and worst case cost of insertion. Thus, B' should be set to minimize insertion cost.

Figure 4 shows the insertion cost of using BufferHash, based on our previous analysis, on two flash-based media. (The SSD performs better because it uses multiple flash chips in parallel.). For the flash chip, both amortized and worst-case cost minimize when the buffer size B' matches the flash *block size*. The situation is slightly different for SSDs; as Figure 4(b) and (c) show, a large buffer reduces average latency but increases worst case latency. An application should use its tolerance for average- and worst-case latencies and our analytical results to determine the desired size of B' and the number of super tables B/B'.

7 Implementation and Evaluation

In this section, we measure and dissect the performance of various hash table operations in our CLAM design under a variety of workloads. Our goal is to answer the following key questions:

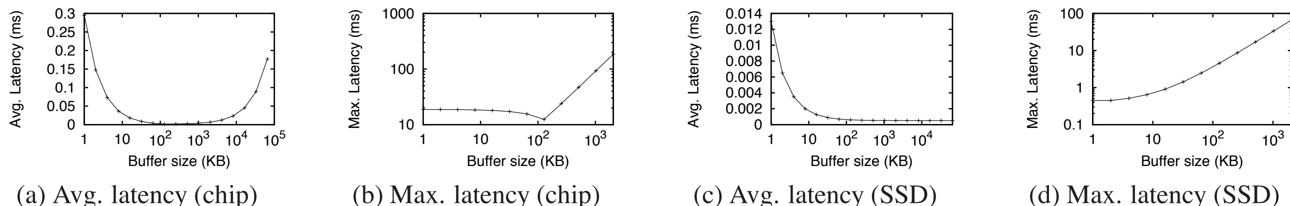
(*i*) What is the baseline performance of lookups and inserts in our design? How does the performance compare against existing disk-based indexes (e.g., the popular Berkeley-DB)? Are there specific workloads that our approach is best suited for?

(*ii*) To what extent do different optimizations in our design – namely, buffering of writes, use of bloom filters and use of bit-slicing – contribute towards our CLAM's overall performance?

(*iii*) To what extent does the use of flash-based secondary storage contribute to the performance?

(*iv*) How well does our design support a variety of hash table eviction policies?

We start by describing our implementation and how we configure it for our experiments.



(a) Avg. latency (chip) (b) Max. latency (chip) (c) Avg. latency (SSD) (d) Max. latency (SSD)

Figure 4: *Amortized and worst-case insertion cost on a flash chip and an Intel SSD. Only flash I/O costs are shown.*

7.1 Implementation and Configuration

We have implemented BufferHash in ∼3000 lines of C++ code. The hash table in a buffer is implemented using Cuckoo hashing [25] with two hash functions. Cuckoo hashing utilizes space efficiently and avoids the need for hash chaining in the case of collisions.

To simplify implementation, each partition is maintained in a separate file with all its incarnations. A new incarnation is written by overwriting the portion of file corresponding to the oldest incarnation in its super table. Thus, the performance numbers we report include small overheads imposed by the `ext3` file system we use. One can achieve better performance by writing directly to the disk as a raw device, bypassing the file system.

We run the BufferHash implementation atop two different SSDs: an Intel SSD (model: X18-M, which represents a new generation SSD), and a Transcend SSD (model: TS32GSSD25, which represents a relatively old generation but cheaper SSD).

7.1.1 Configuring the CLAM

As mentioned in §3, our key motivating applications like WAN optimization and deduplication employ hash tables of size 16-32GB. To match this, we configure our CLAMs with 32GB of slow storage and 4GB of DRAM. The size of a buffer in a super table is set to 128KB, as suggested by our analysis in §6.4. We limit the utilization of the hash table in a buffer to 50% as a higher utilization increases hash collision and the possibility of re-building the hash table for cuckoo hashing. Also, each hash entry takes 16 bytes of space. Thus, each buffer (and each incarnation) contains 4096 hash entries.

According to the analysis in §6.4, the optimal size of buffers for the above configuration is 266MB. We now experimentally validate this. Figure 5 shows the variation of false positive rates as the memory allocated to buffers is varied from 128KB to 3072MB in our prototype. The overall trend is similar to that shown by our analysis in §6.4, with the optimal spurious rate of 0.0001 occurring at a 256MB net size of buffers. The small difference from our analytically-derived optimal of 266MB arises because our analysis does not restrict the optimal number of hash functions to be an integer.

