A Self-Tuning, Self-Protecting, Self-Healing Session State Management Layer

Benjamin C. Ling and Armando Fox
Stanford University
{bling, fox}@cs.stanford.edu

Abstract

Management of semi-persistent state, such as user-session state, is one factor that complicates failure management in clustered three-tier Internet applications [5]. We observe that the specific properties of user-session state can be exploited to design a lightweight state storage layer that offers many of the same ease-of-management and ease-of-recovery properties as stateless components such as Web servers. We describe SSM, a self-tuning, self-protecting, and self-healing session state management layer that provides a storage and retrieval mechanism for semi-persistent, serial-access user session state. SSM is fast, scalable, fault-tolerant, and recovers instantly from individual node failures. Any SSM node may be rebooted at any time and there is no special recovery code, so the performance cost of "eager" recovery is near zero, simplifying recovery policy management when SSM is integrated into a larger system.

1. Introduction

The concept of a user session is present in nearly all client-facing applications, including web-based applications. A user interacts with the application for a period of time, called a session, until she signs out or her session expires after a fixed time interval. During this time, the user's interaction with the application may produce temporary data relevant to the user's session, e.g. which step the user has completed in the application workflow. Upon completion of the session, this data is no longer needed.

Such web-based applications are typically complex, constructed in a multi-tier arrangement [5] for separation of concerns and scalability. A major challenge of such application installations is keeping them highly available—even well-run services achieve only two to three nines (99% to 99.9%) availability [23]. Redundancy and fast failover have traditionally been used to mask failures (and thereby speed recovery) for stateless application components such as Web server front-ends. We ask whether the specific properties of session state can be exploited to construct a stateful component (the session state store) that presents the same availability and ease-of-management opportunities as stateless building blocks.

In particular, we present a self-tuning, self-protecting, and self-healing session state management layer that provides a storage and retrieval mechanism for semi-persistent serial-access user session state. SSM is fast, scalable, fault-tolerant, and optimized for recovery speed and ease of recovery management. By recovery speed, we mean that fast failover recovers from individual SSM node failures, and no data is lost and all data can continue to be read and written during the recovery interval. By ease of recovery management, we mean that deciding the correct recovery *policy* for a larger system that includes SSM is easy: SSM nodes can be recovered simply by restarting them at any time, and since the performance cost of recovery is near zero, there is no penalty for accidentally "over-recovering". This property makes it easier to use techniques such as fault model enforcement [24] that may use overly-conservative recovery strategies to assure recovery will succeed. By self-tuning, we mean that SSM discovers its own maximum load capacity, without requiring a system administrator to specifically state the load capacity of each component or to conduct staging experiments to discover its aggregate capacity. By self-protecting, we mean that SSM uses admission control to protect itself from collapsing from overload. Finally, by self-healing, we mean that SSM is able to continue functioning correctly in the presence of non-properly functioning modules, either in degraded performance, incorrect results, livelocks, or crashes. SSM should be able to 1) detect when any of the above conditions occur, and 2) take appropriate actions, either by restarting the faulty module, shutting it down permanently, or simply continue functioning with the faulty module. SSM recovers instantly from individual node failures; data is replicated, and no component in SSM stores hard state or requires hard state to operate, and therefore, recovery is simple and usually requires at most a restart of the process. Furthermore, by

PROCEEDINGS OF THE AUTONOMIC COMPUTING WORKSHOP FIFTH ANNUAL INTERNATIONAL WORKSHOP ON ACTIVE MIDDLEWARE SERVICES (AMS'03) ISBN 0-7695-1983-0/01 $17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY

increasing the number of redundant copies of data that is stored, SSM is able to proactively reboot components to recover from performance failures at individual nodes.

In Section 2 we discuss the particular class of session state that SSM addresses, the qualities of the state, and how we can exploit its distinct properties to build a simple yet efficient state store. In Section 3, we discuss existing solutions and why they are inadequate. In Section 4, we discuss the design and implementation of SSM. In Section 5, we evaluate the system and its implementation. In Section 6, we discuss future work. In Section 7, we present related work, and then conclude.

2. Exploiting Specific Properties of Session State

We describe a large subcategory of session state by describing how it is used, its properties, the requirements it imposes on its storage, as well as the functionality required to support it. Various different categories of session state exist. However, in the remainder of this paper, we will use the term "session state" to refer to the *subcategory* of session state which we describe below.

