Session State: Beyond Soft State

Benjamin C. Ling, Emre Kıcıman and Armando Fox
{bling, emrek, fox}@cs.stanford.edu

ABSTRACT

The cost and complexity of administration of large systems has come to dominate their total cost of ownership. Stateless and soft-state components, such as Web servers or network routers, are relatively easy to manage: capacity can be scaled incrementally by adding more nodes, rebalancing of load after failover is easy, and reactive or proactive ("rolling") reboots can be used to handle transient failures. We show that it is possible to achieve the same ease of management for the state-storage subsystem by subdividing persistent state according to the specific guarantees needed by each type. While other systems [21, 19] have addressed persistent-until-deleted state, we describe SSM, an implemented store for a previously unaddressed class of state – user-session state – that exhibits the same manageability properties as stateless or soft-state nodes while providing firm storage guarantees. In particular, any node can be proactively or reactively rebooted at any time to recover from transient faults, without impacting online performance or losing data. We then exploit this simplified manageability by pairing SSM with an application-generic, statistical-anomaly-based framework that detects crashes, hangs, and performance failures, and automatically attempts to recover from them by rebooting faulty nodes as needed. Although the detection techniques generate some false positives, the cost of recovery is so low that the false positives have limited impact. We provide microbenchmarks to demonstrate SSM's built-in overload protection, failure management and self-tuning. Finally, we benchmark SSM integrated into a production enterprise-scale interactive service to demonstrate that these benefits need not come at the cost of significantly decreased throughput or response time.

1. INTRODUCTION

The cost and complexity of administration of systems is now the dominant factor in total cost of ownership for both hardware and software [34]. In addition, since human operator error is the source of a large fraction of outages [6], attention has recently been focused on simplifying and ultimately automating administration and management to reduce the impact of failures [14, 21], and where this is not fully possible, on building self-monitoring components [22]. However, fast, accurate detection of failures and recovery management remains difficult, and initiating recovery on "false alarms" often incurs an unacceptable performance penalty; even worse, initiating recovery on "false alarms" can cause incorrect system behavior when system invariants are violated (e.g., only one copy of X should be running at a given

time) [22].

Operators of both network infrastructure and interactive Internet services have come to appreciate the high-availability and maintainability advantages of stateless and soft-state [36] protocols and systems. The stateless Web server tier of a typical three-tier service [4] can be managed with a simple policy: misbehaving components can be reactively or proactively rebooted, which is fast since they typically perform no special-case recovery, or can be removed from service without affecting correctness. Further, since all instances of a particular type of stateless component are functionally equivalent, overprovisioning for load redirection [4] is easy to do, with the net result that both stateless and soft-state components can be overprovisioned by simple replication for high availability.

However, this simplicity does not extend to the stateful tiers. Persistent-state subsystems in their full generality, such as filesystem appliances and relational databases, do not typically enjoy the simplicity of using redundancy to provide failover capacity as well as to incrementally scale the system. We argue that the ability to use these HA techniques can in fact be realized if we subdivide "persistent state" into distinct categories based on durability and consistency requirements. This has in fact already been done for several large Internet services [1, 33, 42, 30], because it allows individual subsystems to be optimized for performance, fault-tolerance, recovery, and ease-of-management.

In this paper, we make three main contributions:

1. We focus on *user session state*, which must persist for a bounded-length user session but can be discarded afterward. We show why this class of data is important, how its requirements are different from those for persistent state, and how to exploit its consistency and workload requirements to build a distributed, self-managing and recovery-friendly session state storage subsystem, SSM. SSM provides a probabilistic bounded-durability storage guarantee for such state. Like stateless or soft-state components, any node of SSM can be rebooted without warning and without compromising correctness or performance of the overall application. No node performs special-case recovery code. Additional redundancy allows multiple simultaneous failures. As a result, SSM can be managed using simple, "stateless tier" HA techniques for incremental scaling, fault tolerance, and overprovisioning.

2. We demonstrate the resulting simplicity of recovery management by combining SSM with a generic statistical-monitoring failure detection tool. Pinpoint

looks for "anomalous" behaviors (based on histori-
cal performance or deviation from the performance of
peer nodes) and immediately coerces any misbehav-
ing node to crash and reboot. Although false posi-
tives do occur, the simplicity and low cost of recov-
ery (crash and reboot) makes them a minor considera-
tion, greatly simplifying SSM's failure detection and
management strategy. Combined with SSM's addi-
tive increase/multiplicative decrease admission control
that protects it from being driven to saturation under
overload, the result is a largely self-managing subsys-
tem using entirely generic detection and recovery tech-
niques.

3. We summarize the design choices and lessons, along
with the system architecture requirements that allow
the approach to work, and highlight design principles
that can be applied to other systems.

In Section 2, we describe the properties of the subcategory
of session state on which we focus, as well as its associated
workload, and existing solutions. In Section 3, we present
the design and implementation of SSM, a recovery-friendly
and self-managing session state store. In Section 4, we de-
scribe the integration of SSM with Pinpoint to enable the
system to be self-healing. In Section 5, we present a se-
ries of benchmarks demonstrating the recovery-friendly and
self-managing features of SSM. In Section 6, we insert SSM
into an existing production internet application and com-
pare its performance, failure, and recovery characteristics
with the original implementation. In Section 7, we discuss
the general design principles extracted from the design and
implementation of SSM. We then discuss related and future
work, and then conclude.

2. WHY SESSION STATE?

In networking systems, signaling systems for flow state [9]
fall in between two extremes: hard-state and soft-state [35].
In hard-state systems, state is explicitly written once and
remains written unless explicitly removed; special mecha-
nisms exist to remove orphaned state. In contrast, in soft-
state systems, state automatically expires unless refreshed
by the writer, so no such special mechanisms are needed.
Session state lies somewhere in between: unlike hard state,
its maximum overall lifetime and inter-access interval are
bounded, so persistence guarantees need only respect those
bounds; unlike soft state, it cannot be reconstructed from
other sources if lost, unless the user is asked to repeat all
steps that led to the construction of the state.

Nearly all nontrivial Internet services maintain session
state, but they either store it as hard state because that
is what most storage systems provide, or store it ephemer-
ally (in RAM of otherwise stateless components) because
it is cheaper and faster. The former is overkill, the latter
does not provide adequate guarantees of persistence, espe-
cially in the face of transient failures. Table 2 compares and
contrasts the different types of state.

For the remainder of this paper, we will use the term
"session state" to refer to the subcategory of user-session

	Hard/Persistent	Soft/Session
Write Method	Write once	Refresh
Deletion Method	Explicit	Expiration
Orphan Cleanup	Manual	Automatic

Table 1: Key differences among hard, persistent, soft, and
session state.

state we now describe. Many associate session state with
"shopping cart," but the class of session state we address
is significantly broader than just shopping carts. An ex-
ample of application session state that we address includes
user workflow state in enterprise applications. In particular,
today's enterprise applications, such as those in J2EE [41],
are often accessed via a web browser. All application state,
such as context and workflow, is stored on the server and is
an example of what we are calling session state. In essence,
user workflow state in enterprise applications is equivalent to
temporary application state on a desktop application. An-
other example of session state is travel itineraries from on-
line travel sites, which capture choices that users have made
during the shopping process. Shopping carts, used in nearly
all online shopping sites, can also be an example of session
state.[1]

To understand how session state is typically used, we use
the example of a user working on a web-based enterprise-
scale application to illustrate the typical flow sequence. A
large class of applications, including J2EE-based and web
applications in general, use the interaction model below:

- User submits a request, and the request is routed to
  a stateless application server. This server is part of
  what is often called the middle-tier.

- Application server retrieves the full session state for
  user (which includes the current application state).

- Application server runs application logic

- Application server writes out entire (possibly modi-
  fied) session state

- Results are returned to the user's browser

Session state is in the critical path of each interaction,
since user context or workflow is stored in session state. Loss
of session state is seen as an application failure to the end
user, which is usually considered unacceptable to the ser-
vice provider. Typical session state size is between 3K-200K
bytes [40].

Some important properties/qualities of the session state
we focus on are listed below. Session state:

1. Is accessed in a serial fashion by a single user (no con-
   current access). Each user reads her own state, usu-
   ally keyed by a deterministic function of the user's ID,
   so an advanced query mechanism to locate the user's

---

[1]In practice, some e-commerce sites store shopping cart con-
tents in persistent state so that unpurchased items in the
cart will persist across visits

state is unnecessary. Furthermore, the client is typically responsible for storing the necessary metadata to retrieve the state.