Note that the spurious rate is low even at 2GB (0.01). We select this configuration – 2GB for buffers and 32GB for slow storage – as the candidate configuration for the rest of our experiments. This gives us 16 incarnations per buffer and total of 16,384 buffers in memory.

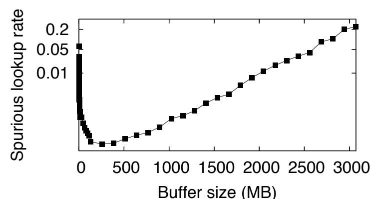


Figure 5: *Spurious rate vs. memory allocated to buffers; BufferHash configured with 4GB RAM and 32GB SSD.*

# of flash I/O	Probability		Latency (ms)	
	0% LSR	40% LSR	Flash chip	Intel SSD
0	0.9899	0.6032	0	0
1	0.0094	0.3894	0.24	0.31
2	0.0005	0.0073	0.48	0.62
3	0.00005	0.00003	0.72	0.93

Table 2: *A deeper look at lookup latencies.*

7.2 Lookups and Inserts

We start by considering the performance of basic hash table operations, namely lookups and inserts. We study other operations such as updates in §7.4.

We use synthetic workloads to understand the performance. Each synthetic workload consists of a sequence of lookups and insertions of keys. For simplicity, we focus on a single workload for the most part. In this workload, every key is first looked up, and then inserted. The keys are generated using random distribution with varying range; the range effects the lookup success rate (or, "LSR") of a key. These workloads are motivated by the WAN optimization application discussed in §8. We also consider other workloads with different ratio of insert and lookup operations in §7.2.3. Further, in order to stress-test our CLAM design, we assume that keys arrive in a continuous backlogged fashion in each workload.

7.2.1 Latencies

In Figure 6(a), we show the distribution of latencies for lookup operations on our CLAM with both an Intel and a Transcend SSD (the curves labeled **BH + SSD**) . This workload has an LSR of approximately 40%. Around 62% of the time, the lookups take little time ($<$ 0.02ms) for both Intel and Transcend SSD, as they are served by either the in-memory bloom filters or in-memory buffers. 99.8% of the lookup times are less than 0.176ms for the

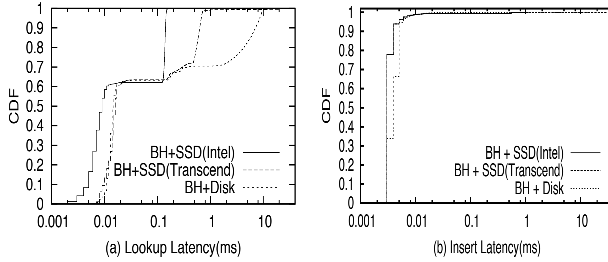


Figure 6: *CLAM latencies on different media.*

Intel SSD. For Transcend SSD, 90% of the lookup times are under 0.6ms, and the maximum is 1ms. The Intel SSD offers significantly better performance than Transcend SSD.

To understand the lookup latencies better, we examine the flash I/Os required by a lookup operation in our CLAM prototypes, under two different lookup success ratios in Table 2. Most lookups go to slow storage only when required, i.e., key is on slow storage (which happens in 0% of cases for LSR = 0 and in slightly under 40% of cases for LSR = 0.4); spurious flash I/O may be needed in the rare case of false positives (recall that BufferHash is configured for 0.01 false positive rate). Nevertheless, $>$ 99% lookups require at most one flash read only.

In Figure 6(b), we show the latencies for insertions on different CLAM prototypes. Since BufferHash buffers writes in memory before writing to flash, most insert operations are done in memory. Thus, the average insert cost is very small (0.006 ms and 0.007 ms for Intel and Transcend SSDs respectively). Since a buffer holds around 4096 items, only 1 out of 4096 insertions on average requires writing to flash. The worst case latency, when a buffer is flushed to the SSD and requires erasing a block, is 2.72 ms and 30 ms for Intel and Transcend SSDs, respectively.

On the whole, we note that our CLAM design achieves good lookup and insert performance by reducing unsuccessful lookups in slow storage and by batching multiple insertions into one big write.

7.2.2 Comparison with DB-Indexes

We now compare our CLAM prototypes against the hash table structure in Berkeley-DB (BDB) [6], a popular database index. We use the same workload as above. (We also considered the B-Tree index of BDB, but the performance was worse than the hash table. Results are omitted for brevity.) We consider the following system configurations: (1) **DB+SSD**: BDB running on an SSD, with BDB recommended configurations for SSDs, and (2) **DB+Disk**: BDB running on a magnetic disk.

