PRS: A Reusable Abstraction for Scaling Out Middle Tiers in the Datacenter

Atul Adya* John Dunagan† Alec Wolman†

Abstract: *Scale-out datacenter applications are of enormous commercial interest, yet they are frustratingly hard to build. A common design pattern is to locate most of the application logic in a middle tier of soft-state servers and to provide storage using a separate backend system. The Partitioning and Recovery Service (PRS) is an infrastructure service that makes it easier to build these middle tier applications: it provides strong consistency on the soft state, supports arbitrary operations, and helps in recovery if state is ever lost. The PRS manages scaling out the middle tier across many machines, and hides the complexities of scale-out through a simple and powerful abstraction. We have built and evaluated the PRS, and other developers have built and deployed four scale-out soft-state datacenter applications using the PRS. Our experience with these applications confirms that the PRS significantly simplifies the construction of scale-out soft-state datacenter applications.*

1 Introduction

In recent years there has been a massive push in the computer industry to build enormous datacenters. These datacenters are used to deliver a class of compelling and commercially important applications, such as instant messaging, social networking and web search. Building these large distributed applications is an extremely challenging task. Fortunately, both the academic community and industry have proposed software infrastructure that encapsulates common functionality, making it easier to build and deploy new applications: Dynamo [18], Dryad [36], MapReduce [12], BigTable [15], DDS [48], and Chubby [10] are some notable examples.

A common pattern in building datacenter applications is to split functionality into stateless frontend servers, soft-state middle-tier servers containing complex application logic, and a backend storage system. Much prior work has focused on scalable backend storage systems. In contrast, the Partitioning and Recovery Service (PRS) makes it easier to build soft-state middle tiers by providing *strong consistency* on the soft state, *help recovering* if state is lost, and still allowing the application to perform *arbitrary operations*. Applications make use of the PRS by linking in libraries at the frontends and middle tier servers; the libraries talk to a replicated manager. To provide consistency, the manager runs a lease protocol; to allow scaling out the middle tier, the manager dynamically partitions a key space across the middle tier servers and communicates this key placement to the frontends; and to provide recovery, the manager must correlate lost leases with its key placement decisions. Finally, the manager and libraries support arbitrary operations by staying out of the path of all frontend and middle-tier communication. To the best of our knowledge, ours is the first attempt to abstract out strongly consistent soft-state partitioning into a reusable component.

Soft state has been previously defined in multiple ways in the literature. For this paper, we define soft state as state that can be *automatically* recreated, typically with some modest cost. Many datacenter applications use soft state, including web site middle tiers [43], various distributed caches [40, 37], Instant Messaging (IM) [31, 56, 20], VoIP chat [41], and video conferencing [29]. Middle tiers and distributed caches typically rely on a backend storage system to enable recreating lost state, whereas IM, VoIP chat and video conferencing systems rely on clients. In this latter class of applications, if state is lost, clients republish their presence data or reattempt rendezvous. Our definition of soft state does not include data that requires significant human effort to recreate, e.g., shopping carts [53, 8]. If an application wants to operate on such data in the middle tier, it should commit changes to a backend storage system. With this model, the shopping cart is hard state, but the copy of the shopping cart maintained by the middle tier is soft state.

We explain how developers use the PRS by walking through the example of an IM application shown in Figure 1. Requests enter the datacenter through a load balancer, and arrive at an arbitrary frontend. The frontend extracts the username from the request and asks the PRS which middle-tier node it should contact to find that user's IM presence data; the frontend then routes the request. When the request arrives at the middle-tier node, the middle-tier node checks with the PRS that this des-

{Microsoft Corporation*, Microsoft Research†}, Redmond, WA.
{adya, jdunagan, alecw}@microsoft.com

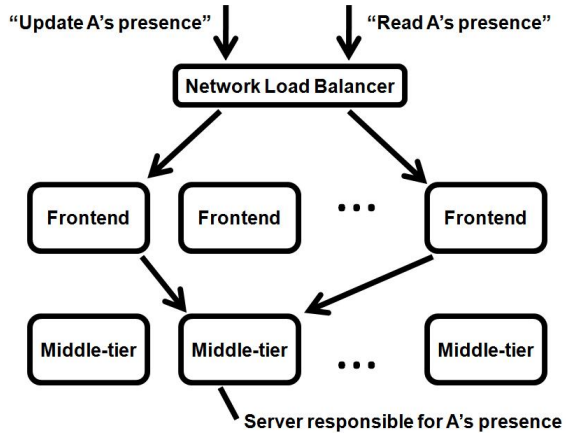

Figure 1. *A datacenter IM application. Requests enter through a network load balancer, arrive at some arbitrary frontend, and are forwarded to the appropriate middle tier node.*

tination is still correct, and then it processes the request. Independently, the PRS notifies the appropriate frontend if any middle-tier state is lost so that the frontend can take appropriate recovery actions, e.g., asking an IM client to republish its presence.

The PRS has been used to build four applications that are currently deployed and serving live customer requests as part of the Live Mesh product [30]. Live Mesh is a new developer platform and an end-user application for sharing and synchronization; it relies on the PRS for all its soft-state scale-out needs. Other developers wrote all four applications, and in Section 3 we report on how the PRS simplified the development of these applications.

We now describe each of the three key properties the PRS provides to middle-tier developers: strong consistency on soft state, help recovering and support for arbitrary operations. *Strong consistency* in the PRS context means that applications can treat their data as having at most one copy, and thus all operations on a key can be serialized. Strong consistency is sometimes criticized for hurting availability, but recent industry experience in datacenters has shown that the availability penalty can be made tolerably small (e.g., BigTable [15] describes the perceived downtime of Chubby [10] as 0.0047%). Furthermore, weak consistency has two significant downsides: it is more difficult to program against [1], and it can lead to a poor user experience. For example, in an IM system, if user A updates her presence status at one server, and then user B attempts to read A's presence from a different server, B will receive stale data, and be unable to communicate with A. In contrast, with the PRS there may be a short period of unavailability if user A's presence is lost, but this is quickly fixed by republishing, and inconsistent state is never exposed to end users. Though the PRS enables strong consistency, the PRS does not mandate it – applications are free to deviate from single-copy semantics. Indeed, of the four applications using the PRS, three need strong consistency, while the fourth needs only weak consistency, allowing for improved availability while still leveraging the PRS's other benefits. Finally, like many other datacenter abstractions [48, 15], the consistency guarantee only applies to operations on the data associated with a given key, not operations that span multiple keys.

The PRS provides *help recovering* soft state by delivering notifications at frontends when recovery is needed; applications can then perform this recovery in an application-specific manner. Recovery notifications are a natural responsibility for the PRS because it directs the assignment of keys to servers: when a server crashes, the PRS can signal the ranges of keys it is assigning to new nodes, and hence what keys need to be republished. Integrating recovery notifications with partitioning leads to a simpler overall system than providing recovery notifications through a separate service or requiring each application to detect the need for recovery on its own.

Supporting *arbitrary operations* means that when a request arrives at the node responsible for a given key, the node can execute arbitrary application code to process the request, and it can leverage the PRS to get strong consistency on its soft state. This is a departure from prior datacenter infrastructure services that support arbitrary operations but only on weakly-consistent soft state (e.g., BASE semantics [4]), or that support strong consistency but only within a storage service (e.g., BigTable [15]).