2. Is semi-persistent. Session state must be present for a fixed interval T, the application-specific session timeout (usually on the order of minutes to hours), but should expire after T.

3. Is written out in its entirety, and usually updated on every interaction.

Given these properties, the functionality necessary for a session state store can be greatly simplified, relative to fully-general ACID guarantees provided by a relational datbase. Each simplification corresponds to an entry in the previous numbered list:

1. No synchronization is needed. Since the access pattern corresponds to an access of a single user making serial requests, no conflicting accesses exist, and hence race conditions on state access are avoided, which implies that locking is not needed. In addition, a single-key lookup API is sufficient. Since state is keyed to a particular user and is usually only accessed by that user, a general query mechanism is not needed.

2. State stored by the repository need only be semi-persistent – a temporal, lease-like [17] guarantee is sufficient, rather than the durable-until-deleted guarantee that is made in ACID [18].

3. Atomic update is sufficient for correctness, since partial writes do not occur. Once session state is modified, any of its previous values may be discarded.

Relative to the specific requirements of session state, SSM does, in a sense, provide ACID guarantees: atomicity and bounded durability are provided, and consistency and isolation are made trivial by the access pattern.

As a generalization, the class of state that we address need not necessarily be single-user; as long as state ownership is explictly passed between parties, which is common in today's enterprise applications [23], the techniques discussed in this paper applies.

2.1 Existing Solutions

Frequently, enterprises use either a relational database (DB) or a filesystem or filesystem appliance (FS) to store session state, because they already use a DB or FS for persistent state. This potentially simplifies management, since only one type of administrator is needed. However, there are several drawbacks to using either a DB or FS, besides the costs of additional licenses, which are detailed in previous work [28].

In addition, DB and file systems are well-known to be difficult to administer and tune. Each must be configured and tuned for a particular workload. Even for a skilled and costly administrator, this remains a difficult and often error-prone process that is repeated as the workload changes.

In contrast, in-memory solutions avoid several of the drawbacks of FS/DB, and are generally faster than FS/DB
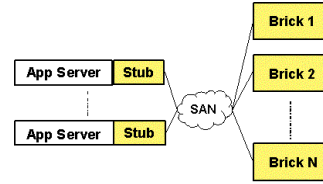

Figure 1: **Architecture of SSM. Stubs are stateless and are used by application servers to read and write state. Bricks are diskless components that store session state.**

oriented solutions, but make it difficult to provide both performance and correctness guarantees. Existing in-memory solutions require a user to be pinned to a particular server, which prevents the application-processing tier from remaining truly stateless, since each server must both run application logic and store session state. Because of pinning, load-balancing can only be done across users but not across requests, and hotspots are harder to alleviate. A detailed discussion of in-memory solutions can be found in previous work [27, 28].

3. PROPOSED SOLUTION: SSM

We now describe the design and implementation of SSM, a lightweight session-state store. We make the following assumptions about the operating environment, which are typical of large-scale services [4]: a physically secure cluster interconnected by a commercially-available high-throughput, low-latency system area network (SAN); and an uninterruptible power supply to reduce the probability of a system-wide simultaneous hardware outages. We assume that a network partition is not possible. If a network partition occurs, we assume it is likely that a catastrophic failure has occurred; SSM is unable to recover from catastrophic faults. The Java prototype consists of 872 semicolons and runs on the UC Berkeley Millennium Cluster, consisting of 42 IBM xSeries 330 1U rackmounted PCs, each running Linux 2.4.18 on Dual 1.0 GHz Intel Pentium III CPUs and 1.5GB ECC PC133 SDRAM, connected via Gigabit Ethernet.

3.1 SSM Overview

SSM has two components: bricks and stubs. Bricks, each consisting of a CPU, network interface and RAM (no disk), provide storage; stubs dispatch read and write requests to bricks. Figure 1 shows the basic architecture of SSM.

On a client request, the application server will first ask the stub to read the client's session state, and after application processing, to write out the new session state. The general strategy employed by the stub for both reads and writes is "send to many bricks, wait for few to reply," to avoid having a request depend on any *specific* brick. Upon completion of the write request, a cookie containing the ids of the bricks that processed the write is sent back to the client.

A brick stores session state objects using an in-memory hash table. Each brick sends out periodic multicast beacons to indicate that it is alive. Each stub keeps track of which bricks are currently alive by listening to the beacons; stubs receive the announcements and make connections to
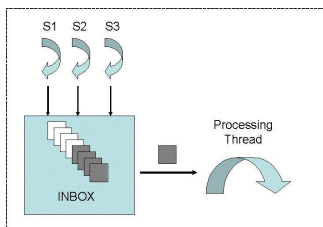
Figure 2: **Internal structure for brick. S1, S2, and S3 are threads that read incoming requests off of sockets from stubs 1, 2 and 3, and place the requests into an inbox. One main processing thread fetches requests from the inbox, and maintains the internal data structures.**

the bricks via TCP/IP. We choose TCP/IP as the communication mechanism for read/write request traffic because reliable and ordered messaging enables easy prototyping.

When a stub contacts a brick, a stream is created between the two, which lasts until either component ceases executing. Each brick has a list of streams corresponding to the stubs that has contacted it. The brick has one main processing thread, which fetches requests from a shared inbox, and handles the request by manipulating the internal data structures. A single monitor thread handles the internal data structures. In addition, the brick has an additional thread for each stub communicating with the brick; each of these communication threads puts requests from the corresponding stub into the shared inbox. Figure 2 shows an illustration of the brick thread structure.

The write function `Write(HashKey H, Object v, Expiry E)` exported by the stub returns a cookie if the write succeeds or throws an exception otherwise. The returned cookie is passed back to the client (Web browser) for storage, as it stores important metadata that will be necessary for the subsequent read. Existing solutions for session state also rely on storing this metadata on the client.

The read function `Read(Cookie C, HashKey H)` returns the most recently written value for hash key H, or throws an exception if the read fails. If a read/write returns to the application, then it means the operation was successful. On a read, SSM guarantees that the returned value is the most recently written value by the user.

The stub dispatches write and read requests to the bricks. Before we describe the algorithm describing the stub-to-brick interface, let us define a few variables. Call W the write group size. Call R the read group size. On a write request, a stub *attempts* to write to W of the bricks; on a read request, it attempts to read from R bricks.

Define WQ as the size of the write set, which is the minimum number of bricks that must return "success" to the stub before the stub returns to the caller. $WQ - 1$ is the number of simultaneous brick failures that the system can tolerate before possibly losing data. R is the size of the candidate read set; only 1 brick need to reply to service a read request succesfully. Note that $1 \leq WQ \leq W$ and $1 \leq R \leq WQ$. In practice, we use $W = 3, WQ = 2, R = 2$.

Lastly, call t the request timeout interval, the time that the stub waits for a brick to reply to an individual request,

usually on the order of milliseconds. t is different from the session expiration, which is the lifetime of a session state object, usually on the order of minutes. We use t and *timeout* interchangeably in this paper. In practice, t is a rough upper bound on the time an application is willing to wait for the writing and retrieval of a session state object, usually on the order of tens to hundreds of milliseconds since session state manipulation is in the critical path of client requests.