Figures 7(a) and (b) show the lookup and the insert latencies for the two systems. The average lookup and insert latencies for **DB+Disk** are 6.8 ms and 7 ms respectively. More than 60% of the lookups and more than 40% of the inserts have latencies greater than 5 ms, corresponding to high seek cost on disks. Surprisingly, for the Intel SSD, the average lookup and insert latencies are also high – 4.6 ms and 4.8 ms respectively. Around 40% of lookups and 40% inserts have latencies greater than 5 ms! This is counterintuitive given that Intel SSD has significantly faster random I/O latency (0.15ms) than magnetic disks. This is explained by the fact that the low latency of an SSD is achieved only when the write load on the SSD is "low"; i.e., there are sufficient pauses between bursts of writes so that the SSD has enough time to clean dirty blocks to produce erased blocks for new writes [17]. Under a high write rate, the SSD quickly uses up its pool of erased blocks and then I/Os block until the SSD has reclaimed enough space from dirty blocks via garbage collection.

This result shows that existing disk based solutions that send all I/O requests to disks are not likely to perform well on SSDs, even if SSDs are significantly faster than disks (i.e., for workloads that give SSDs sufficient time for garbage collection). In other words, these solutions are not likely to exploit the performance benefit of SSDs under "high" write load. In contrast, since BufferHash writes to flash only when a buffer fills up, it poses a relatively "light" load on SSD, resulting in faster reads.

We do note that it is possible to supplement the BDB index with an in-memory Bloom filter to improve lookups. We anticipate that, on disks, a BDB with in-memory Bloom filter will have similar lookup latencies as a BufferHash. However, on SSDs, a BufferHash is likely to have a better lookup performance—because of the lack of buffering, insertions in BDB will incur a large number of small writes, which adversely affect SSDs' read performance due to fragmentation and background garbage collection activities.

7.2.3 Other Workloads

We evaluate how our CLAM design performs with workloads having different relative fractions of inserts and lookups. Our goal is to understand the workloads where the benefits of our design are the most significant. Table 3 shows the variation of the latency per operation with different lookup fractions in the workload for **BH+SSD** and **DB+SSD** on Transcend-SSD.

As Table 3 shows, the latency per operation for BDB decreases with increasing fraction of lookups. This is due to two reasons. First, (random) reads are significantly cheaper than random writes in SSDs. Since the increasing lookup fraction increases the overall fraction of flash reads, it reduces the overall latency per operation. Second, even the latency of individual lookups decreases with increasing fraction of lookups (not shown in the table). This is because, with a smaller fraction of flash

Fraction of lookups	Latency per operation (ms)	
	Bufferhash	Berkeley DB
0	0.007	18.4
0.3	0.01	13.5
0.5	0.09	10.3
0.7	0.11	5.3
1	0.12	0.3

Table 3: *Per-operation latencies with different lookup fractions in workloads. LSR=0.4 for all workloads and Transcend SSD is employed.*

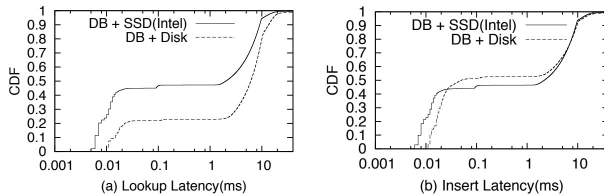


Figure 7: *Berkeley-DB latencies for lookups and inserts.*

write I/O, SSDs involve less garbage collection overhead that could interfere with all flash I/Os in general.

In contrast, for a CLAM, the latency per operation improves with decreasing fraction of lookups. This can be attributed to buffering due to which the average insert latency for a CLAM is reduced. As Table 3 shows, our CLAM design is 17× faster for write-intensive workloads than for read-intensive workloads.

7.3 Dissecting Performance Benefits

In what follows, we examine the contribution of different aspects of our CLAM design on the overall performance benefits it offers.

7.3.1 Contribution of BufferHash optimizations

The performance of our CLAM comes from three main optimizations within BufferHash: (1) buffering of inserts, (2) using Bloom filters, and (3) using windowed bit-slicing. To understand how much each of these optimizations contributes towards CLAM's performance, we evaluate our Intel SSD-based CLAM without one of these optimizations at a time.