1.1 Our Contributions

The primary contribution of this paper is a novel abstraction that is powerful and flexible, yet simple to use. The PRS abstraction is *powerful* because it encapsulates significant pieces of complex functionality: scaling out to a varying number of machines, providing strong consistency, and signaling when state may need to be republished. This allows developers to build sophisticated applications without having to implement such functionality themselves. The PRS abstraction is *flexible* because it supports arbitrary operations, it allows these operations to tradeoff availability and consistency at their discretion, and it does not constrain the protocols between the frontends and middle-tier. The PRS *simplifies* the frontend by telling the application where to route requests to in the middle tier. The PRS simplifies recovery by telling the application when to recover and which items need recovering. The PRS simplifies the middle tier by telling the application whether or not to process a request. Further evidence for the PRS's simplicity comes from the fact that other developers used the PRS to successfully build and deploy four applications. They did not have to worry about many of the complexities of scaling out and operating in a distributed environment, allowing them to focus

primarily on the semantics of their applications.

Other important contributions of this paper are:

- a detailed description of the PRS API and guarantees;
- how other developers used the PRS to build four different soft-state scale-out datacenter applications; and
- the design, implementation and experimental evaluation of the PRS.

2 Overview and API

To provide more context for the discussion of the PRS API and semantic guarantees, we begin this section by giving a brief overview of the PRS architecture. We will give a more detailed description of the PRS design and implementation in Section 4.

Frontends that want to route requests link in the Lookup library. Middle-tier servers that want to serve requests link in the Owner library. We describe these servers as being located in a frontend or middle-tier to best associate them with current practices. However, nothing in the PRS design prevents other usage models (one example is described in Section 3), and there is no requirement that a backend exist behind the middle-tier. The Lookup and Owner libraries provide their guarantees to applications by coordinating through a logically centralized Manager. The Manager is replicated across multiple machines for fast failover. Figure 2 shows this architecture.

The Manager partitions the entirety of the key space to the various Owners through a lease protocol. In contrast to many previous lease managers (e.g., Chubby [10]), Owner libraries do not ask for a lease on a particular key, or even a particular range of keys. Instead, Owners simply notify the Manager of their liveness, and the Manager *assigns* ranges in the key space to Owners. To keep the load distribution even, when a new Owner is added, the Manager recalls leases from other Owners as necessary and grants them to the new Owner.

The Manager uses consistent hashing to determine the ranges to assign to each server. Each Owner is given a fixed number of virtual nodes (64 in the current implementation), with more virtual nodes leading to a more even distribution of load across the servers. Lookup libraries periodically synchronize to the Manager's map of lease assignments.

The PRS provides two safety properties:

- **Recovery Notification Guarantee:** Any soft-state loss at a middle-tier server is signaled to all frontends within a bounded period of time.
- **Single-copy Guarantee:** Because there is at most one valid copy of the state, operations on the state can be serialized at the node holding this copy.

We explain how applications make use of the recovery notification guarantee in Section 2.1, and how they make use of the single-copy guarantee in Section 2.2.

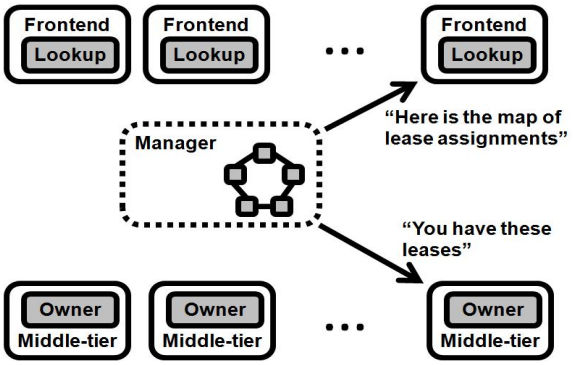

Figure 2. *Overview of the PRS Architecture: Frontends link in the Lookup library, Middle-tier servers link in the Owner library, and both libraries talk to a replicated logically centralized Manager.*

2.1 Lookup API

The PRS API is shown in Figure 3. A frontend that wishes to route a request for some key calls Lookup(). The key supplied to Lookup() is typically derived by hashing some input string, such as the user's instant messaging address. In this setting, hash collisions are only a minor inconvenience – two items that hash to the same key cannot be split apart for the purpose of load-balancing.

The PRS design does not force the use of hashing. By simply supplying the input string as the key, applications can also partition on ranges of the input space, e.g., range [a-j] to node1, range [k-z] to node2 (as in BigTable [15]). However, all applications currently using the PRS prefer hashing, and so the current implementation only supports 64-bit keys, not arbitrary length keys.

The URL returned by Lookup() is whatever the middle-tier server specifies when it instantiates its Owner library. This allows the application to use any protocol it desires between the frontends and middle-tier. For some callers, this URL is simply an IP address and a TCP port number. The semantics of Lookup() are that it is allowed to return a stale address. The staleness can be checked (and the request rejected) at the middle-tier using the Owner API. Because the Manager rapidly propagates updated versions of the lease map to frontends, frontends should retry after a short backoff on such rejected requests.

The semantics of RecoveryNotificationUpcall() are that the frontend is guaranteed to receive an upcall within a bounded period of time whenever any Owner loses its lease. The upcall indicates the ranges within the key space that need to be recovered. In the IM example, the frontend should pass this notification along to any connected clients that had stored state within the affected key range, so that they can republish.

The recovery notification guarantee is conservative; recovery notifications may also be delivered even though an

```
// Lookup API
URL Lookup(Key key)
void RecoveryNotificationUpcall(KeyRange[] toRecover)

// Owner API
bool GetOwnershipHandle(Key key,
  out OwnershipHandle handle)
bool CheckContinuousOwnership(OwnershipHandle handle)
void OwnershipUpcall(KeyRange[] grant,
  KeyRange[] revoke)
```

Figure 3. *The PRS API. Asynchronous versions of the calls are omitted. Upcalls are given as arguments to the relevant constructors.*

Owner still holds its leases. For example, if the frontend's Lookup library loses contact with the Manager, recovery notifications will be delivered on all ranges. In this case, the frontend is no longer receiving updates to its map of lease assignments, and hence may no longer be able to route requests. Clients should be robust to this frontend failure mode, and they should reconnect to other frontends. Upon reconnection, they must also republish their state.

2.2 Owner API

The Owner API provides two calls to ensure consistency: GetOwnershipHandle() checks that a lease is currently held and creates a handle, while CheckContinuousOwnership() checks that the lease has been continuously held since the handle was created. A middle-tier server can use these calls to guarantee strong consistency: operations on the state associated a key can be serialized. We explain how to do this in general by walking through the example shown in Figure 4: a distributed soft-state in-memory hashtable with strong consistency and recovery notifications (though no code is needed in Figure 4 to enable recovery notifications).

The general pattern to using the Owner API is shown at the top of Figure 4. When a request arrives at a middle-tier server, step (1) is to call GetOwnershipHandle() to check whether it should serve the request. If this call succeeds, step (2) is to validate any previously stored state. In the example, Put() can skip this because any previously stored state will be replaced regardless.

Step (3) is to perform an arbitrary local operation, and then to store any created state along with the owner handle. Developers can treat Figure 4 as defining a template: an application's arbitrary local operation should be cut-and-paste into step (3), and then the result stored along with the handle, as in Put(). Get() skips the step of storing state because it makes no modifications.