3.2 Basic Read/Write Algorithm

The basic write algorithm can be described as "write to many, wait for a few to reply." Conceptually, the stub writes to more bricks than are necessary, namely W, and only waits for WQ bricks to reply. Sending to more bricks than are necessary allows us to harness redundancy to avoid performance coupling; a degraded brick will not slow down a request. In the case where WQ bricks do not reply within the timeout, the stub throws an exception so that the caller can handle the exception and act accordingly (e.g., signal to the end user to come back later), rather than being forced to wait indefinitely. This is part of the system applying backpressure. The algorithm is described below:

```
Cookie Write(HashKey H, Object v, Expiry E)
    throws SystemOverloadedException

0  Time wakeup = getCurrentTime() + timeout;
1  int cs = checksum(H, v, E);
2  Brick[] repliedBricks = {};
3  Brick[] targetBricks = chooseRandomBricks(W);
4  foreach brick in targetBricks
5     do WriteBrick(H, v, E, cs);
6  while (repliedBricks.size < WQ)
7     Time timeleft = wakeup - getCurrentTime();
8     Brick replied = receiveReply(timeleft);
9     if (replied == null) break;
10     repliedBricks.add(replied);
11 if (repliedBricks.size < WQ)
12    throw new SystemOverloadedException();
13 int check = checksum(H, repliedBricks, cs, E);
14 return new Cookie(check, H, repliedBricks, cs, E);
```

The stub handles a read by sending the read to R bricks, waiting for only 1 brick to reply:

```
Object Read(Cookie c) throws CookieCorruptedExcept,
  SystemOverloadedExcept, StateExpiredExcept,
  StateCorruptedExcept

0  int check = c.checksum;
1  int c2 = checksum(c.H, c.repliedbricks, c.cs, c.E);
2  if (c2 != check)
3     throw new CookieCorruptedException();
4  if (isExpired(c.E))
5     throw new StateExpiredException();
6  Brick[] targetBricks = c.repliedBricks;
7  foreach brick in targetBricks
8     do RequestBrickRead(H, E, cs);
9
10 Brick replied = receiveReply(timeout);
11 if (replied == null)
12    throw new SystemOverloadedException();
13 retval = replied.objectValue;
14 if (c.cs != checksum(retval))
15    throw new StateCorruptedException();
16 return retval;
```

3.3 Garbage Collection

For garbage collection of bricks, we use a method seen in generational garbage collectors [8]. For simplicity, earlier we

described each brick as having one hash table. In reality, it has a set of hash tables; each hash table has an expiration. A brick handles writes by putting state into the table with the closest expiration time after the state's expiration time. For a read, because the stub sends the key's expiration time, the brick knows which table to look in. When a table's expiration has elapsed, it is discarded, and a new one is added in its place with a new expiration.

3.4 Load capacity discovery and admission control

In addition to the basic read/write algorithm, each stub maintains a sending window (SW) for each brick, which the stub uses to determine the maximum number of in-flight, non-acked requests the stub can send to the recipient brick.

The stub implements a additive-increase, multiplicative-decrease (AIMD) algorithm for maintaining the window; the window size is additively increased on a successful ack and multiplicatively decreased on a timeout. When a request times out, the stub reduces its sending window to the brick accordingly. In the case when the number of in-flight messages to a brick is equal to the SW, any subsequent requests to that brick will be disallowed until the number of in-flight messages for that brick is less than the SW. If a stub cannot find a suitable number of bricks to send the request to, it throws an exception to the caller indicating that the system is overloaded.

Each stub stores temporary state for only the requests that are awaiting responses from bricks. The stub performs no queueing for incoming requests from clients. For any request that cannot be serviced because of overload, the stub rejects the request *immediately*, throwing an exception indicating that the system is temporarily overloaded.

In addition, each brick performs admission control; when a request arrives at the brick, it is put in a queue. If the request timeout has elapsed by the time that the brick has dequeued the request, the request is disregarded and the brick continues to the service the next queued request.

Note that the windowing mechanism at the stub and the request rejection at the brick protect the system in two different ways. At the stub, the windowing mechanism prevents any given stub from saturating the bricks with requests. However, even with the windowing mechanism, it is still possible for multiple stubs to temporarily overwhelm a brick (e.g. the brick begins garbage collection and can no longer handle the previous load). At the brick, the request rejection mechanism allows the brick to throw away requests that have already timed out in order to "catch up" to the requests that can still be serviced in a timely manner.

3.5 Failure and Recovery

In SSM, recovery of any component that has failed is simple; a restart is all that is necessary to recover from a non-persistent failure. No special case recovery code is necessary.

On failure of a client, the user perceives the session as lost, e.g., if the browser crashes, a user does not necessarily expect to be able to resume his interaction with a web application. If cookies for the client are persisted, as is often the case, then the client may be able to resume his session when the browser is restarted.

On failure of a stateless application server, a restart of the server is sufficient for recovery. After restart, the stub on the server detects existing bricks from the beacons and can reconstruct the table of live bricks. The stub can immediately begin handling both read and write requests; to service a read request, the necessary metadata is provided by the client in the cookie. To service a write request, all that is required is a list of WQ live bricks.

On failure of a brick, a simple restart of the brick is sufficient for recovery. The contents of its memory are lost, but since each hash value is replicated on $WQ - 1$ other bricks, no data is lost. The next update of the session state will recreate WQ new copies; if $WQ - 1$ additional failures occur before then, data may be lost. Section 3.7 sketches an analysis of the minimum required mean-time-to-failure (MTTF) of a brick to achieve a given level of availability under these assumptions.

A side effect of having simple recovery is that clients, servers, and bricks can be added to a production system to increase capacity. For example, adding an extra brick to an already existing system is easy. Initially, the new brick will not service any read requests since it will not be in the read group for any requests. However, it will be included in new write groups because when the stub detects that a brick is alive, the brick becomes a candidate for a write. Over time, the new brick will receive an equal load of read/write traffic as the existing bricks, since load balancing is done per request and not per hash key.

3.6 Recovery Philosophy

Previous work has argued that rebooting is an appealing recovery strategy in cases where it can be made to work [7]: it is simple to understand and use, reclaims leaked resources, cleans up corrupted transient operating state, and returns the system to a known state. Even assuming a component is reboot-safe, in some cases multiple components may have to be rebooted to allow the system as a whole to continue operating; because inter-component interactions are not always fully known, deciding *which* components to reboot may be difficult. If the decision of which components to reboot is too conservative (too many components rebooted), recovery may take longer than really needed. If it is too lenient, the system as a whole may not recover, leading to the need for another recovery attempt, again resulting in wasted time.

By making recovery "free" in SSM, we largely eliminate the cost of being too conservative. If an SSM brick is *suspected* of being faulty – perhaps it is displaying fail-stutter behavior [3] or other characteristics associated with software aging [15] – there is essentially no penalty to reboot it prophylactically. This can be thought of as a special case of fault-model enforcement: treat any performance fault in an SSM brick as a crash fault, and recover accordingly. In recent terminology, SSM is a *crash-only* subsystem [7].

3.7 Brick MTTF vs. Availability

Before presenting experimental results, we illustrate the relationship between MTTF for an individual brick and the availability of data for SSM as a whole. We assume indepen-
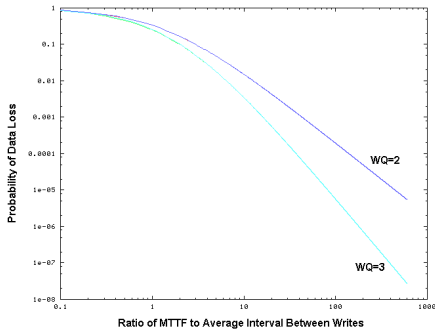
Figure 3: **Probability of data loss with WQ=2 and 3. The x-axis is the ratio of MTTF to the session expiration time. The y-axis is the probability that all WQ copies are lost before the subsequent write.**

dent failures; when failures are correlated in Internet server clusters, it is often the result of a larger catastrophic failure that session state would not be expected to survive [22]. We describe a natural extension to SSM to survive site failures in section 8.

Let brick failure be modeled by a Poisson process with rate μ (i.e., the brick's MTTF is $1/\mu$), and let writes for a particular user's data be modeled by a Poisson process with rate λ. (In other words, in practice $1/\lambda$ is the session expiration time, usually on the order of minutes or tens of minutes.) Then $\rho = \lambda/\mu$ is intuitively the ratio of the write rate to the failure rate, or equivalently, the ratio of the MTTF of a brick vs. the write interarrival time.

A session state object is lost if all WQ copies of it are lost. Since every successful write re-creates WQ copies of the data, the object is *not* lost if at most $WQ - 1$ failures occur between successive writes of the object. Equations 1 and 2 show this probability for $WQ = 3$ and $WQ = 2$ respectively; figure 3 shows the probabilities graphically.

$$P_{noloss}^{WQ=3} = \frac{\rho(\rho^2 + 6\rho + 11)}{(\rho + 1)(\rho + 2)(\rho + 3)} \qquad (1)$$

$$P_{noloss}^{WQ=2} = \frac{\rho(\rho + 3)}{(\rho + 1)(\rho + 2)} \qquad (2)$$

Table 2 summarizes the implication of the equations in terms of "number of nines" of availability. For example, to achieve "three nines" of availability, or probability 0.999 that data will not be lost, a system with $WQ = 2$ must be able to keep an individual brick from crashing for an interval that is 43.3 times as long as the average time between writes. Adding redundancy ($WQ = 3$) reduces this, requiring an MTTF that is only 16.2 times the average time between writes. For example, if the average time between writes is 5 minutes and $WQ = 3$, three nines can be achieved as long as brick MTTF is at least 81 minutes.