The effect of buffering is obvious; without it, all insertions go to the flash, yielding an average insertion latency of ∼4.8ms at high insert rate i.e. continuous key insertions (compared to ∼0.006ms with buffering). Even at low insert rate, average insertion latency is ∼0.3ms and thus buffering gives significant benefits.

Without Bloom filters, each lookup operation needs to check many incarnations until the key is found or all incarnations have been checked. Since checking an incarnation involves a flash read, this makes lookups slower. The worst case happens with 0% redundancy, in which case each lookup needs to check all 16 incarnations. Our experiments show that even for 40% and 80% LSR, the average flash I/O latencies are 1.95ms and 1.5ms respectively without using Bloom filters. In contrast, using Bloom filters avoids expensive flash I/O, reducing flash I/O costs to 0.06ms and 0.13ms for 40% and 80% LSR respectively and giving a speedup of 10-30×.

Bit-slicing improves lookup latencies by ∼20% under low LSR, where the lookup workload is mostly memory bound. However, the benefit of using bit-slicing becomes negligible under a high LSR, since the lookup latency is then dominated by flash I/O latency.

7.3.2 Contribution of Flash-based Storage

The design of BufferHash is targeted specifically toward flash-based storage. In this section, we evaluate the contribution of the I/O properties of flash-based storage to the overall performance of our CLAM design. To aid in this, we compare two CLAM designs: (1) **BH+SSD**: BufferHash running on an SSD and (2) **BH+Disk**: BufferHash running on a magnetic disk (Hitachi Deskstar 7K80 drive). We use a workload with 40% look-up success rate over random keys with interleaved inserts and lookups.

Figures 6(a) and (b) also show the latencies for lookups and inserts in **BH+Disk**. Lookup latencies range from 0.1 to 12ms, an order of magnitude worse than the SSD prototypes (**BH+SSD**) due to the high seek latencies in disks. The average insert cost is very small and the worst case insert cost is 12ms, corresponding to a high seek latency for disk. Thus, the use of SSD contributes to the overall high performance of CLAM.

Comparing Figures 6 and 7, we see that **BH+Disk** performs better than **DB+SSD** and **DB+Disk** on both lookups and inserts. This shows that while using SSDs is important, it is not sufficient for high performance. It is crucial to employ BufferHash to best leverage the I/O properties of the SSDs.

7.4 Eviction Policies

Our experiments so far are based on the default FIFO eviction policy of BufferHash which we implement using the full discard primitive (§5.1.2). As stated earlier, the design of BufferHash is ideally suited for this policy. We now consider other eviction policies.

LRU. We implemented LRU using the full discard primitive as noted in §5.1.2. Omitting the details of our evaluation in the interest of brevity, we note that the performance of lookups was largely unaffected compared to FIFO; this is because the "re-insertion" operations that help emulate LRU happen asynchronously without blocking lookups (§5.1.2). In the case of inserts, the in-memory buffers get filled faster due to re-insertions, causing flushes to slower storage to become more frequent. The resulting increase in average insert latency, however, is very small: with a 40%-LSR workload having equal fractions of lookups and inserts, the average insert latency increases from 0.007ms to 0.008ms on Transcend SSD.

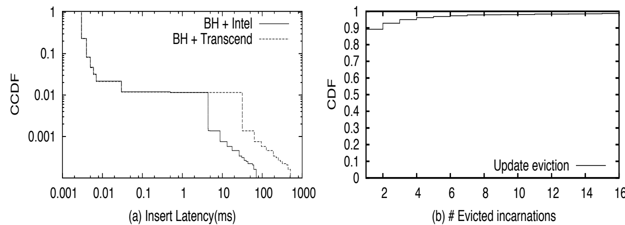


Figure 8: *(a) CCDF of insert latencies for the update-based policy. Both axes are in log-scale. (b) CDF of the number of incarnations tried upon a buffer flush. In both cases, the workload has 40% LSR and equal fractions of inserts and lookups.*

Partial discard. We now consider the two partial discard policies discussed in §5.1.2: the *update-based policy*, where only the stale entries are discarded, and the *priority-based policy*, where entries with priority lower than a threshold are discarded. We use a workload of 40% update rate using keys generated by random distribution. Figure 8(a) shows the CCDF of insert latencies on Transcend and Intel SSDs for the update-based policy. We note that an overwhelming fraction of the latencies remain unchanged, but the rest of the latencies (1%) worsen significantly. On the whole, this causes the average insertion cost to increase significantly to 0.56ms on Transcend SSD and 0.08ms on an Intel SSD. Nevertheless, this is still an order or magnitude smaller than the average latency when employing BerkeleyDB on SSDs. For priority-based policy, we used priority values equally distributed over keys. With different thresolds, we obtained similar qualitative performance (results omitted for brevity).