Internally, an ownership handle consists of a key, a lease generation number, and the Manager's nonce. If a later operation arrives for this key, step (2) of the later operation will use the stored handle's lease generation number to check that the lease has been continuously held, and thus that the stored state is still valid.

```
// General pattern to guarantee serializability:
// (1) Check that request has arrived at correct node
// (2) Validate that existing state is not stale
// (3) Perform arbitrary local operation;
//     store handle with any created local state
// (4) Check that lease has covered entire operation
// (5) Expose state externally (e.g., return)

bool Put(key, value) {
  // (1) check that this is the correct node
  ok = GetOwnershipHandle(key, out handle);
  if (!ok) return false; // avoid wasted work
  // (2) validate previously stored state
  // in this case, skip because about to overwrite it
  // (3) perform arbitrary local operation;
  //     store handle with any created local state
  // in this case, simply store value and handle
  this.state[key] = value;
  this.handles[key] = handle;
  // (4) check that lease covered operation
  ok = CheckContinuousOwnership(handle);
  if (!ok) return false; // lease was lost
  return true; // (5)
}

// If this node is the owner, yet the item does not
// exist, return true with value=null
bool Get(key, out result) {
  // (1) check that this is the correct node
  ok = GetOwnershipHandle(key, out handle);
  if (!ok) return false; // avoid wasted work
  // (2) validate previously stored state
  storedHandle = this.handles[key];
  if (!CheckContinuousOwnership(storedHandle))
     this.state[key] = null;
  // (3) perform arbitrary local operation;
  //     store handle with any created local state
  // in this case, simply set ''out value''
  value = this.state[key];
  // (4) check that lease covered operation
  ok = CheckContinuousOwnership(handle);
  if (!ok) return false; // lease was lost
  return true; // (5)
}
```

Figure 4. *Using the Owner API to build a distributed soft-state in-memory hashtable with strong consistency and recovery notifications.*

In step (4), the node checks that this entire operation has been covered by the lease without interruption. If the check succeeds, the lease guarantees that no other operation on this state occurred in parallel on another node. If the check fails, the operation also fails and the node returns failure to the frontend, who can then retry. In the hashtable example, a lost lease does not require rolling back the operation – when a future call attempts to access this state, it will check the corresponding stored lease handle (step (2)) and know that the stored state is invalid.

Step (5) is to return to the caller, or more generally, to expose state from this operation to another node. Put() does not expose state; Get() exposes state back to the caller. In general, the middle-tier server might want to expose state to other servers before returning, e.g., communicating with a backend storage system. The simplest way to handle multiple such external state exposures is to treat each exposure as marking the end of one operation and the beginning of the next; each individual operation should simply perform steps (1)-(5). Though it may

sometimes be possible to elide some of the Owner API calls, there is little benefit to reasoning about such subtleties because all Owner API calls are purely local.

Applications are free to skip using the Owner API to guarantee serializability; they can answer requests using arbitrary local (and potentially stale) data. However, we believe many application developers prefer strong consistency because of the simpler semantics: updates and reads never go to separate copies of the data, and there is no need to reconcile data with distinct histories, either defining a merge operation or having to think about lost updates in the case of last-writer-wins reconciliation. These are the standard reasons for valuing strong consistency, and they are valuable even without data durability.

The final part of the Owner API is the OwnershipUpcall(). The 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. However, it is worth noting that the upcall by itself does not guarantee strong consistency. There is no local scheduling guarantee ensuring an upcall conveying lost ownership will occur before a new operation on the lost key is received. Consistency requires using GetOwnershipHandle() and CheckContinuousOwnership() as described previously.

2.3 Beyond Serializability

Although operations on a key are serialized at the middle-tier, a client receiving a response cannot know that the data it contains is still current; the data may have changed after the response was sent. This is identical to the model clients have even with a traditional database: strong semantics while the request is within the database, but no guarantee that the state is still valid by the time the response is received. Happily, there are standard techniques to address this problem. For example, one can associate a version number with the state and do optimistic concurrency control [9, 55]. This is often encapsulated in a library where operations abort if the version read by the client is no longer the version at the server.

Interestingly, most of the operations exposed by the four PRS applications discussed in Section 3 did not need version numbers. To understand why, we consider the Queue application, which stores a message queue under each key. When an enqueue operation arrives, the application applies it (enqueueing the message) regardless of whether the queue state has changed since the caller last read it. Thus, there is no need for version numbers. Many of the operations in other applications had similar semantics. Prior work on concurrency control suggests this is a common pattern [54].

It is also possible to use the lease generation numbers for optimistic concurrency control across servers. This model was also mentioned in Chubby [10]: like PRS ownership handles, Chubby sequencers encapsulate a lease generation number. By passing lease generation numbers along with messages, remote nodes can discard a message if it does not contain the most recent lease generation number observed by that node.

2.4 Discussion

Ownership change upcalls and recovery notifications cover two distinct state recovery scenarios. Ownership change upcalls occur at the middle-tier; they are needed by applications like a distributed cache where newly granted leases can trigger pre-fetching of state from a backend storage system and lease invalidations can trigger garbage collection. In contrast, recovery notifications occur at frontends; they are needed for applications like IM where clients are connected to the frontends, and the frontends need to ask the clients to republish their state.

To demonstrate the flexibility of the PRS, we describe how it could be used to build a replicated in-memory store. To replicate data k times in the distributed hashtable of Figure 4, the frontend can simply call Lookup() with k different keys derived from the original key in a consistent way (e.g., $key_i = hash(name, i)$), do a Put() or Get() on each one, and return success only if a quorum respond. However, this does not change the soft-state guarantees of the system, it just reduces the failure probability. Because applications using the PRS already have to deal with state recreation, we believe few will go to the trouble of replicating within the middle tier.

To the best of our knowledge, providing strong consistency on soft state has not previously been defined in the literature. One definition is simply "ACID without the Durability." Another way of thinking about it is to imagine a set of clients operating on traditional hard state, and to throw in an additional "demonic" client that occasionally issues delete operations.

3 Experience, Lessons and Surprises

Other developers on the Live Mesh [30] product team have used the PRS to build four soft-state scale-out datacenter applications. These applications are deployed and serving live customer requests. The applications provide significant functionality yet they are much simpler than if they had to implement partitioning, recovery and consistency on their own. These four "applications" are services rather than end-user applications. They are designed to be used both by the Live Mesh product and by other datacenter products.

These four applications demonstrate the ability of the PRS to support a variety of uses. Some of the ways that the developers used the PRS surprised us; we describe these surprises along with the lessons we learned.

- **Sectioned-Document:** The Sectioned-Document application is a kind of distributed hashtable. A key is mapped to a document. The document contains a version number, and updates can be made conditional on the document's current version for the reasons described in Section 2.3. Each document is sub-divided into a number of named sections, so that reads and updates can be done at a finer granularity than the whole document. Different instances of this application play different roles: for example, the activity service (AS) is used to store recent activity for each user, while the device connectivity service (DCS) is used to store information about how to contact devices. Documents are expired using a TTL value unless client requests explicitly refresh them; this provides robust resource reclamation if clients silently fail.
- **Publish-Subscribe:** The Publish-Subscribe application exports Subscribe() and PublishEvent() operations. This supports rendezvous between datacenter applications, and it can be used as a building block to enable client rendezvous. Subscriptions are similarly maintained using TTLs and explicit client refreshes.
- **Queue:** The Queue application is used to deliver messages to clients that may be offline. A client connects to its queue and drains any buffered messages. Queues are also maintained using TTLs and explicit client refreshes.
- **RSS-Processor:** The RSS-Processor application performs coalescing and transformation of incoming RSS [46] feeds using semantics specific to the types of RSS feeds it is processing. For example, it may take two RSS items that describe different activities by a single user (e.g., "Alice updated her profile" and "Alice added Bob as a friend") and produce a single item combining the descriptions ("Alice updated her profile and added Bob as a friend."). The processor also generates completely new RSS entries in response to other external events (e.g., Alice's IM presence being updated in the IM application may trigger producing "Alice is now online" in the RSS feed). Unlike the other three applications, the RSS-Processor is designed assuming its output will be directed to users rather than other applications, enabling it to select weak consistency. Cached RSS feed state is expired using TTLs.