Another way to look at it is to fix the ratio of MTTF to the write interval. Figure 4 sets this ratio to 10 (intuitively, this means roughly that writes occur ten times as often as failures) and illustrates the effect of adding redun-

	$WQ = 2$	$WQ = 3$
1 Nine	3	2
2 Nines	12.7	6.5
3 Nines	43.3	16.2
4 Nines	140	37.2
5 Nines	446.8	82.4

Table 2: For WQ=2 and 3, the necessary ratio of MTTF to average interval between writes in order for probability of a subsequent write to achieve a certain number of nines
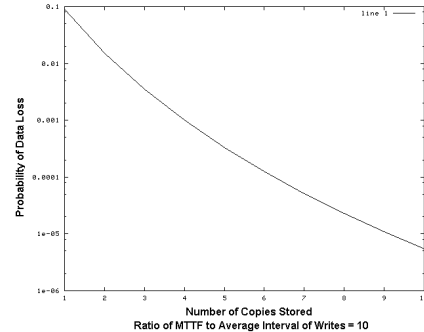

Figure 4: **We fix the ratio of MTTF to the average interval between writes to 10. The x-axis represents the number of copies written. The y-axis represents the probability that all copies are lost.**

dancy (modifying WQ) on data loss.

4. PINPOINT + SSM = SELF-HEALING

Pinpoint is a framework for detecting likely failures in componentized systems. To detect failures, a Pinpoint server dynamically generates a model of the "good behavior" of the system. This good behavior is based on both the past behavior and the majority behavior of the system, under the assumption that most of the time, most of the system is likely to be behaving correctly.

When part of the system deviates from this believed good behavior, Pinpoint interprets the anomaly as a possible failure. Once Pinpoint notices a component generating a threshold number of these anomalies, Pinpoint triggers a restart of the component.

To detect failures in bricks, Pinpoint monitors each brick's vital statistics, shown in Table 3. Each brick sends its own statistics to the Pinpoint server at one-second intervals. Statistics are divided into *activity* and *state* statistics.

Activity statistics, e.g., the number of processed writes, represent the rate at which a brick is performing some activity. When Pinpoint receives an activity statistic, it compares it to the statistics of all the other bricks, looking for highly deviant rates. Because we want to be able to run SSM on a relatively small number of nodes, we calculate the median absolute deviation of the activity statistics. This metric is robust to outliers even in small populations, and lets us identify deviant activity statistics with a low-false positive rate.

Statistic	Description	Type
NumElements	Number of objects stored	State
MemoryUsed	Total memory used	State
InboxSize	Size of Inbox in last second	State
NumDropped	Dropped request in last second	Activity
NumReads	Reads processed in last second	Activity
NumWrites	Writes processed in last second	Activity

Table 3: Statistics reported to Pinpoint by Brick

State statistics represent the size of some state, such as the size of the message inbox. In SSM, these state statistics often vary in periodic patterns, e.g., in normal behavior, the MemoryUsed statistic grows until the garbage collector is triggered to free memory, and the pattern repeats. Unfortunately, we do not know *a priori* the period of this pattern – in fact, we cannot even assume a regular period.

To discover the patterns in the behavior of state statistics, we use the Tarzan algorithm for analyzing time series [25]. For each state statistic of a brick, we keep an N-length history or time-series of the state. We discretize this time-series into a binary string. To discover anomalies, Tarzan counts the relative frequencies of all substrings shorter than k within these binary strings. If a brick's discretized time-series has a surprisingly high or low frequency of some substring as compared to the other brick's time series, we mark the brick as potentially faulty. This algorithm can be implemented in linear time and linear space, though we have found we get sufficient performance from our simpler nonlinear implementation.

Once a brick has been identified as potentially faulty through three or more activity and state statistics, we conclude that the brick has indeed failed in some way; Pinpoint restarts the node. In the current implementation, a script is executed to restart the appropriate brick, though a more robust implementation might make use of hardware-based leases that forcibly reboot the machine when they expire [11].

Because restarting a brick will only cure transient failures, if Pinpoint detects that a brick has been restarted more than a threshold number of times in a given period, which is usually indicative of a persistent fault, it can take the brick off-line and notify an administrator.

5. EXPERIMENTAL RESULTS

In this section, we highlight some of the key features from the design of SSM. We present benchmarks illustrating each of the recovery-friendly, self-tuning, and self-protecting features of SSM. We also present numerous benchmarks demonstrating the self-healing nature of SSM when integrated with Pinpoint.

Each benchmark is conducted on the UC Berkeley Millennium Cluster. Our load generator models hyperactive users who continually make requests to read and write session state; each hyperactive user is modeled using a thread which does a sequence of alternating write and read requests, which is representative of the workload for session state,

as described earlier in Section 2. As soon as a request returns, the thread immediately makes a subsequent request. In the following benchmarks, we vary the number of sending threads as well as the number of bricks. All bricks are run on separate, dedicated machines.

5.1 Recovery-Friendly

In a sufficiently-provisioned, non-overloaded system, the failure and recovery of a single brick does not affect:

- Correctness. As described above, the failure of a single brick does not result in data loss. In particular, SSM can tolerate $WQ - 1$ simultaneous brick failures before losing data.

  A restart of the brick does not impact correctness of the system.

- Performance. So long as W is chosen to be greater than WQ and R is chosen to be greater than 1, any given request from a stub is not dependent on a particular brick. SSM harnesses redundancy to remove coupling of individual requests to particular bricks.

  A restart of the brick does not impact performance; there is no special case recovery code that must be run anywhere in the system.

- Throughput. A failure of any individual brick does not degrade system throughput in a non-overloaded system. Upon first inspection, it would appear that all systems should have this property. However, systems that employ a buddy system or a chained clustering system [19, 26] fail to balance the resulting load evenly. Consider a system of four nodes A, B, C, and D, where A and B are buddies, and C and D are buddies. If each node services load at 60 percent of its capacity and subsequently, node D fails, then its buddy node C must attempt to service 120 percent of the load, which is not possible. Hence the overall system throughput is reduced, even though the remaining three nodes are capable of servicing an extra 20 percent each.

  In SSM, because the resulting load is distributed evenly between the remaining bricks, SSM can continue to handle the same level of throughput so long as the aggregate throughput from the workload is lower than the aggregate throughput of the remaining machines.

  The introduction of a new brick or a revived brick never decreases throughput; it can only increase throughput, since the new brick adds new capacity to the system. A newly restarted brick, like every other brick, has no dependencies on any other node in the system.

- Availability. In SSM, all data is available for reading and writing during both brick failure and brick recovery. In other systems such as unreplicated file systems, data is unavailable for reading or writing during failure. In DDS [19] and in Harp [29], data is available for reading and writing after a node failure, but data is

not available for writing during recovery because data is locked and is copied to its buddy en masse.

This benchmark is intended to show the recovery-friendly aspect of SSM. In this benchmark, W is set to 3, WQ is set to 2, *timeout* is set to 60 ms, R is set to 2, and the size of state written is 8K.

We run four bricks in the experiment, each on a different physical machine in the cluster. We use a single machine as the load generator, with ten worker threads generating requests at a rate of approximately 450 requests per second.
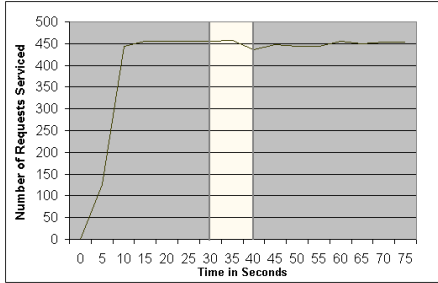
Figure 5: **SSM running with 4 Bricks. One brick is killed manually at time 30, and restarted at time 40. Throughput and availability are unaffected. Although not displayed in the graph, all requests are all fulfilled correctly, within the specified timeout.**

We induce a fault at time 30 by killing a brick by hand. As can be shown from the graph, throughput remains unaffected. Furthermore, all requests complete successfully; the load generator showed no failures. This microbenchmark is intended to demonstrate the recovery-friendly aspect of SSM. In a non-overloaded system, the failure and recovery of a brick has no negative effect on correctness, system throughput, availability, or performance. All generated requests completed within the specified timeout, and all requests returned successfully.