Three factors contribute to the higher latency observed in the tail of the distribution above: (1) When a buffer is flushed to slow storage, there is the additional cost of reading entries from the oldest incarnation and finding entries to be retained. (2) In the worst case, all entries in the evicted incarnation may have to be retained (for example, in the update-based policy this happens when none of the entries in the evicted incarnation have been updated or deleted). In that case, the in-memory buffer becomes full and is again flushed causing an eviction of the next oldest incarnation. These "cascaded evictions" continue until some entries in an evicted incarnation can be discarded, or all incarnations have been tried. In the latter case, all entries of the oldest incarnation are discarded. Cascaded evictions contribute to the high insertion cost seen in the tail of the distribution in Figure 8(a). (3) Since some of the entries are being retained after eviction, the buffer starts filling up more frequently and number of flushes to slow storage increases as a result.

We find that when buffer becomes full on a key insertion and needs to be flushed, the overall additional cost of insertion operations on Transcend SSD is 17.4ms (on average) for the update-based policy. Of this, the additional cost arising from reading and checking the entries of each incarnation is 1.62ms. Note that this is the only additional cost incurred when an incarnation eviction does not result in cascaded evictions. For priority-based policy, this cost is lower – 1.48 ms. The update-based policy is more expensive as it needs to search Bloom filters to see if an entry has already been updated.

We further find that only on rare occasions do cascaded evictions result in multiple incarnations getting accessed. In almost 90% of the cases where cascades happen, no more than 3 incarnations are tried as shown in Figure 8(b). On average, just 1.5 incarnations are tried (i.e., 0.5 incarnations are additionally flushed to slow storage, on average).

Thus, our approach supports FIFO and LRU eviction well, but it imposes a substantially higher cost for a small fraction of requests in other general eviction policies. The high cost can be controlled by loosening the semantics of the partial discard policies in order to limit cascaded evictions. For instance, applications using the priority-based policy could retain the top-k high priority entries rather than using a fixed threshold on priority. It is up to the application designer to select the right tradeoff between the semantics of the eviction policies and the additional overhead incurred.

7.5 Evaluation Summary

The above evaluation highlights the following aspects of our CLAM design:

(1) BufferHash on Intel SSD offers lookup latency of 0.06 ms and insert latency of 0.006 ms, and gives an order of magnitude improvement over Berkeley DB on Intel SSD.

(2) Buffering of writes significantly improves insert latency. Bloom filters significantly reduce unwanted lookups on slow storage, achieving $10\times$-$30\times$ improvement over BufferHash without bloom filters. Bit-slicing contributes 20% improvement when the lookup workload is mostly memory bound.

(3) For lookups, BufferHash on SSDs is an order of magnitude better than BufferHash on disk. However, SSDs alone are not sufficient to give high performance.

8 WAN Optimizer Using CLAM

In this section, we study the benefits of using our CLAM prototypes in an important application scenario, namely, WAN optimization.

A typical WAN optimizer has three components:

(1) Connection management (CM) front-end: When bytes from a connection arrive at the connection management front-end, they are accumulated into buffers for a short amount of time (we use 25ms). The buffered object

data is divided into chunks by computing content-based chunk boundaries using Rabin-Karp fingerprints [38, 34]. A SHA-1 hash is computed for each chunk thus identified.

(2) Compression engine (CE): The CE maintains a large content cache on a magnetic disk. SHA-1 fingerprints of cached content are stored in a large hash table. Fingerprints handed over by the CM are looked up in the hash table to identify similarity against prior content chunks. After redundancy has been identified, the incoming object is compressed and handed over to the network subsystem (described next). The object's chunks are inserted into the content cache in a serial fashion, and SHA-1 hashes for its chunks are inserted into the hash table with pointers to the on-disk addresses of corresponding chunks.

The CE's hash table can be stored either in a CLAM or using BDB on flash. The CLAM is configured with 4GB RAM and 32GB of Transcend SSD. The CLAM implements the full BufferHash functionality, including lazy updates with FIFO eviction as well as windowed bit slicing. For BDB-based WAN optimizer, we implement FIFO eviction from the hash table by maintaining an in-memory delete list of invalidated old hash table entries. The BDB hash table is also 32GB in size.