We use the example of a user working on a web-based enterprise-scale customer support application to illustrate how session state is often used. The user is simultaneously handling multiple customer email requests, and possibly taking a phone call.

A large class of applications, including J2EE-based and web applications in general, use the interaction model below:

- User submits a request (to add notes to a case), request is routed to a stateless application server. This server is often referred to as the middle-tier.
- Application server retrieves the full session state for user (which includes the current application state).
- Application server runs application logic (to add the notes to the case)
- Application server writes out entire session state
- Results are returned to the user's browser

Session state must be present on each interaction, since user context or workflow is stored in session state. If it is not, the user's workflow and context is lost, which is seen as an application failure to the end user, and is usually unacceptable from a product requirement standpoint. Session state retrieval is in the critical path of the control path – processing cannot continue unless session state has been retrieved. Typically, session state is on the order of 3K-200K [2].

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

1. **Is accessed in a serial fashion (no concurrent access).** Each user reads her own state. Unlike state in its full generality, session state is accessed in a fixed pattern of alternating reads and writes: *Read 1* of session state for user U is followed by *Write 1,* which is followed by *Read 2,* followed by *Write 2.*
2. **Is semi-persistent.** Session state must be present for a fixed interval *T*, but can be deleted after *T* has elapsed. *T* is application specific, and usually on the order of minutes to hours.
3. **Is keyed to a particular user.** An advanced query mechanism to do arbitrary searches is not needed.
4. **Is updated on every interaction.** Session state such as user context in a web-based application is updated in its entirety on every interaction, as described earlier. A new copy of the state is written on every interaction.
5. **Does not need ACID [7] semantics**. Session state is transient, and state that requires transactions is not included in the class of session state we address.

Given these properties, the functionality necessary for a session state store can be greatly simplified (each point 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.
2. **State stored by the repository need only be semi-persistent –** a temporal, lease-like [3] guarantee is sufficient, rather than the "durable" guarantee that is made in ACID [7].
3. **Single-key lookup API is sufficient.** Since state is keyed to a particular user and is only accessed by that user, a general query mechanism is not needed.
4. **Previous values of state keyed to a particular user may be discarded.**
5. **Only atomic update is necessary**; partial writes of a user's state would result in internal inconsistency and incorrect behavior. Each write writes out all of the user's session state; consistency is trivial and isolation is guaranteed. Durability is not needed.

IEEE COMPUTER SOCIETY

3. Inadequacy of existing solutions

Currently, session state storage is done with one of the following mechanisms: Relational database (DB), file system (FS), single-copy in-memory, replicated in-memory.

Frequently, enterprises use either the DB or 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 to handle session state, besides the costs of additional licenses and complexity of administration:

- **D1 Contention.** Unless a separate DB/FS is created for session state, requests for session state and requests for persistent objects contend for the same resources. Session state read/write requests are frequent, which can interfere with requests for persistent objects that are housed by the same physical resource.
- **D2 Failure and recovery is expensive.** If a crash occurs, recovery of the DB or FS may be slow, often on the order of minutes or even hours. Recovery time for a DB can be reduced if checkpointing is done frequently; however, checkpointing reduces performance under normal operation. There exist DB/FS solutions that have fast recovery, but these tend to be quite costly [9]. Even if recovery is on the order of seconds, in a large scale application, hundreds or thousands of users may see a failure if they attempt to contact the server at the time of recovery.
- **D3 Session cleanup is an afterthought.** After state is put into a DB or FS, some process has to come back and look at the data and expire it, or else the data continues growing without bound. Reclaiming expired sessions degrades performance of other requests to the DB or FS.
- **D4 Potential performance problems**. Reading/writing state objects to a DB/FS may sometimes incur a disk access in addition to a network roundtrip.

On the other hand, in-memory solutions (IMS) avoid several of the drawbacks of FS and DB, and are generally faster than FS/DB oriented solutions. Existing in-memory solutions require a user to be pinned to a particular server. The application-processing tier is no longer stateless; session state is being stored by the application server; it must serve the dual roles of application processing as well as providing state storage. Because of pinning, load-balancing can only be done at the user level and not at the request level.

If only a single copy of a user's session state is stored on a corresponding application server, on a server crash, state for some users is lost. The crash will be manifested to users as an app failure, which is usually unacceptable.

A primary-secondary scheme is often used for a replicated solution, such as the one adopted by BEA WebLogic™ [5], a J2EE application server. Further details can be found in [5]. Updates are synchronously propagated to both primary and secondary servers.