To see how these applications work together, we consider what happens when a device comes online. First, it connects to the datacenter and updates the document that holds its connectivity information (using the Sectioned-Document store). The user's other devices have set up subscriptions using the Publish-Subscribe application so that changes in device connectivity are sent to their message queues. These other devices retrieve this message from the Queue application, and thus learn how to make a direct connection to the device that just came online.

The other developers did not always use the PRS in the ways we anticipated. We now describe in detail some of the PRS usage that was most surprising to us.

3.1 Varying the Consistency Model

Two applications chose to deviate from the standard usage of the Owner API. We describe their behavior here to better understand how applications can exploit the flexibility of the PRS API with regards to checking for consistency.

The Publish-Subscribe application achieved better availability while still maintaining strong consistency by exploiting its particular semantics: although calls to Subscribe must use the standard consistency check, the subscriptions they create are effectively immutable, and so calls to Publish do not need to do standard consistency checks about the validity of stored subscriptions. This is analogous to a cache of immutable data that can avoid many of the consistency checks associated with read/write data. We were surprised by the decision to work through this more subtle consistency guarantee, but it indicates the importance developers place on availability.

The RSS-Processor was the only application that chose weak consistency – if an operation arrives at any middle-tier node, that node processes it without checking whether it is the owner. Because of this, RSS operations may be omitted or performed multiple times, and the human reading the RSS feed has to cope. For the RSS-Processor application, stronger semantics (such as exactly-once processing) would have had two costs. First, stronger semantics would have decreased availability, requiring calls to be failed if they arrived at the wrong node. Second, stronger semantics would have required frequent calls to a storage system to persist processing results, a significant multiplier on the application's overhead. To avoid these costs, the RSS-Processor application chose weak consistency. Because the PRS separates liveness and item placement from consistency, the RSS-Processor application could use the PRS to handle its scale-out requirements, and it was still allowed to trade off consistency in favor of availability.

3.2 Centralized Recovery Notification Delivery

The Queue application allows clients to avoid maintaining persistent connections to a variety of application frontends. Instead, all messages from the various applications are funneled to a client's queue, where the client can later retrieve them. This set of messages must include PRS recovery notifications, as the client uses these to determine that it needs to republish some state. To ensure the Queue application can deliver recovery notifications for each application, it simply links in the Lookup library for each application, e.g., to deliver recovery notifications for Sectioned-Document applications, it links in

the Sectioned-Document Lookup library. This will notify the Queue middle tier whenever a Sectioned-Document middle-tier crashes. Though this design is obviously sensible, we had not anticipated that any middle-tier servers would themselves link in the Lookup library and never call Lookup().

3.3 Admission Control

In contrast to the other applications using the PRS, the RSS-processor application also implements application-level backoff: the number of outstanding requests at a frontend destined for any particular middle tier server is not allowed to exceed a fixed cap. Because the PRS is not in the communication path between the frontends and the middle tier nodes, it is easy to implement application-specific request throttling – the PRS does not force all applications to agree on a single throttling policy.

3.4 TTLs

All of the applications built using the PRS use TTLs. TTLs are used to garbage collect state when clients leave the system. In the absence of recovery notifications, TTLs could also be used to restore availability after a crash: the client could periodically republish state independent of whether there has been a failure. However, this would require very short TTL values (i.e., frequent polling). The need for rapid republishing is most acute when the state is shared, e.g., refreshed by one client, but operated on by many other clients. Because the PRS provides recovery notifications, all four applications can choose TTLs based solely on the need to garbage collect state for clients that have left the system. Because garbage collection can be done infrequently, the TTLs can be made quite long, leading to significant performance savings.

Surprisingly to us, none of the applications make use of the PRS's upcalls from the Owner library when ranges are revoked. The upcalls allow for rapid reclamation of state when responsibility for part of the key space is transferred to another Owner, but the applications chose to instead lazily rely on TTLs expiring. Given their existing need to use TTLs, this simple strategy reduced the number of moving parts at some resource cost.

We continue to believe that owner node upcalls will be useful to future applications using the PRS. To give one example, a colleague suggested building an application to perform some periodic task, and partitioning responsibility for that periodic task among a pool of servers using the PRS. In this hypothetical usage, the servers doing the periodic task link in the Owner library, and the Lookup library is not needed. The owner node upcalls provide a simple way for each server to learn the subset of the key space it should work on.

Application	Lines of Code	Consistency
Sectioned-Document (used by AS and DCS)	5,788	Strong
Publish-Subscribe	4,164	Strong
Queue	5,695	Strong
RSS-Processor	5,019	Weak

Table 1. *Lines of code and consistency model for each of the four applications built on the PRS.*

3.5 Summary

Overall, we feel these four soft-state applications substantially support the value of the PRS abstraction. These applications are non-trivial, collectively consisting of over 20,000 lines of C# code; the exact numbers, excluding test code and common utility libraries, are given in Table 1 along with the application's consistency model. By way of comparison, the PRS is about 20,000 lines of code itself; providing the PRS as a reusable component was an enormous savings compared to having each application implement their own distributed protocols providing PRS-like functionality. For all four applications, the PRS solved the scale-out problems for the developers of the application: monitoring middle-tier liveness, determining item placement on the middle-tier servers and conveying the item placement map to the frontends. The PRS enabled different applications to make different consistency choices simply by varying their usage of the PRS API. However, all three applications that have other applications as their clients chose strong consistency: these applications need to expose simple semantics to ease the task of writing client applications. Finally, the applications did find it beneficial to use the PRS abstraction of rapid recovery notifications to trigger recreation of soft state when needed, avoiding the need to build an additional membership and item tracking infrastructure service.

4 Design and Implementation

In this section we provide a detailed description of the PRS design, and how it provides both the single-copy guarantee and the recovery notification guarantee. We start by examining the operations and state at the Manager, and then move to the interactions of the Lookup and Owner libraries with the Manager. We next describe the details of the Manager's internal architecture and the Manager's scalability. We conclude by describing the overall complexity of the PRS.

4.1 Manager

The Manager maintains a table of all the ranges currently leased to Owner libraries, 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. The removal of an Owner is similar. The use of leasing ensures the single-copy guarantee. Changes to the lease table are kept in a change log used to efficiently synchronize Lookup libraries (see Section 4.2), and the change log is periodically truncated.

Every new lease assignment is done with a new lease generation number. If an Owner restarts, it is granted new leases, not simply extended the leases that it previously owned. If an Owner crashes and does not come back, new leases on its ranges are issued to other nodes. This guarantees that the Manager knows about every loss of soft state at an Owner, and this knowledge is always reflected in a change to the lease generation numbers for the lost ranges. In Section 4.2, we describe how every Lookup library is guaranteed to learn of every changed lease generation number, and how they use this to meet the recovery notification guarantee.

The particular algorithm that the Manager runs to assign leases is consistent hashing. Each Owner is assigned 64 virtual nodes. The choice of algorithm used to assign ranges to nodes is encapsulated in a single pluggable module within the Manager code. We found that this encapsulation was helpful for software engineering, and we used it to such ends as implementing tests that use fixed hashing over a set of Owners.

4.2 Lookup

Each Lookup library maintains a complete (though potentially stale) copy of the lease table: for every range, it knows both its lease generation number and the node holding the lease. This allows implementing Lookup() as a purely local operation. Due to the use of ranges, this only requires about 200KB of state per 100 Owners, a small amount for a frontend server.

