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 applications can use different strategies for state re-creation,

[†] adya@google.com. Work done while at Microsoft.
[*] {jdunagan, alecw}@microsoft.com.

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

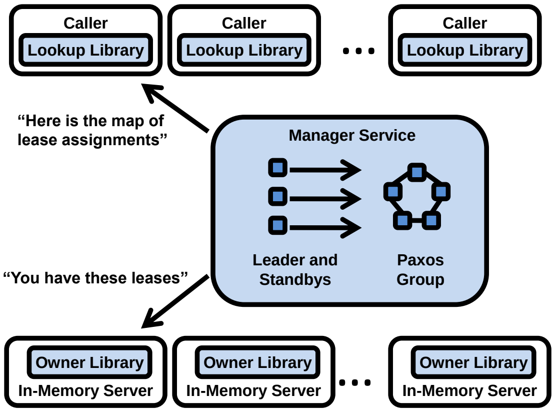

Figure 1: *Servers using Centrifuge link in libraries that talk to a Centrifuge Manager service.*

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, 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.

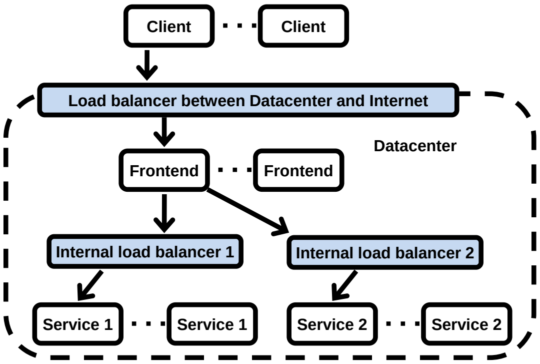

Figure 2: *Datacenter applications are often divided into multiple component services, and servicing clients requests frequently requires communicating with multiple such services. Centrifuge is designed to replace only the internal network load balancers used by the component services.*

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.

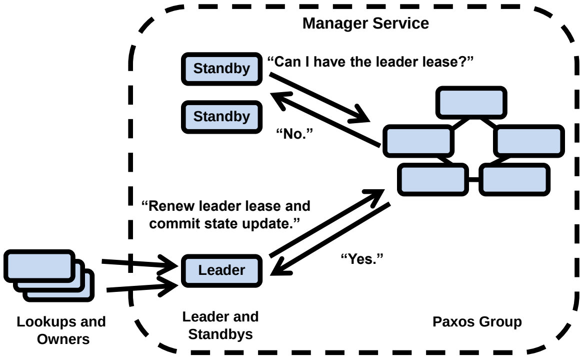

Figure 3: *In the Manager service, one set of servers run a Paxos group that provides a state store and a leader election protocol, and another set of servers act as either leader or standby. Only the current leader executes the logic for partitioning, lease management, and communication with Lookups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

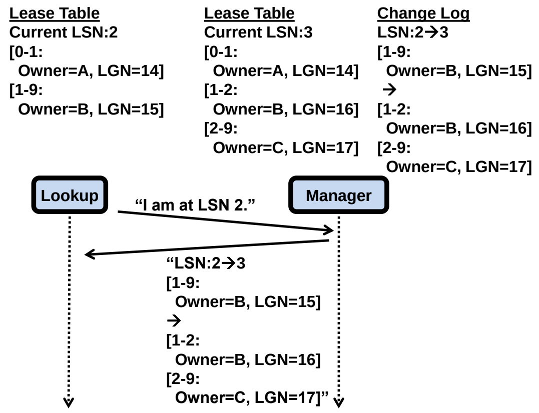

Figure 4: *Example of the protocol between the Lookup and the Manager.*

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

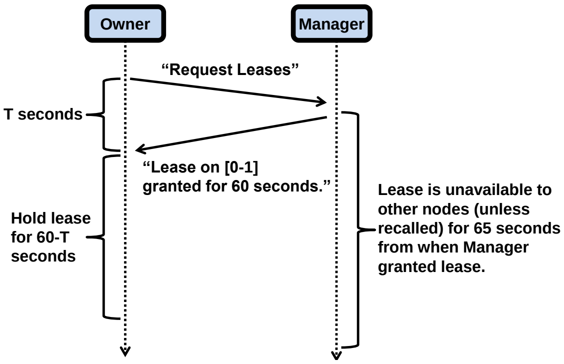
Owner
Manager
"Request Leases"
T seconds
"Lease on [0-1] granted for 60 seconds."
Hold lease for 60-T seconds
Lease is unavailable to other nodes (unless recalled) for 65 seconds from when Manager granted lease.

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 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.

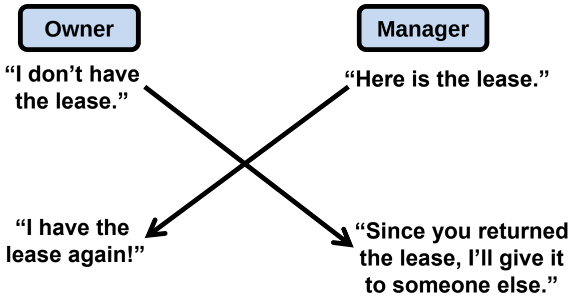
Owner
Manager
"I don't have the lease."
"Here is the lease."
"I have the lease again!"
"Since you returned the lease, I'll give it to someone else."

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.*

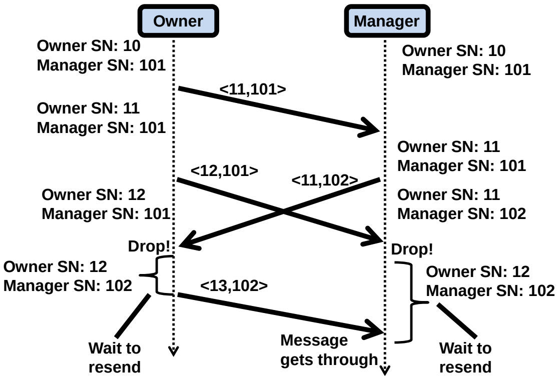

Figure 7: *The Manager and Owner use sequence numbers to filter out messages races. When a message race occurs, they wait for a random backoff, and then resend.*

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.