5.2 Self-Tuning

The use of AIMD allows the stubs to adaptively discover the capacity of the system, without requiring an administrator to configure a system and a workload, and then run experiments to determine whether the system services the workload in an acceptable fashion. In the manual process, if the workload increases drastically, the configuration may need to be changed.

In SSM, the allowable amount of time for session state retrieval and storage is specified in a configurable timeout value. The system tunes itself using the AIMD mechanism, maximizing the number of requests that can be serviced within that time bound. SSM automatically adapts to higher load gracefully. Under overload, requests are rejected instead of allowing latency to increase beyond a reasonable threshold. If SSM is is deployed in an environment with a pool of free machines, Pinpoint can monitor the number of requests that are rejected, and start up new bricks to accomodate the increase in workload.

This benchmark is intended to show that SSM discovers the maximum throughput of the system correctly. Recall that in SSM, read and write requests are expected to complete within a timeout. We define *goodput* as the number of requests that complete within the specified timeout. Offered load is goodput plus all requests that fail. Requests that complete after that timeout are not counted toward goodput. In this benchmark, W is set to 3, WQ is set to 2, timeout is set to 60 ms, R is set to 1, and the size of state written is 8K. We use 3 bricks.

First, we discover the maximum goodput of the basic system with no admission control or AIMD. We do so by varying the number of sending threads to see where goodput plateaus. We run separate experiments; first we generate a load with 120 threads corresponding to roughly 1900 requests per second, and then with 150 threads, corresponding to roughly 2100 requests per second. Figure 6 shows that goodput plateaus around 1900-2000 requests per second.
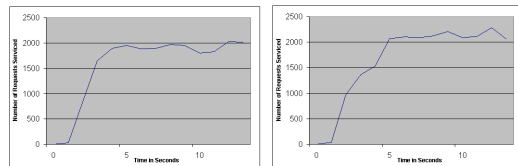

Figure 6: **SSM running with 3 Bricks, no AIMD or admission control. The graph on the left shows a load of 120 threads sending read and write requests of session state. The graph on the right shows a load of 150 threads. sending threads. System throughput peaks at around 1900-2000 requests per second.**

We continue increasing the load until goodput drops to zero. Goodput eventually drops to zero because the rate of incoming requests is higher than the rate at which the bricks can process, and eventually, the brick spends all of its time fulfilling timed-out requests instead of doing useful work. As can be seen in the lightened portion of Figure 7, the bricks collapse under the load of 220 threads, or about 3400 requests a second; requests arrive at a rate faster than can be serviced, and hence the system goodput falls to zero at time 11.
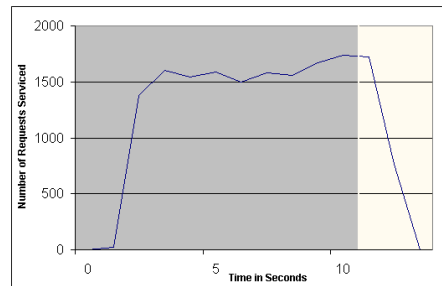

Figure 7: **SSM running with 3 Bricks, no AIMD or admission control. The bricks collapse at time 11 under the load of 220 threads generating requests.**

After manually verifying the maximum goodput of the system, we turn on the self-protecting features, namely by allowing the stub to use the AIMD sending window size and by forcing bricks to service only requests that have not timed

out, and run the experiment again.

We generate an even higher load than what caused goodput to fall to zero in the basic system, using 240 threads, corresponding to roughly 4000 requests per second. As seen in figure 8, SSM discovers the maximum goodput and the system continues to operate at that level. Note that this means that the system is rejecting the excess requests, since the bricks are already at capacity, and the excess load is simply being rejected; the percentage of rejected requests is discussed in the next section. We sketch a simple and reasonable shedding policy in future work.
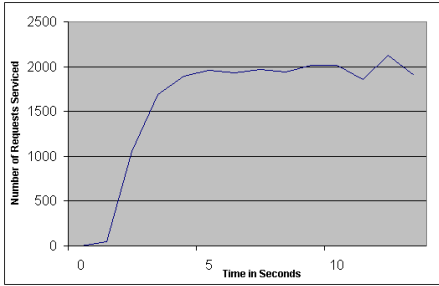

Figure 8: **SSM running with 3 Bricks, with AIMD and admission control. SSM discovers maximum goodput of around 2000.**

5.3 Self-Protecting

SSM protects its components from collapsing under overload. The use of AIMD and admission control allow SSM to protect itself. In particular, the maximum allowable pending non-acked requests that a stub can generate for a particular brick is regulated by the sending window size, which is additively increased on success and multiplicatively decreased on failure. This prevents the stubs from generating load that results in brick overload; each stub exerts backpressure [43] on its caller when the system is overloaded. In addition, bricks actively discard requests that have already timedout, in order to service only requests that have the potential of doing useful work.

This benchmark is intended to show that SSM protects itself under overload. Part of the self-protecting aspect of SSM is demonstrated in the previous benchmark; SSM's goodput does not drop to zero under heavier load. In this benchmark, W is set to 3, WQ is set to 2, timeout is set to 60 ms, R is set to 1, and the size of state written is 8K.

SSM's use of the self-protecting features allows SSM to maintain a reasonable level of goodput under excess load. Figure 9 shows the steady state graph of load vs. goodput in the basic system without the self-protecting features. Figure 10 shows the steady state graph of load vs. goodput in SSM with the self-protecting features enabled. The x-axis on both graphs represents the number of load-generating machines; each machine runs 12 threads. The y-axis represents the number of requests. We start with the load generator running on a single machine, and monitor the goodput of SSM after it has reached steady state. Steady state is usually reached in the first few seconds, but we run each configuration for 2 minutes to verify that steady state be-

havior remains the same. We then repeat the experiment by increasing the number of machines used for load generation.

Comparison of the two graphs shows two important points:

- The self-protecting features protect the system from overload and falling off the cliff and allows the system to continue to do useful work.

- Extends useful life of the system under overload. Without the self-protecting features, we see that maximum goodput is around 1900 requests per second, while goodput drops to half of that at a load of 13 machines, and falls to zero at 14 machines. With the self-protecting features, maximum goodput remains the same, while goodput drops to half of the maximum at 24 machines, and goodput trends to zero at 37 machines, because SSM begins spending the bulk of its processing time trying to protect itself and turning away requests and is unable to service any requests successfully. With self-protecting features turned on, the system continues to produce half of goodput at 24 machines vs. 13 machines, protecting the system from almost double the load.

Note that in Figure 9 where goodput has dropped to zero, as we increase the number of machines generating load that the offer load increases only slightly, staying around 1500 failed requests per second. This is because each request must wait the full timeout value before returning to the user; the requests that are generated will arrive at the bricks, but will not be serviced in time. However, in Figure 10, the number of failed requests increases dramatically as we increase the number of machines. Recall that the load generator models hyperactive users that continually send read and write requests; each user is modeled by a thread. When one request returns, either successfully or unsuccessfully, the thread immediately generates another request. Because SSM is self-protecting, the stubs say "no" to requests right away; under overload, requests are rejected immediately. The nature of the load generator then causes another request to be generated, which is likely to be rejected as well. Hence the load generator continues to generate requests at a much higher rate than in Figure 9 because unfulfillable requests are immediately rejected.

5.4 Self-Healing

The ability of a system to heal itself without requiring administrator assistance greatly simplifies management. However, accurate detection of faults is difficult. Usually, acting on an incorrect diagnosis such as a false positive results in degraded system performance, availability, correctness, or throughput. In SSM, the ability to reboot any component without affecting correctness and availablility, and to a degree, performance and throughput, coupled with a generic fault-detecting mechanism such as Pinpoint, gives rise to a self-healing system.