The Lookup-Manager protocol synchronizes the Lookup library to the Manager's lease table every 30 seconds. Whenever the Manager receives a LookupRequest from a Lookup library, it either sends a snapshot of the lease table, or it sends the changes since the last time it communicated with the Lookup library. The Manager chooses to send a complete snapshot if either the change log has been truncated, or if the snapshot is more compact than the change log. The Manager also creates a nonce when it is initialized and includes this nonce in all its messages. The nonce allows the Lookup library to recognize Manager crashes and discard stale messages from previous Manager incarnations.

On receiving a LookupResponse from the Manager, the Lookup library must update its state. If the LookupResponse contains a series of changes, the Lookup library applies them in order to its copy of the lease table. If the LookupResponse contains a snapshot of the lease table, the Lookup library switches to using the new snapshot.

To meet the guarantee that every soft state loss at an Owner library is signaled as a recovery notification, the Lookup library must trigger a recovery notification for every range where the lease generation number has changed. If the LookupResponse contains a series of changes, the Lookup library knows that every change has a new lease generation number, and it can trigger recovery notifications appropriately. If the LookupResponse contains a new snapshot of the lease table, the Lookup library has to diff the snapshot in the message with the existing copy in the Lookup library, and it must then trigger a recovery notification on every range where the lease generation number has changed.

4.3 Owner

Each Owner library maintains all the ranges that are currently leased to it, including the lease generation number. This allows GetOwnershipHandle() and CheckContinuousOwnership() to be implemented as purely local calls (testing whether a range is currently held and whether the lease generation number in an ownership handle matches the current generation number, and hence has been held continuously). Owners send a LeaseRequest message every 15 seconds to the Manager. This LeaseRequest signals their liveness and specifies the leases where they want renewals. The Manager sends back a LeaseResponse 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. This refusal forces the Manager to issue a new grant on the range, thus triggering a change in the lease generation number. As mentioned in Section 4.1, this change in lease generation number is used to ensure the recovery notification guarantee.

We found that sending the complete lease table for an Owner in the LeaseRequest and LeaseResponse messages significantly simplified development and debugging of the lease protocol. We rarely had 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, we could look only at the previous message and the current message to see how they were incompatible. Also, sending the complete lease table has reasonable overhead – each Owner holds 64 ranges, each range is 32B, and this adds up to only 2KB per Owner.

Lease recall in this design has one subtle issue, depicted in Figure 5. When an Owner wishes to drop a lease (acknowledging a lease recall request), it sends a LeaseRequest that contains only the ranges it wants to renew. This improves liveness, as the Manager will quickly

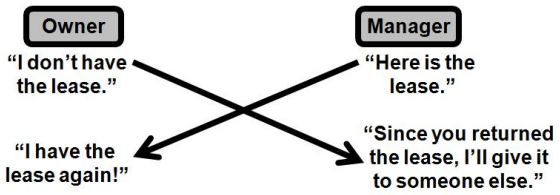

Figure 5. *Without care, message crisscross can lead to violating the single-copy guarantee.*

find out that the range is safe to lease to someone else. However, the Manager needs to be sure that there is no earlier message en route to the Owner that the Owner will interpret as regranting this range.

We solved this problem by adding two <*sequence number, nonce*> pairs in all lease messages. The sender of a message includes both its own <*sequence number, nonce*> pair and the most recent <*sequence number, nonce*> pair it heard from the remote node. The sequence numbers allowed us to drop one of the messages whenever a crisscross occurred. The nonces allow recognizing crashes and discarding stale messages, as in the Section 4.2.

4.4 Manager Failover

Figure 6 shows the internal architecture of the PRS Manager. The PRS Manager is composed of several Manager servers and a Paxos group. The bulk of the code runs in the Manager server, while the Paxos group provides a single limited but important function: allowing the Manager servers to compete to acquire a leader lease, and hence quickly failover to a new server if one crashes. This leader lease conveys responsibility for interacting with Lookup and Owner libraries.

If the current leader fails, one of the other Manager servers is able to acquire the lease after a short period of time. On failover, all previously granted leases are allowed to expire before any new leases are granted. This is a semantically correct yet expensive operation – all soft state being managed by this particular PRS Manager is discarded and must be recreated. Although Paxos is not being used here to provide state reliability, it is still providing a highly available mechanism for coordinating the choice of a new Manager server. This allows the PRS to quickly recover without operator intervention. Although we have designed the protocol for the Manager to store its state in Paxos and thereby eliminate the performance cost of individual Manager failures, this is not something we have finished implementing.

The Lookup and Owner libraries maintain a list of all Manager servers. In the common case, the Lookup and Owner libraries only send messages to the Manager server holding the leader lease. If they ever find the leader to be unresponsive, they start sending their messages to all the Manager servers until they find a responsive leader, at

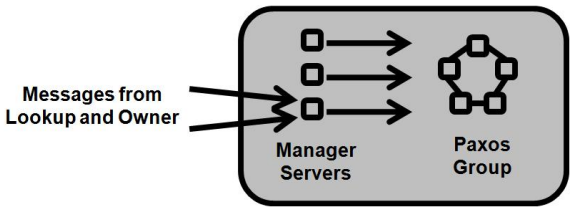

Figure 6. *Internal Architecture of the PRS Manager: Messages from Lookup and Owner libraries arrive at a Manager server holding a leader lease; a Paxos group runs the lease protocol and elects a leader from among the Manager servers.*

which point they switch back.

4.5 Manager Scalability

The PRS is designed to work within a datacenter management paradigm where the incremental unit of capacity is a cluster of approximately 400 machines. Products like Live Mesh are 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 the PRS – it must be able to support all the machines within a single cluster.

Partitioning across clusters is also needed, but inter-cluster partitioning requires tackling a different set of hard problems related to geographic distribution: greater scale and a less reliable network. When there are multiple clusters, the current PRS design can be used within each cluster, but a separate mechanism is needed to partition resources across clusters. We are currently in the process of designing and implementing such a mechanism.

Some approaches to geographic distribution may be incompatible with hashing; Chord [21] represents an approach that fundamentally relies on hashing, while DNS uses an administratively defined hierarchy. Because the PRS allows hashing but does not require it (Section 2.1), we are optimistic that it is compatible with a variety of mechanisms for partitioning across clusters.

To further improve the scalability of the PRS, we have implemented adaptive load management and state migration (i.e., when key ranges are transferred between owners, the data is also transferred, instead of relying on it being republished). The PRS running in production does not include these features; they are part of the augmented version of the PRS that we evaluate in Section 7. Owners report their incoming request rate as their load. The adaptive load management algorithm uses these load measurements to subtract a virtual node from any Owner that is 10% above the mean load, and to add a virtual node to any Owner that is 10% below the mean load (currently it is integrated with consistent hashing). The basic design for state migration is for the PRS to transfer the lease, for

Component	Lines of Code
Manager Only	7,432
Owner library Only	1,629
Lookup library Only	1,242
Shared	9,888

Table 2. *Lines of code in each component of the PRS.*

the application to transfer the data, and for them to coordinate via upcalls.

4.6 Code Complexity

To give an idea of the overall complexity of each of these components, we report in Table 2 the amount of code unique to each one, as well as the additional PRS code they all share. This count excludes both test code and common utility libraries (such as logging support). All code is written in C#. The "Shared" category includes a simple messaging layer over TCP. The Paxos group is implemented by adding a leader election operation to a pre-existing generic Paxos layer; this code is not included in the totals in the above Table.

5 Scale-Out Design Alternatives

In this section, we compare the PRS approach to several alternative approaches for building scale-out soft-state datacenter applications.