(3) Network sub-system (NS): The NS simply transmits the bytes handed over by CE over the outgoing network link. In commercial WAN optimizers, the NS uses an optimized custom TCP implementation that can send data at the highest possible rate (without needing repeated slow start, congestion avoidance etc.)

In order to focus on the efficacy of CE, we employ two simplifications in our evaluation: (1) We emulate a high-speed CM by pre-computing chunks and SHA-1 fingerprints for objects. (2) To emulate TCP optimization in NS, we simply use UDP to transmit data at close to link speed and turn off flow and control congestion control.

In our experiments, we vary the WAN link speed from 10Mbps to 0.5Gbps.

Evaluation: We use real packet traces in our evaluation. These traces were collected at University of Wisconsin-Madison's access link to the Internet and at the access link of a high volume Web server in the university. From these packet traces we construct object-level traces by grouping packets with the same connection 4-tuple into a single object and using an inactivity timeout of 10s. We also conducted thorough evaluation using a variety of synthetic traces where we varied the redundancy fraction. We omit the results for brevity and note that they are qualitatively similar.

Scenarios. We study two scenarios both based on replaying traces against our experimental setup:

(1) *Throughput test:* All objects arrive at once. We then measure the total time taken to transmit the objects with and without using our WAN optimizer. The ratio of the latter to the former measures the extent to which the WAN optimizer helps improve effective capacity of the attached WAN link, and we refer to it as the *effective bandwidth improvement*.

(2) *Acceleration under high load:* Here, objects arrive at a rate matching the link speed; thus, the link is 100% utilized when there is no compression. For each object, we measure the time difference from object arrival to the last byte of the object being sent, with and without WAN optimization. In either case, we also measure the throughput the object achieves (= effective size/time difference). When WAN optimization is used, the time difference includes the time to fingerprint the object, look for matches and compress the object. In addition, it may include delays due to earlier objects (e.g., updating the index with fingerprints for the earlier object). Finally, we measure the *per object throughput improvement* as the ratio of an object's throughput with and without the WAN optimizer.

8.1 Benefits of Using CLAMs

Scenario 1: Figure 9 shows the effective bandwidth improvement using CLAM-based and BDB based WAN-optimizers at different link speeds. Both WAN optimizers use Transcend-SSD. Figure 9(a) shows the results for a high (50%) redundancy trace (i.e., optimal improvement factor 2). The BDB-based WAN optimizer gives close-to-optimal improvement ($2\times$) at low link speeds of up to 10Mbps. However, at higher link speeds it becomes a bottleneck and drastically reduces the effective bandwidth instead of improving it. In comparison, CLAM-based WAN-optimizer gives close-to-ideal improvement at $10\times$ higher (100Mbps) link speeds and gives reasonable improvements even at 200 Mbps. It becomes a bottleneck at 400Mbps making its usage obsolete at such speeds. Using Intel-SSD, the CLAM-based WAN-optimizer can run up to 500 Mbps while offering close to ideal improvement, but using Intel-SSD with BDB does not improve the situation significantly. A similar trend was observed for the low (15%) redundancy trace (i.e., optimal improvement factor 1.18) whose results are shown in Figure 9(b). In this case, CLAM-based WAN-optimizer is able to operate at even higher link speeds while giving close to ideal improvement. This is because, when redundancy is low, lookups in the case of CLAMs seldom go to flash, which results in higher throughput.

Scenario 2: We fix the link speed to be 10Mbps for this analysis, because BDB is ineffective at higher speeds. We use a trace with 50% redundancy. We now take a closer look at the improvements by CLAMs and BDB. Figures 10 (a) and (b) show the relative throughput improvement on an object-by-object basis (Only improvements up to factor of 2 is shown). We see that

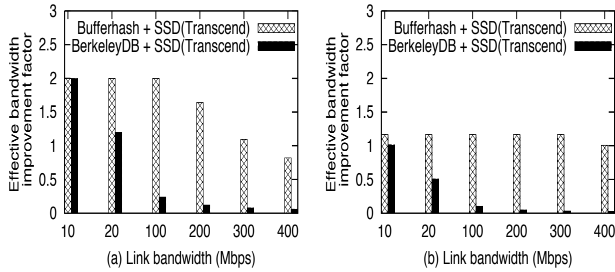


Figure 9: *Effective capacity improvement vs different link rates for (a) 50% and (b) 15% redundancy traces.*