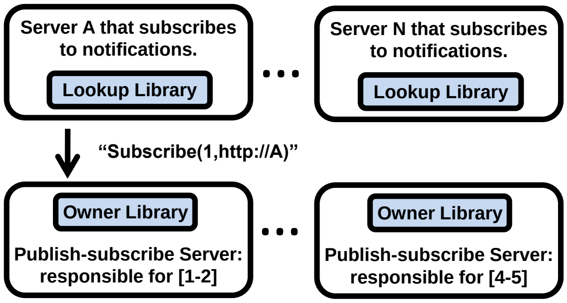
Server A that subscribes to notifications.
Lookup Library
Server N that subscribes to notifications.
Lookup Library
"Subscribe(1,http://A)"
Owner Library
Publish-subscribe Server: responsible for [1-2]
Owner Library
Publish-subscribe Server: responsible for [4-5]

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 continuously, 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.

```
// 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.*

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.

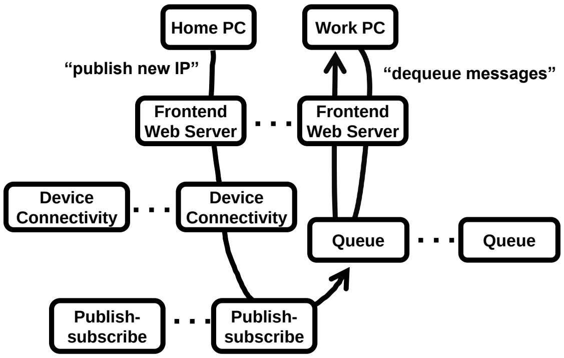

Figure 11: *How three Centrifuge-based services within the Live Mesh application cooperate to enable a work PC to connect to a home PC that has just acquired a new IP address.*

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 LossNotificationUpcall() 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.

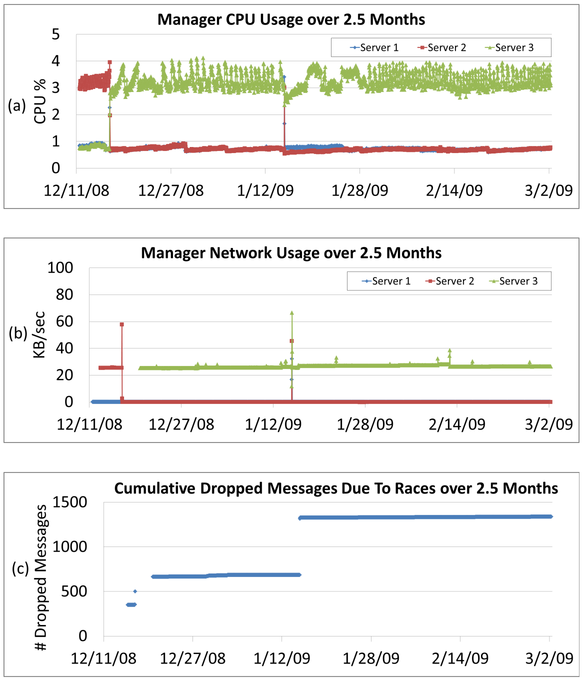

Figure 12: *CPU and network load measurements from the leader and standbys in the Centrifuge Manager service deployed in production over 2.5 months, as well as messages dropped due to races.*

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 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.

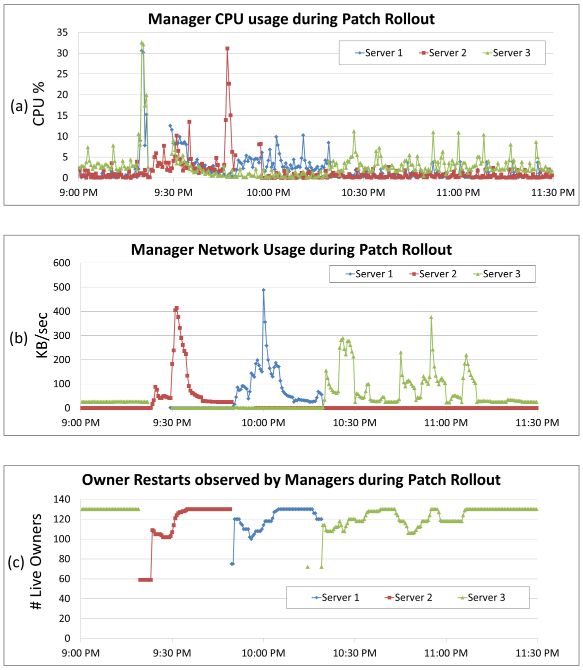

Figure 13: *Measurements from the evening of 1/15/2009 for the leader and standby servers in the Manager service deployed in production, capturing CPU, network load, and Owner restarts.*

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.

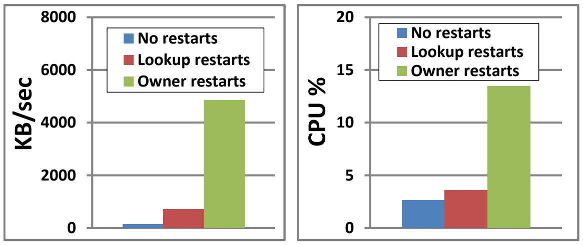

Figure 14: *The CPU and network load on the Manager leader under rapid restarts for a large number of Owners and Lookups.*

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.