5.1 Weak Consistency

Many datacenter infrastructure services provide only weak consistency: they do not guarantee that requests on the same item will be routed to the same node. For example, Dynamo [18] provides weakly consistent scale-out storage, Tempest [53] provides weakly consistent in-memory replicated state, and network load balancers [14] perform weakly consistent request routing.

Weak consistency forces developers to confront a well-known set of hard problems, such as implementing application-specific reconciliation for divergent replicas [11], or using a last-writer-wins policy that simply drops previously successful operations. Previous work has argued that these problems are manageable for particular applications: shopping carts in Dynamo [18], dynamic personalized content in the Ninja [49] project, and academic citation search in OverCite [24] are notable examples. However, strong consistency provides a much simpler programming model for developers, and it is a better match for the desired behavior of many applications (e.g., three of the four applications built using the PRS).

Weakly consistent systems do not preclude a strongly consistent programming model, but to get this simpler programming model, one has to add some additional mechanism. For example, Amazon Web Services [7] and Google App Engine [19] provide only weakly consistent request routing to frontends, but they can support a strongly consistent programming model using their strongly consistent storage. In later sections, we compare the PRS to leveraging such storage systems.

5.2 Stored Procedures and Queries

Turning now to approaches that provide strong consistency, we consider integrating all the application logic into a scale-out storage system. This requires treating the application's state as hard state, regardless of the application's desired semantics. Typically, application logic executed within a storage system is either a stored procedure or a dynamically interpreted query. This approach allows leveraging the storage system's consistency, rather than requiring an additional mechanism like the PRS. There are many candidate scale-out storage systems for such an approach: DDS [48], BigTable [15], S3 [51], SimpleDB [52], the storage system described in Niobe [23]), GFS [47], Lustre [32], Farsite [2, 13]), etc.

Keeping application logic separate from the storage system has significant benefits. The first is performance: the storage system's use of replication and/or durability incurs overheads that are not needed for soft state. Debuggability and operability of the storage system are also improved by keeping application logic separated: if they are integrated, it becomes harder to assign responsibility for any performance and reliability problems that may arise. Restricting the storage system's programming model (e.g., by only allowing SQL) can mitigate this, but such restrictions make the application developer's job harder. The PRS avoids these difficult tradeoffs by providing a programming model for arbitrary operations on soft state with strong consistency.

5.3 Remixing Existing Infrastructure Services

Another approach to supporting soft-state scale-out datacenter applications is to extract functional subsystems from existing datacenter infrastructure services, modifying them as necessary to compose a new system that better targets soft-state applications. We will use Chubby [10], GFS and BigTable as examples of such existing services. To better compare these approaches to the PRS, we briefly review their relevant aspects.

In GFS, a centralized manager (the GFS master) partitions file chunks among GFS chunkservers. In BigTable, a centralized manager (the BigTable master) partitions the key space among tablet servers. In both cases, Chubby is used to elect the centralized manager from a pool of candidates. For BigTable, Chubby also provides liveness monitoring of tablet servers, helping the BigTable master to reassign key ranges from failed tablet servers. Chubby does not provide partitioning directly, as witnessed by the need to build both the GFS master and Bigtable master;

the closer analogues to the PRS are the BigTable or GFS masters combined with Chubby.

These existing technologies do not provide two critical aspects of the PRS: recovery notifications and reusable partitioning. One can imagine designing a new system supporting recovery notifications by extending the GFS or BigTable master and using it with Chubby. One can also imagine extending either the GFS or BigTable master to provide partitioning as a reusable abstraction that supports arbitrary operations and flexible consistency. However, it is the PRS's contribution to define these abstractions, to show their usefulness, and to demonstrate one possible implementation.

5.4 The Load/Operate/Store Pattern

Another approach to strong-consistency at the middle-tier is to keep all the state in the scale-out storage system between requests. The middle tier then loads the relevant state at the beginning of every request, operates on it, and stores it before returning. Optimistic concurrency control may be used to prevent simultaneous operations on the same state. By making the middle tier effectively stateless, consistency becomes trivial. This approach is described by Ling and Fox in their work on session state [28].

Involving the storage system on every operation sacrifices the performance benefits of soft state. For example, pure read operations at the middle tier must read from the storage system on every operation to guarantee consistency. For services where latency is critical to the user experience, this is highly undesirable. Avoiding these overheads requires caching data at the middle-tier with consistency guarantees, and the PRS enables this.

5.5 Static Partitioning

Static partitioning refers to keeping a static mapping of keys to middle-tier nodes at all frontends, e.g., by loading the mapping from a configuration file at startup. This provides strong consistency without the complexities of a lease manager, but it incurs a large operational overhead. For example, adding middle-tier nodes to accommodate load requires manual intervention to stop all frontends from sending requests on some of the keys while the mapping of keys to middle-tier nodes is updated. Maintenance windows allow this kind of upgrade, but they are highly undesirable in modern datacenter applications. The PRS avoids such drawbacks by providing dynamic membership and reconfiguration without sacrificing strong consistency.

5.6 Session State

The session state abstraction for middle-tier state provides strong consistency but restricts the sharing semantics: only the client that created some data can operate on the data. If a user logs in using a different client, their session state is unavailable. This restriction eliminates the need for maintaining the location of the state within the datacenter – it can instead be passed back to the caller in a cookie. BEA WebLogic [26] and the Session State Store [8] exploit these semantics to either simplify their implementation or provide new functionality. Trickles [6] exploits these semantics to push all state back to the client on every message.

The session state sharing semantics fundamentally restrict the middle tier developer. Instant messaging, other real-time communication applications, consistent caches, and three of the four deployed applications we presented in Section 3 all require a user experience that allows the use of multiple clients and enables sharing between different users. The PRS meets the need for simultaneous strong consistency and unrestricted sharing semantics in the middle tier.

5.7 Summary

None of these six approaches to scaling out datacenter applications simultaneously provides the programming model benefits of strong consistency, arbitrary operations and guaranteed recovery notifications, the manageability benefits of dynamic partitioning, and the performance benefits of operating on soft state. The PRS provides all these desirable properties simultaneously.

6 Related Work

In the previous section we analyzed a number of design alternatives to the PRS, and this led to comparing the PRS to systems such as BigTable, S3, DDS, GFS, Chubby and others. In this section we compare the PRS to systems that are less applicable to scaling out soft-state datacenter applications, but which have some other aspect in common with the PRS.

Systems designed for the wide-area have made very different tradeoffs from the PRS. Peer-to-peer overlays (e.g., Chord [21]), wide-area anycast systems (e.g., GIA [27] and Castro et al [35]), and DHTs (e.g., DHash [16]) provide routing and storage primitives, but none provide support for strong consistency. Various systems have explored moving computation to cache nodes in the context of the web. These systems have either provided no consistency beyond the standard web TTL-based mechanisms (e.g., Active Cache [42]) or they have only provided consistency on hard state (e.g., Na Kika [44]). Some wide area file-systems support strong consistency (e.g., OceanStore [50] and WheelFS [25]); the PRS differs in its support for soft state, arbitrary operations and recovery notifications.