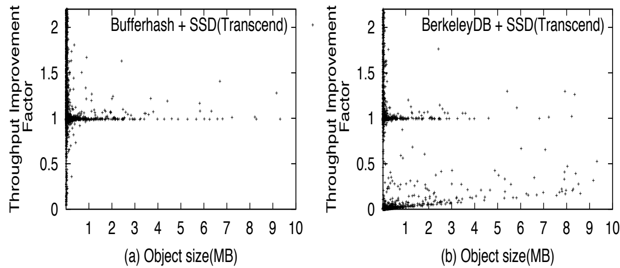


Figure 10: *Heavy load scenario: Throughput improvement per object for (a) BufferHash-based CLAM using Transcend SSD and (b) Berkeley-DB on Transcend SSD.*

Berkeley-DB has a negative effect on the throughputs of a large number of objects (compared to ideal), especially objects 500KB or smaller; their throughput is worsened by a factor of two or more due to the high costs of lookups and inserts (the latter for fingerprints of prior objects). Our CLAM also imposes overhead on some of these objects, but this happens on far fewer occasions and the overhead is significantly lower. Also, the average per-object improvement is 3.1 for our CLAM, which is 65% better than BDB (average improvement of 1.9).

9 Conclusions

We have designed and implemented CLAMs (Cheap and Large CAMs) for high-performance content-based networked systems that require large hash tables (up to 100GB or more) with support for fast insertion, lookups and updates. Our design uses a combination of DRAM and flash storage along with a novel data structure, called BufferHash, to facilitate fast hash table operations. Our CLAM supports a larger index than DRAM-only solutions, and faster hash operations than disk- or flash-only solutions. It can offer a few orders of magnitude more hash operations/s/$ than these alternatives. We have incorporated our CLAM prototype in a WAN optimizer and showed that it can enhance the benefits significantly.

Our design is not final, but it is a key step toward supporting high speed operation of modern data-intensive networked systems. It may be possible to design better CLAMs by leveraging space-saving ideas from recent systems such as FAWN [13] (to help control the amount of DRAM needed by BufferHash), using coding techniques such as floating codes (for better eviction support), or by using newer memory technologies such as Phase Change Memory (which can support much better read/write latencies than flash).

Acknowledgements. The authors would like the following people for their comments that helped improve our work directly or indirectly: Tom Anderson, Remzi Arpaci-Dusseau, Hari Balakrishnan, Flavio Bonomi, Patrick Crowley (our shepherd), Vivek Pai, Jennifer Rexford, Vyas Sekar, Srinivasan Seshan, Scott Shenker, Michael Swift and David Wetherall. This work is supported in part by an NSF FIND grant (CNS-0626889), an NSF CAREER Award (CNS-0746531), an NSF NetSE grant (CNS-0905134), and by grants from the UW-Madison Graduate School and Cisco.

References

[1] BlueCoat: WAN Optimization. `http://www.bluecoat.com/`.

[2] Cisco Wide Area Application Acceleration Services. `http://www.cisco.com/en/US/products/ps5680/Products_Sub_Category_Home.html`.

[3] Computerworld - WAN optimization continues growth. `www.computerworld.com.au/index.php/id;1174462047;fp;16;fpid;0/`.

[4] Disk Backup and deduplication with DataDomain. `http://www.datadomain.com`.

[5] Dropbox. `http://www.getdropbox.com`.

[6] Oracle Berkeley-DB. `http://www.oracle.com/technology/products/berkeley-db/index.html`.

[7] Peribit Networks (Acquired by Juniper in 2005): WAN Optimization Solution. `http://www.juniper.net/`.

[8] Riverbed Networks: WAN Optimization. `http://www.riverbed.com/solutions/optimize/`.

[9] Riverbed SteelHead Product Family. `http://www.riverbed.com/docs/SpecSheet-Riverbed-FamilyProduct_xx50_SMC-VE.pdf`.

[10] WAN Optimization Design. Private communication with a major vendor.

[11] D. Abadi et al. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.

[12] A. Anand, S. Kappes, A. Akella, and S. Nath. Building cheap and large cams using bufferhash. Technical Report 1651, University of Wisconsin, 2009.

[13] D. Andersen, J. Franklin, M. Kaminsky, A. Phan-

ishayee, L. Tan, and V. Vasudevan. Fawn: A fast array of wimpy nodes. 2009.

[14] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. Stream: The stanford stream data manager. *IEEE Data Engineering Bulletin*, 26, 2003.