As discussed earlier, in SSM, a single brick can be restarted without affecting correctness, performance, availability, or throughput; the cost of acting on a false positive
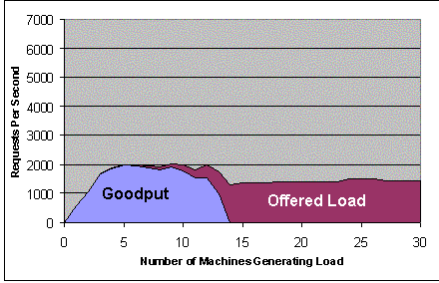
Figure 9: **Steady state graph of load vs. goodput. SSM running without self-protecting features. Goodput peaks at around 1900 requests per second. Half of system goodput is reached at 13 load generating machines, and system goodput drops to 0 at 14 machines.**
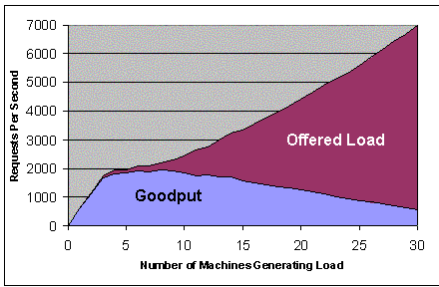

Figure 10: **Steady state graph of load vs. goodput. SSM running with self-protecting features. Goodput peaks at around 1900 requests per second. Half of system goodput is reached at 24 load generating machines, and system goodput trends to 0 at 37 machines.**

on SSM is very low, so long as the system does not make false positive errors with too high a frequency. For transient faults, Pinpoint can detect anomalies in brick performance, and restart bricks accordingly.

The following microbenchmarks demonstrate SSM's ability to recover and heal from transient faults. We attempt to inject realistic faults for each of SSM's hardware components—processor, memory, and network interface. We assume that for CPU faults, the brick will hang or reboot, as is typical for most such faults [20].

To model transient memory errors, we inject bitflip errors into various places in the brick's address space. To model a faulty network interface, we use FAUMachine [10], a Linux-based VM that allows for fault-injection at the network level, to drop a specified percentage of packets. We also model performance faults, where one brick runs more slowly than the others. In all of the following experiments, we use six bricks; Pinpoint actively monitors all of the bricks.

Obviously, restarting a brick does not cure a persistent fault. Although we model only transient faults, Pinpoint can keep track of the number of times a particular brick has been rebooted in a certain time interval, and shut the brick down or notify an administrator completely if it exceeds a threshold number of restarts.

5.5 Memory Fault in Stack Pointer

This benchmark is intended to show the self-healing nature of SSM under the presence of a memory fault; performance and throughput is unaffected, and SSM recovers from the fault.

Using ptrace(), we monitor a child process and change its memory contents. In this benchmark, $W = 3, WQ = 2, R = 2$, data size is 8KB, and we increase t to 100ms to account for the slowdown of bricks using ptrace. Figure 11 shows the results of injecting a bitflip in the area of physical memory where the stack pointer is held. The fault is injected at time 14; the brick crashes immediately. The lightened section of figure 11 (time 14-23) is the time during which only five bricks are running. At time 23, Pinpoint detects that the brick has stopped sending heartbeats and should be restarted, and restarts the brick; the system tolerates the fault and successfully recovers from it.
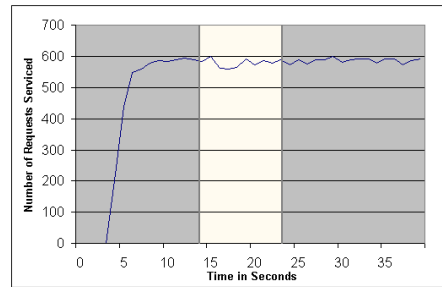

Figure 11: **Fault Injection: Memory Bitflip in Stack Pointer.**

5.6 Memory Fault in Data Value/Checksum

This benchmark is intended to show the self-healing nature of SSM under the presence of a memory fault in its internal data structures. Figure 12 shows the injection of a bitflip in the object of a session state object that has just been written and is about to be read. The fault is injected at time 18. The brick is configured to exit upon the detection of a checksum error, and does so immediately at time 18. The lightened section of figure 12 (time 18-29) is the time during which only five bricks are running. At time 29, Pinpoint detects that the brick has stopped sending heartbeats and should be restarted, and restarts the brick; the system tolerates the fault and successfully recovers from it.
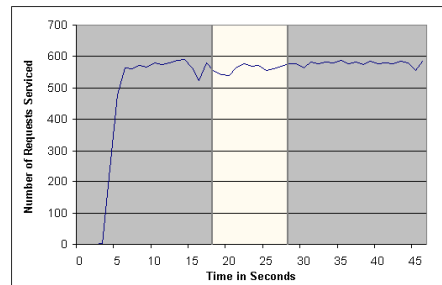

Figure 12: **Fault Injection: Memory Bitflip in hashtable**

5.7 Network Performance Fault

This benchmark is intended to show that SSM can tolerate and recover from transient network faults. We use FAUMachine to inject a fault at the brick's network interface. In particular, we cause the brick's network interface to drop 70 percent of all outgoing packets. Figure 13 shows this experiment running on FAUmachine. Note that FAUmachine overhead causes the system to perform an order of magnitude slower; we run all six bricks on FAUMachine. In this benchmark, $W = 3, WQ = 2, R = 2$, and we increase t to 700ms and decrease the size of the state written to 3KB to adjust to the order of magnitude slowdown.
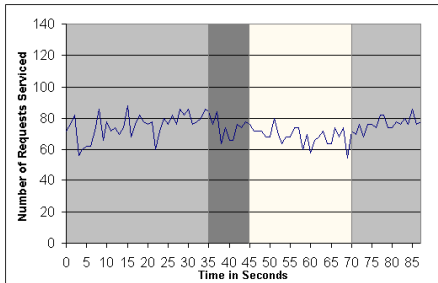

Figure 13: **Fault Injection: Dropping 70 percent of outgoing packets. Fault injected at time 35, brick killed at time 45, brick restarted at time 70.**

The fault is injected at time 35; however, the brick continues to run with the injected fault for 10 seconds, as shown in the darkened portion of figure 13. At time 45, Pinpoint detects and kills the faulty brick. The fault is cleared to allow network traffic to resume as normal, and the brick is restarted. Restart takes significantly longer using the FAUMachine, and the brick completes its restart at time 70. During the entire experiment, all requests complete correctly in the specified timeout and data is available at all times. Throughput is affected slightly, as expected, as only five bricks are functioning during times 45-70; recall that running bricks on FAUMachine causes an order of magnitude slowdown.

5.8 CPU/Memory Performance Fault

SSM is able to tolerate performance faults, and Pinpoint is able to detect performance faults and reboot bricks accordingly. In the following benchmark with 6 bricks and 3 load-generating machines, we inject performance failures in a single brick by causing the brick to sleep for 1ms before handling each message. This per-message performance failure simulates software aging. In figure 14, we inject the fault every 60 seconds. Each time the fault is injected, Pinpoint detects the fault within 5-10 seconds, and reboots the brick. All requests are serviced properly.

6. END TO END APPLICATION BENCHMARKS

In this section, we integrate SSM with a production, enterprise-scale application. We also modify the application to use disk to store session state, as a baseline comparison.
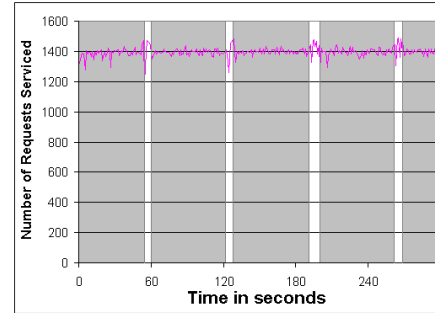

Figure 14: **Performance Fault: Brick adds 1ms sleep before each request; faults injected every 60 seconds, Pinpoint detects failure within 5-10 seconds, and brick is restarted.**

We compare the integrated solution with the unmodified application, as well as the application modified to use disk to store session state.

The application we use is a simplified version of Tellme's [31] Email-By-Phone application; via the phone, users are able to retrieve their email and listen to the headers of various folders by interacting with voice-recognition and voice-synthesis systems integrated with the email server. The application logic itself is written in Java and run on Resin [37], an XML application server on top of a dual processor Pentium III 700 MHz machine with 1G RAM, running Solaris. We use 3 bricks for the benchmark. All machines are connected via switched ethernet, and held in a commercial hosting center.

Session state in this application consists of the index of the message the user is currently accessing, the name of the folder that is currently being accessed, and other workflow information about the user's folders. In the original application, the session state is stored in memory only; a crash of the application server implies a loss of all user sessions, and a visible application failure to all active users.

We use Silk Performer [39] to generate load to the application. The load generator simulates users that start the application, listen to an email for three seconds, and progress to the next email, listening to a total of 20 emails. The test is intended to establish a baseline for response time and throughput for the unmodified application. We vary the load from ten users to 160 users; at 170 users, the application server begins throwing "Server too busy" errors. Unmodified, the application server can handle 160 simultaneous users, average a response time of 0.793 seconds.