Many distributed systems programming models have been proposed in the past; we only touch here on those

that resemble some aspect of the PRS. TACT [57], Lazy Replication [45], PRACTI [38] and others have explored letting clients make different availability-consistency tradeoffs within a single system, but they do not explore soft state or dynamically partitioning data across a set of servers. Sinfonia [34] proposes a programming model that encapsulates scale-out for building hard-state datacenter services, while the PRS is designed for soft state. Hilda [17, 39] proposes to improve middle-tier programming using a declarative query language analogous to SQL, and it uses a centralized database for concurrency control and persistence; the PRS does not have this scaling bottleneck. Membership services (e.g., SWIM [3]) and failure detection services (e.g., Heartbeat [33]) provide failure notifications about individual nodes; FUSE [22] provides failure notifications about FUSE groups, which typically correspond to data stored on multiple machines. By integrating partitioning with recovery notifications, the PRS can provide notifications that ranges of the key space may have failed, a better fit for the PRS's target class of applications.

7 Evaluation

The PRS is deployed in production, and as of April 2008, it supports live user traffic on over 250 frontend nodes and over 50 middle-tier nodes. In this deployment, failures have been quite rare. For example, over a one week period, only one middle-tier node lost its lease, and this was due to a permanent hardware failure. However, the PRS Manager was intentionally crashed once in the production environment to measure the impact on the system; Manager failover worked as designed, and the applications rapidly recreated their soft state.

The rest of this section evaluates a version of the PRS that extends the production version with two additional features: it performs adaptive load management and it moves ranges of items without requiring state recreation. For our experiments, it is configured to run with a single Manager, omitting the occasional call to a Paxos group for leader election. Our evaluation focuses on the scalability, availability, and adaptability of the PRS.

Our experimental cluster consists of 11 2.4 GHz Xeon dual-processor servers with hyper-threading enabled, 1 Gbps NICs, and 2-4 GB of RAM each. We dedicate one server to run the Manager, five servers to run a large number of frontends, and five servers to run a large number of middle-tier nodes. This small number of servers does not prevent evaluating the PRS at scale. First, the Manager is isolated on a single machine, just as it is in the production environment. The Manager's interactions with the frontends and middle-tier nodes are the same as if the frontends and middle-tier nodes had been spread out one node per machine. Because the Manager determines how far the PRS can scale, the experimental configuration allows realistically evaluating this aspect of the PRS. Second, all frontends and middle-tier nodes have to communicate across a machine boundary; co-locating many frontends on one machine and many middle-tier nodes on another machine does not eliminate any communication overhead. On each physical server, all frontend or middle-tier nodes run within a single process to reduce memory overhead, but they all create their own threads and their own instances of the Lookup or Owner objects.

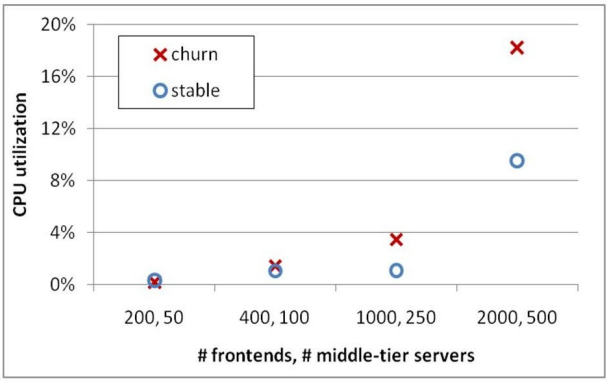

Figure 7. *The CPU load on the manager as we increase the number of frontends and middle-tier servers.*

The test application we run on the frontends and middle-tier nodes is a simplified version of Instant Messaging. In the test application, frontends can issue inserts, updates and reads, while the middle-tier servers perform these operations and store the user presence (IM address, presence status and last known IP address). We have found that the deployed applications using the PRS typically have about four to five times as many frontends as middle-tier nodes, and so we chose a similar ratio in our evaluation.

7.1 Scalability

We begin by evaluating the scalability of the PRS both with and without failures. Figures 7 and 8 show the CPU and network load at the Manager as the number of frontends and middle-tier nodes is increased. In the absence of frontend/middle-tier failures, the CPU and network load on the Manager are quite small, and the Manager easily scales to 2000 frontends and 500 middle-tier nodes. The Manager's current bottleneck to scaling further is memory consumption during churn, when nodes are rapidly added and/or removed – this is not a case that the current implementation is extensively optimized for. In the production environment, the Manager maintains less than 1 MB of state when there is no change in the system. In our experiments, we similarly find that memory consumption is quite small except when the set of nodes in the system is changing. However, as described in Section 4.5, scal-

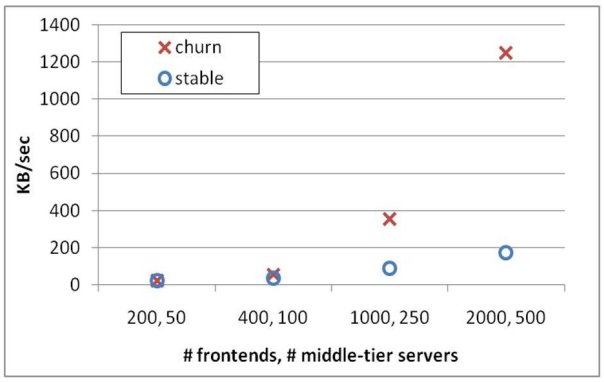

Figure 8. *The network load on the manager as we increase the number of frontends and middle-tier servers.*

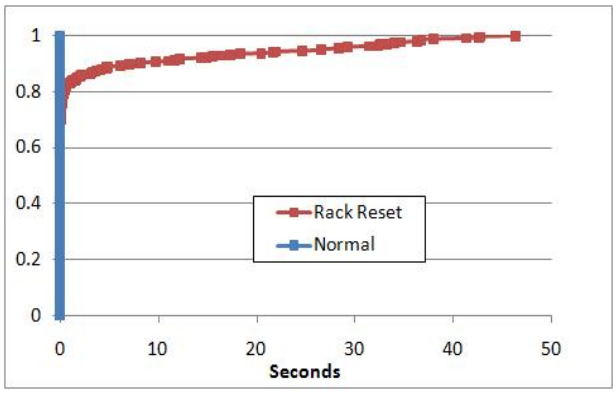

Figure 9. *CDF of the latency for frontends to complete operations: normal operation compared to immediately after a rack of servers resets.*

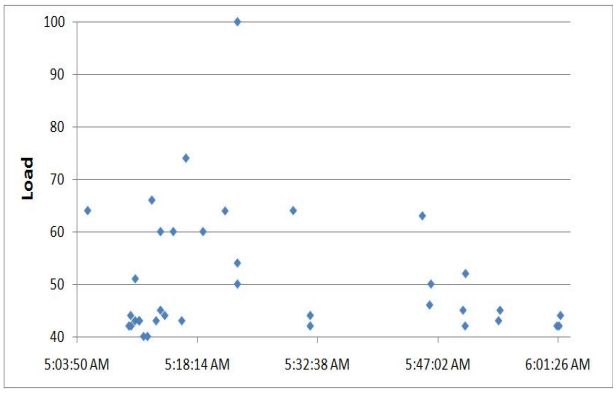

Figure 10. *Servers that were above the threshold for adaptive load management over the course of an hour.*

ing to 2500 nodes meets our current needs. Also, because the Manager scales to these 2500 nodes on a 32-bit OS running on a server purchased in 2003, we are optimistic that further scaling is possible; the production version is 64-bit.

To measure the effect of failures, we cause each node to randomly restart with a mean node lifetime of 8 hours. At the largest scale in our evaluation, 2500 machines, a mean node lifetime of 8 hours corresponds to about one machine failure every 12 seconds. This is significantly more often than is typical in a managed datacenter environment – we have observed over the past few months in our production environment that datacenter machines typically run for weeks between restarts. We choose this adversarial restart rate simply to show that the Manager can tolerate far more than a realistic server restart rate; even here, the PRS can scale up to 2500 nodes.