[15] L. Arge. The buffer tree: A new technique for optimal i/o-algorithms. In *4th International Workshop on Algorithms and Data Structures (WADS)*, 1995.

[16] A. Badam, K. Park, V. S. Pai, and L. Peterson. Hashcache: Cache storage for the next billion. In *NSDI*, 2009.

[17] L. Bouganim, B. Jónsson, and P. Bonnet. uFLIP: Understanding flash IO patterns. In *CIDR*, 2009.

[18] A. Broder and M. Mitzenmacher. Using multiple hash functions to improve IP lookups. In *IEEE INFOCOM*, 2001.

[19] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2005.

[20] J. Bromley and W. Hutsell. Improve application performance and lower costs. Texas Instruments, http://www.texmemsys.com/files/f000240.pdf.

[21] A. Caulfield, L. Grupp, and S. Swanson. Gordon: Using flash memory to build fast, power-efficient clusters for data-intensive applications. In *ASPLOS*, 2009.

[22] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: a stream database for network applications. In *SIGMOD*, 2003.

[23] P. J. Desnoyers and P. Shenoy. Hyperion: high volume stream archival for retrospective querying. In *USENIX*, 2007.

[24] Endace Inc. Endace DAG3.4GE network monitoring card. `http://www.endace.com/`, 2009.

[25] U. Erlingsson, M. Manasse, and F. McSherry. A cool and practical alternative to traditional hash tables. In *Proceedings of the 7th Workshop on Distributed Data and Structures (WDAS'06)*, 2006.

[26] C. Faloutsos and S. Christodoulakis. Signature files: An access method for documents and its analytical performance evaluation. *ACM Transactions on Information Systems*, 2(4):267–288, 1984.

[27] H. Finucane, Z. Liu, and M. Mitzenmacher. Designing floating codes for expected performance. In *Allerton*, 2008.

[28] R. Hagmann. Reimplementing the cedar file system using logging and group commit. In *ACM SOSP*, 1987.

[29] G. Iannaccone, C. Diot, D. McAuley, A. Moore, I. Pratt, and L. Rizzo. The CoMo white paper. Technical Report IRC-TR-04-17, Intel Research, 2004.

[30] A. Jiang, V. Bohossian, and J. Bruck. Floating codes for joint information storage in write asymmetric memories. In *ISIT*, 2007.

[31] A. Kirsch and M. Mitzenmacher. The power of one move: Hashing schemes for hardware. In *IEEE INFOCOM*, 2008.

[32] T. Koponen, M. Chawla, B. Chun, A. Ermolinskiy, K. Kim, S. Shenker, and I. Stoica. A data-oriented (and beyond) network architecture. In *ACM SIGCOMM*, 2006.

[33] G. Mathur, P. Desnoyers, D. Ganesan, and P. Shenoy. Capsule: an energy-optimized object storage system for memory-constrained sensor devices. In *ACM SenSys*, 2006.

[34] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. *SIGOPS Oper. Syst. Rev.*, 35(5), 2001.

[35] S. Nath and P. B. Gibbons. Online Maintenance of Very Large Random Samples on Flash Storage . In *VLDB*, 2008.

[36] S. Nath and A. Kansal. FlashDB: dynamic self-tuning database for NAND flash. In *ACM/IEEE IPSN*, 2007.

[37] H. Pucha, D. G. Andersen, and M. Kaminsky. Exploiting similarity for multi-source downloads using file handprints. In *NSDI*, 2007.

[38] M. Rabin. Fingerprinting by random polynomials. Technical report, Harvard University, 1981. Technical Report, TR-15-81.

[39] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *ACM SOSP*, 1991.

[40] StorageSearch. RAM SSDs versus Flash SSDs—technology and price trends. http://www.storagesearch.com/ssd-ram-v-flash.html.

[41] StreamBase Inc. Streambase: Real-time low latency data processing with a stream processing engine. `http://www.streambase.com/`, 2009.

[42] N. Tolia, M. Kaminsky, D. G. Andersen, and S. Patil. An architecture for Internet data transfer. In *Proc. 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, May 2006.

[43] Z. Wei, K. Yi, and Q. Zhang. Dynamic external hashing: The limit of buffering. In *ACM SPAA*, 2009.

[44] D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W. A. Najjar. Microhash: an efficient index structure for fash-based sensor devices. In *USENIX FAST*, 2005.

[45] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *FAST*, 2008.