We modify the application to write session state to disk, to establish comparison values for an external persistent state store. We simply write out the user's session state to a file on disk. The modified application reaches capacity at 120 simultaneous users, with an average response time of 1.716 seconds.

Lastly, we integrate SSM with the application to store state. Three other machines are configured as bricks. A switch sits between the three bricks and the application server. The integrated application reaches capacity at 120 simultaneous users, with an average response time of 0.863
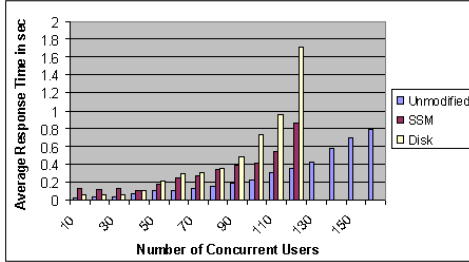
Figure 15: **Latency vs. load for 10 to 160 users. The original application can handle a capacity of 160 simultaneous users. The modified application using disk or using SSM can each handle 120 users.**

seconds.

Figure 15 summarizes the results. Compared to storing session state in-memory-only, using our prototype of SSM imposes a 25 percent throughput penalty on the overall application: the maximum number of simultaneous users is reduced from 160 to 120, although the per-user response times are roughly equal in the two cases, so users perceive no latency penalty.

Compared to using a filesystem, SSM supports just as many concurrent users, but delivers better response time: with 120 active users, the application using disk runs more than twice as slowly as the application using SSM.

In summary, integrating SSM with the application imposes a 25 percent throughput overhead compared to in-memory-only, but preserves throughput and delivers better response time than the disk solution. Neither the in-memory nor the filesystem solution provide SSM's high availability, self-recovery and self-healing.

7. DISCUSSION

SSM bricks can be built from simple commodity hardware. From a few months experience working with SSM, bricks perform very predictably, which in turn allows detection of anomalies to be extremely simple and accurate. In this section we try to distill what properties of SSM's design and algorithms give rise to these properties.

7.1 Eliminate Coupling

In SSM, we have attempted to eliminate all coupling between nodes. Bricks are independent of other bricks, which are independent of stubs. Stubs are independent of all other stubs; each stub is regulated by an AIMD sending window which prevents it from saturating the system.

In traditional storage systems, a requestor is coupled to a requestee, and the requestor's performance, correctness, and availability are all dependent on the requestee. SSM instead uses single-phase, non-locking operations, allowing writes to proceed at the speed of the fastest WQ bricks instead of being coupled to lagging or failing bricks. Among other things, this makes lengthy garbage collection times unimportant, since a brick performing GC can temporarily fall behind without dragging down the others.

Elimination of coupling comes at a cost: redundancy is

harnessed for performance. Redundant components with reduced coupling gives rise to predictable performance. Coupling elimination has been used in various contexts [21, 13, 5].

Related to coupling is the use of both randomness to avoid deterministic worst cases and overprovisioning to allow for failover. Both techniques are used in large-system load balancing [4], but SSM does this at a finer grain, in the selection of the write set for *each request*.

In the initial implementation of SSM, use of third-party libraries led us to violate our own guideline. We attempted to use a commercially-available JMS (Java Messaging Service) implementation that used a single machine as a message router, quickly becoming a scaling bottleneck. We ultimately discarded JMS in favor of using TCP directly.

7.2 Make Parts Interchangeable

For a write in SSM, any given brick is as good as any other in terms of correctness. For a read, a candidate set of size R is provided, and any brick in the candidate set can function in the place of any other brick. The introduction of a new brick does not adversely disrupt the existing order; it only serves to increase availability, performance, and throughput.

In many systems, certain nodes are required to fulfill certain fixed functions. This inflexibility often causes performance, throughput or availability issues, as evidenced in DDS [19], which uses the buddy system.

In SSM, bricks are all equivalent. Because complete copies of data are written to multiple bricks, bricks can operate independently, and do not require any sort of coordination. Furthermore, as long as one brick from the write quota of size WQ remains, the data is available, unlike in erasure coding systems such as Palimpsest [38], where a certain number of chunks is required to reconstruct data.

7.3 It's OK to Say No

SSM uses both adaptive admission control and early rejection as forms of backpressure. AIMD is used to regulate the maximum number of requests a stub can send to a particular brick; each brick can reject or ignore timed-out requests; the application of TCP, a well-studied and stable networking protocol, allows SSM components to reach capacity without collapse [43]. The goal of these mechanisms is to avoid having SSM attempt to give a functionality guarantee ("I will do it") at the expense of predictable performance; the result is that each request requires a predictable amount of work.

Once again, we inadvertently violated our own principle at a lower layer. Recall that each stub maintains a list of bricks it believes to be alive. An `ObjectOutputStream` is used to communicate with each such brick. In the initial implementation of the stubs, when a network fault was injected at a brick, the call to `ObjectOutputStream::WriteObject()` would block until the write succeeded or an error was thrown. Since this blocked thread holds the mutex for the stream, all other threads attempting to write to the stream are blocked as well; eventually, all threads become blocked. The stub implementation was subsequently modified so that the calling thread puts the read or write request in an outbox, and a separate thread handles the actual write; the outbox then

becomes another backpressure point.

7.4 It's OK to Make Mistakes

The result of the application of redundancy and the interchangeability of components is that recovery is fast, simple, and unintrusive: a brick is recovered by rebooting it without worrying about preserving its pre-crash state, and recovery does not require coordination with other bricks.

As a result, the monitoring system that detects failures is allowed to make mistakes, as long as they are not made too often. In contrast, in other systems, false positives usually reduce performance, lower throughput, or cause incorrect behavior. Since false positives are not a problem in SSM, generic methods such as statistical anomaly based failure detection can be made quite aggressive, to avoid missing real faults.

8. RELATED WORK

Palimpsest [38] describes a scheme for temporal storage for planetary-scale services. Palimpsest requires a user to erasure-code the relevant data, and write it to N replica sites, which may all be under separate administrative control. Like SSM, all metadata for the write is stored on the client. However, Palimpsest is intended for the wide area network, and storage sites may be under different administrative domains. Palimpsest gives no guarantees to its users in terms of storage lifetime; SSM gives probabilistic guarantees that data will be available within some fixed time period.

Several projects have focused on the design and management of persistent state stores [1, 19, 30, 2, 14, 21]. FAB [14] shares many of the same motivations as SSM, including ease of management and recovery; however, FAB is intended at a very different level in the storage hierarchy. FAB is a logical disk system for persistent storage that is intended to replace enterprise-class disk arrays, while in SSM we focus on temporal storage of session state. In addition, in SSM, all metadata is stored at the client, while FAB employs a majority-voting algorithm. Similarly, DStore [21] shares many of the motivations as SSM, but it focuses on unbounded-persistence storage for non-transactional, single-key-index state.

Petal's [26] storage nodes form a logical chain. Petal attempts to make data highly available by placing data on a particular node, and placing a backup copy on either the predecessor or the successor node. Upon failure, the load is divided by the predecessor and successor, whereas in SSM the load redistribution is more even across all nodes. Furthermore, the loss of any two adjacent nodes implies data loss, while in SSM, the number of replicated is configurable.

SSM's algorithm is different from that of quorums [16]. In quorum systems, writes must be propagated to W of the nodes in a replica group, and reads must be successful on R of the nodes, where $R + W > N$, the total number of nodes in a replica group. A faulty node will often cause reads to be slow, writes to be slow, or possibly both. Our solution obviates the need for such a system, since the cookie contains the references to up-to-date copies of the data; quorum systems are used to compare versions of the data to determine which copy is the current copy.

SSM owes many of its motivations to DDS [19]. DDS is similar to SSM in that it focuses on state accessible by single-key lookups. A detailed discussion of the differences between SSM and DDS can be found in previous work [27, 28].

We share many of the same motivations as Berkeley DB [32], which stressed the importance of fast-restart and treating failure as a normal operating condition, and recognized that the full generality of databases is sometimes unneeded.

The windowing mechanism used by the stubs is motivated by the TCP algorithm for congestion control [24]. The need to include explicit support for admission control and overload management at service design time was demonstrated in SEDA [43]; we appeal to this argument in our use of windowing to discover the system's steady-state capacity and our use of "backpressure" to do admission control to prevent driving the system over the saturation cliff.