7.2 Availability

The primary impact of failure is lack of availability. Because the test application frontends republish data and retry failed requests, this availability penalty translates to additional latency. For example, if some data is unavailable for 30 seconds, an operation that normally would complete in 0.1 seconds may instead require 30.1 seconds. The particular strategy implemented by the test application frontends is to retry once every 30 seconds for up to 5 minutes.

We choose a scenario where an entire rack of servers (specifically, 50 servers) is accidentally powered down and then powered back up several minutes later. We start measurement at the moment the rack is brought back online, by which point the old leases have already expired. In this experiment, the entire middle tier is only 250 servers, and there are 1000 frontends. The keys assigned to the 50 new middle-tier servers are unavailable for a short period of time as new leases are granted. We see in Figure 9 that about 1 out of 5 of the keys has a latency of access that stretches out over tens of seconds. This 1-in-5 ratio is what we expect from the fact that the 50 middle-tier nodes we restarted are $1/5^{th}$ of the total number of middle-tier nodes.

Complete reset of an entire rack of servers is a major operational event. Restoring availability in less than a minute on the subset of the key space that was lost is fast compared to even just the time until the middle-tier servers leases expire (one minute). This experiment demonstrate the PRS's effectiveness at restoring availability quickly after a major failure.

7.3 Adaptive Load Management

To evaluate the PRS's ability to do adaptive load management, we generate a Zipf distribution of skewed key popularity, and all the frontends direct their operations to keys chosen according to this distribution. We set the Zipf parameter as $\alpha = 0.8$. Figure 10 begins when the workload starts. The average load during the experiment is 20%, but several servers are significantly more than 40% loaded. Figure 10 shows these heavily loaded

servers. Over the course of the hour, the adaptive load management algorithm reacts to the skew in the workload and re-distributes responsibility away from highly loaded servers, resulting in significantly fewer by the end of the hour.

8 Conclusion

Datacenter applications are of enormous commercial importance. Prior work has made great strides in providing better scale-out software infrastructure, but this infrastructure has not targeted the middle tier: *soft-state* datacenter applications. Previous work on DHTs demonstrated the value of providing request routing without storage [5, 35]. The PRS takes a similar philosophical approach, but we have found that soft-state datacenter applications additionally require strong consistency and recovery notifications. By providing these, the PRS makes it easier to build and deploy compelling soft-state datacenter applications. The PRS has been validated on a number of levels: other developers have built four applications using the PRS; these applications have been deployed and are serving real customer requests; and we have evaluated an experimental version of the PRS. In future work, we plan to explore whether some of the PRS ideas can also be applied in a hard-state context.

References

[1] A. Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, Massachusetts Institute of Technology, 1999.

[2] A. Adya et al. Farsite: federated, available, and reliable storage for an incompletely trusted environment. *OSDI*, 2002.

[3] A. Das et al. SWIM: Scalable Weakly consistent Infection-style process group Membership protocol. *DSN*, 2002.

[4] A. Fox et al. Cluster-Based Scalable Network Services. *SOSP*, 1997.

[5] A. Rowstron et al. Scribe: The design of a large-scale event notification infrastructure. *NGC*, 2001.

[6] A. Shieh et al. Trickles: A Stateless Network Stack for Improved Scalability, Resilience, and Flexibility. *NSDI*, 2005.

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

[8] B. Ling et al. Session state: beyond soft state. *NSDI*, 2004.

[9] Bugzilla. http://wiki.mozilla.org.

[10] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. *OSDI*, 2006.

[11] D. Terry et al. Managing update conflicts in Bayou, a weakly connected replicated storage system. *SOSP*, 1995.

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

[13] J. Douceur and J. Howell. Distributed directory service in the Farsite file system. *OSDI*, 2006.

[14] J. Elson and J. Howell. Handling Flash Crowds from Your Garage. *USENIX ATC*, 2008.

[15] F. Chang et al. Bigtable: A Distributed Storage System for Structured Data. *OSDI*, 2006.

[16] F. Dabek et al. Wide-area cooperative storage with CFS. *SOSP*, 2001.

[17] F. Yang et al. Hilda: A High-Level Language for Data-DrivenWeb Applications. *ICDE*, 2006.

[18] G. DeCandia et al. Dynamo: amazon's highly available key-value store. *SOSP*, 2007.

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

[20] Google Talk. http://www.google.com/talk.

[21] I. Stoica et al. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM*, 2001.

[22] J. Dunagan et al. FUSE: lightweight guaranteed distributed failure notification. *OSDI*, 2004.

[23] J. MacCormick et al. Niobe: A Practical Replication Protocol. *ACM Transactions on Storage*, 3(4):1–43, 2008.

[24] J. Stribling et al. Overcite: A cooperative digital research library. *NSDI*, 2006.

[25] J. Stribling et al. Don't Give Up on Distributed File Systems. *IPTPS*, 2007.

[26] D. Jacobs. Distributed computing with BEA WebLogic server. *CIDR*, 2003.

[27] D. Katabi and J. Wroclawski. A framework for scalable global IP-anycast (GIA). *SIGCOMM*, 2001.

[28] B. Ling and A. Fox. The case for a session state storage layer. *HOTOS*, 2003.

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

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

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

[32] Lustre. http://www.lustre.com.

[33] M. Aguilera et al. Heartbeat: A Timeout-Free Failure Detector for Quiescent Reliable Communication. *Workshop on Distributed Algorithms*, 1997.

[34] M. Aguilera et al. Sinfonia: a new paradigm for building scalable distributed systems. *SOSP*, 2007.

[35] M. Castro et al. Scalable application-level anycast for highly dynamic groups. *NGC*, 2003.

[36] M. Isard et al. Dryad: distributed data-parallel programs from sequential building blocks. *EuroSys*, 2007.

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

[38] N. Belaramani et al. PRACTI replication. *NSDI*, 2006.

[39] N. Gerner et al. Automatic client-server partitioning of data-driven web applications. *SIGMOD*, 2006.

[40] NCache. http://www.alachisoft.com.

[41] Office Communicator. http://office.microsoft.com/communicator.

[42] P. Cao et al. Active Cache: caching dynamic contents on the Web. *Distributed Systems Engineering*, 6(1):43–50, 1999.

[43] Q. Luo et al. Middle-tier database caching for e-business. *SIGMOD*, 2002.

[44] R. Grimm. Na Kika: Secure service execution and composition in an open edge-side computing network. *NSDI*, 2006.

[45] R. Ladin et al. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10(4):360–391, 1992.

[46] RSS. http://www.rssboard.org.

[47] S. Ghemawat et al. The Google file system. *SOSP*, 2003.

[48] S. Gribble et al. Scalable, distributed data structures for internet service construction. *OSDI*, 2000.

[49] S. Gribble et al. The Ninja architecture for robust Internet-scale systems and services. *Computer Networks*, 35(4):473–497, 2001.

[50] S. Rhea et al. Pond: the OceanStore prototype. *FAST*, 2003.

[51] S3. http://aws.amazon.com/s3.

[52] SimpleDb. http://aws.amazon.com/simpledb.

[53] T. Marian et al. Tempest: Soft State Replication in the Service Tier. *DSN-DCCS*, 2008.

[54] W. Weihl. Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types. *ACM Transactions on Programming Languages and Systems*, 11(2):249–282, 1989.

[55] Wikipedia. http://www.wikipedia.org.

[56] Yahoo Messenger. http://messenger.yahoo.com.

[57] H. Yu and A. Vahdat. The costs and limits of availability for replicated services. *SOSP*, 2001.