Zwaenepoel et al [12] are also looking at using generic, low-level statistical metrics to infer high-level application behaviors. In their case, they are looking at CPU counters such as number of instructions retired and number of cache misses to make inferences about the macro-level behavior of the running application.

9. FUTURE WORK

SSM currently does not tolerate catastrophic site failures, but can be extended to do so. When selecting bricks for writes, SSM can be extended to select W_{local} bricks from the local network, and W_{remote} bricks from a remote site. SSM can return from writes when WQ_{local} bricks have replied, and 1 remote brick has replied.

Intelligently shedding load is an area of active research. One policy is to allow only users that are already actively using the system to continue using the system, and to turn new sessions away; this can be done by only allowing writes by users that have valid cookies when the system is overloaded. Alternatively, users can be binned into different classes in some external fashion, and under overload, SSM can be configured to service only selected classes.

We are exploring the use of rolling reboots as a method of proactively avoiding failures.

Currently, Pinpoint monitors statisics that empirically correlate with injected failures; however, we have no proof that they are the most relevant ones. An avenue of future research is to apply statistical learning theory to automatically determine which measurable features best correlate with failures.

10. CONCLUSIONS

A "new wave" of systems research is focusing on the dual synergistic areas of reduced total-cost-of-ownership and managed/hosted online services. Many groups have proposed visions for self-managing, self-adapting, or self-healing systems; we have presented an architecture and implemented prototype that realizes at least some of those behaviors in a state-storage subsystem for online services.

We have also attempted to illuminate one approach to self-management in the context of this work: make recovery so fast and inexpensive that false positives during failure detection become less important, thereby allowing the use of powerful, self-adapting, application-generic failure detection techniques such as statistical-anomaly analysis. We hope that our prototype will both prove useful as a building block for future online services and encourage others working on self-managing systems to explore similar recovery-friendly designs.

Acknowledgments

11. REFERENCES

[1] Anurag Acharya. Reliability on the cheap: How I learned to stop worrying and love cheap PCs. In *2nd Workshop on Evaluating and Architecting System Dependability*, San Jose, CA, Oct 2002. Invited Talk.

[2] Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. Hippodrome: Running circles around storage administration. In *Proc. Conference on File and Storage Technologies (FAST-02)*, pages 175–188, Monterey, CA, 2002.

[3] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Fail-stutter fault tolerance. In *Proc. 8th Workshop on Hot Topics in Operating Systems*, pages 33–38, Elmau/Oberbayern, Germany, May 2001.

[4] Eric Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, July 2001.

[5] Eric Brewer. Running Inktomi. Personal Communication, 2001.

[6] Aaron B. Brown and David A. Patterson. To err is human. In *Proceedings of the 1st Workshop on Evaluating and Architecting System Dependability (EASY)*, Göteborg, Sweden, July 2001. IEEE Computer Society.

[7] George Candea and Armando Fox. Crash-only software. In *Proc. 9th Workshop on Hot Topics in Operating Systems*, Lihue, HI, June 2003.

[8] William Clinger and Lars Hansen. Generational garbage collection and the radioactive decay model. *Proc. SIGPLAN 97 Conference on Programming Language Design and Implementation*, May 1997.

[9] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Proceedings of the ACM SIGCOMM Conference*, pages 3–12, 1989.

[10] FAUMachine. `http://www.FAUmachine.org/`.

[11] Christof Fetzer. Perfect failure detection in timed asynchronous systems. *IEEE Transactions on Computers*, 52(2):99–112, February 2003.

[12] Robert Folwer, Sameh Elnikety Alan Cox, and Willy Zwaenepoel. Using performance reflection in systems software. In *Proc. 9th Workshop on Hot Topics in Operating Systems*, Lihue, HI, June 2003.

[13] Armando Fox and Eric A. Brewer. ACID confronts its discontents: Harvest, yield, and scalable tolerant systems. In *Seventh Workshop on Hot Topics In Operating Systems (HotOS-VII)*, Rio Rico, AZ, March 1999.

[14] S. Frolund, A. Merchant, Y. Saito, S. Spence, and A. Veitch. FAB: Enterprise storage systems on a shoestring. In *Proc. 9th Workshop on Hot Topics in Operating Systems*, Lihue, HI, June 2003.

[15] Sachin Garg, Aard Van Moorsel, K. Vaidyanathan, and Kishor S. Trivedi. A methodology for detection and estimation of software aging. In *Proceedings of the 9th International Symposium on Software Reliability Engineering*, pages 282–292, Paderborn, Germany, Nov 1998.

[16] David K. Gifford. Weighted voting for replicated data. In *Proc. 7th ACM Symposium on Operating Systems Principles*, Pacific Grove, CA, 1979.

[17] Cary G. Gray and David R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proc. 12th ACM Symposium on Operating Systems Principles*, pages 202–210, Litchfield Park, AZ, 1989.

[18] Jim Gray. The transaction concept: Virtues and limitations. In *Proceedings of VLDB*, Cannes, France, September 1981.

[19] Steven D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction. In *Proc. 4th USENIX Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000.

[20] Weining Gu, Zbigniew Kalbarczyk, Ravi Iyer, and Zhenyu Yang. Characterization of linux kernel behavior under errors. In *Proc. International Conference on Dependable Systems and Networks*, San Francisco, CA, June 2003.

[21] Andrew C. Huang and Armando Fox. Decoupling state stores for ease of management. Submitted to FAST, 2004.

[22] Dean Jacobs. Personal Communication, 2003.

[23] Dean Jacobs. Distributed computing with BEA WebLogic server. In *Proceedings of the Conference on Innovative Data Systems Research*, Asilomar, CA, January 2003.

[24] V. Jacobson. Congestion avoidance and control. In *Proceedings of the ACM SIGCOMM Conference*, Stanford, CA, August 1988.

[25] E. Keogh, S. Lonardi, and W Chiu. Finding surprising patterns in a time series database in linear time and space. In *In proc. of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 550–556, Edmonton, Alberta, Canada, Jul 2002.

[26] Edward K. Lee and Chandramohan Thekkath. Petal: Distributed virtual disks. In *Proc. 7th International*

Conference on Architectural Support for Programming Languages and Operating Systems, pages 84–92, Cambridge, MA, 1996.

[27] Ben Ling and Armando Fox. The case for a session state storage layer. In *Proc. 9th Workshop on Hot Topics in Operating Systems*, Lihue, HI, June 2003.

[28] Benjamin C. Ling and Armando Fox. A self-tuning, self-protecting, self-healing session state management layer. In *5th Annual Workshop On Active Middleware Services*, Seattle, WA, 2003.

[29] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp file system. In *Proc. 13th ACM Symposium on Operating Systems Principles*, pages 226–238, Pacific Grove, CA, Oct 1991.

[30] Network Appliance. `http://www.netapp.com/`.

[31] TellMe Networks.

[32] M. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proceedings of the 1999 Summer USENIX Technical Conference*, Monterey, CA, June 1999.

[33] Anil Pal. Yahoo! User Preferences Database. Personal Communication, 2003.

[34] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, and N. Treuhaft. Recovery oriented computing (ROC): Motivation, definition, techniques, and case studies. Technical Report UCB/CSD-02-1175, UC Berkeley, Berkeley, CA, March 2002.

[35] Ji Ping, Zihui Ge, Jim Kurose, and Don Towsley. A comparison of hard-state and soft-state protocols. In *Proceedings of the ACM SIGCOMM Conference*, pages 251–262, Karlsruhe, Germany, Aug 2003.

[36] S. Raman and S. McCanne. A model, analysis, and protocol framework for soft state-based communication. In *Proceedings of the ACM SIGCOMM Conference*, Cambridge, MA, September 1999.

[37] Resin. `http://www.caucho.com/`.

[38] Timothy Roscoe and Steven Hand. Palimpsest: Soft-capacity storage for planetary-scale services. In *Proc. 9th Workshop on Hot Topics in Operating Systems*, Lihue, HI, June 2003.

[39] Silk Performer. `http://www.segue.com/`.

[40] Ujjwal Singh. E.piphany. Personal Communication, 2003.

[41] Sun Microsystems. J2EE platform specification. `http://java.sun.com/j2ee/`, 2002.

[42] Allan Vermeulen. Personal communication, June 2003.

[43] Matt Welsh, David Culler, and Eric Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proc. 18th ACM Symposium on Operating Systems Principles*, pages 230–243, Lake Louise - Banff, Canada, 2001.