There are several potential problems (Note that some of the deficiencies of DB/FS solutions are shared by WebLogic™, as mentioned below):

- **D5 Performance is degraded on secondaries**. D5 is related to D1. Instead of only providing application processing, secondary application servers face contention from session state updates that are propagated from the primaries. This is in comparison to the database and file system solution, where a single application server handles a particular user request. In the primary/secondary solution, secondaries must devote resources to being backups in addition to serving requests.
- **D6 Recovery is more difficult (special case code for failure and recovery).** The middle-tier is now stateful, which makes recovery more difficult. Special-case failure recovery code is necessary, e.g. secondaries must detect the failure of a primary and become the primary and elect a new secondary.
- **D7 Poor failure and recovery performance.** If a server A chooses another server B to be its secondary for all requests, assuming equal load across all servers, upon failure of a primary *A*, the secondary *B* will have to serve double load – *B* must act as primary for all of *A's* requests as well as its own. Similar logic applies to *B's* secondary, which experiences twice the secondary load.
- **D8 Lack of separation of concerns.** The application server now provides state storage, in addition to application logic processing. These two are very different functions, and a system administrator should be able to scale each separately.
- **D9 Performance coupling.** If a secondary is overloaded, then users contacting a primary for that secondary will experience poor performance behavior as well. Because of the synchronous nature of updates from primary to secondary, if the secondary is overloaded, e.g. from faulty hardware or user load, the primary

PROCEEDINGS OF THE AUTONOMIC COMPUTING WORKSHOP FIFTH ANNUAL INTERNATIONAL WORKSHOP ON ACTIVE MIDDLEWARE SERVICES (AMS'03) ISBN 0-7695-1983-0/01 $17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY

will have to wait for the secondary before returning to the user, even if the primary is under-loaded [4].

Note that *any* replication scheme requiring synchronous updates will *necessarily* exhibit performance coupling**.** Whenever a secondary is slow for any reason, any primary served by that secondary will block.

If secondaries are chosen on a per-user level (i.e. users sharing the same primary may have different secondaries), after a failure of a single server, the entire cluster will be coupled, assuming that the load balancer load balances correctly. This is particularly worrisome for large clusters, where node failures are more likely because of the number of nodes. An industry expert has confirmed that this is indeed a drawback for existing solutions [6]. Furthermore, application servers often use shared resources such as thread pools, and slowness in the secondary will hold resources in the primary for longer than necessary.

4. Design and Implementation

In this section, we describe the design and implementation of SSM, discussing the salient features of SSM.

We assume a physically secure cluster, along with a commercially-available high throughput, low latency redundant system area network (SAN) that can achieve high throughput with extremely low latency. An uninterruptible power supply reduces the probability of a system-wide simultaneous hardware outage.

We have implemented a working system using Java, running on the UC Berkeley Millennium Cluster [16]. The cluster is composed 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.

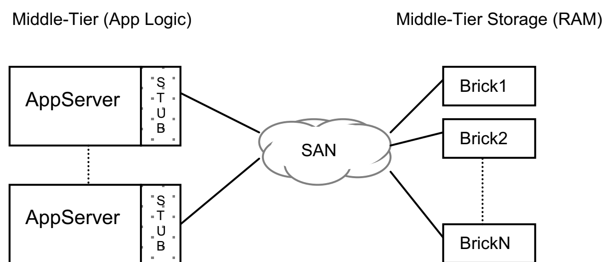

Figure 1. Architecture

Overview

SSM has two components: bricks and stubs. Bricks are the storage mechanism; stubs dispatch requests to appropriate bricks. 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."

A brick stores session state objects by using an in-memory hash table. Conceptually, a brick is simply a processor, a network interface, and memory. Each brick sends out periodic beacons to indicate that it is alive.

The stub is used by applications to read and write session state. The stub interfaces with the bricks to store and retrieve session state. Each stub also keeps track of which bricks are currently alive.

The write interface exported by the stub to the application is `Write(HashKey H, Object v, Expiry E)` and returns a cookie as the result of a successful write, or throws an exception if the write fails. The returned cookie should be stored on the client. The read interface is `Read(Cookie C)` and returns the last 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, we guarantee that the returned value is the most recently written value (recall that the type of session state we are dealing with is accessed serially by a single user).

The stub propagates 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. A stub *attempts* to write to W of the bricks, and read from R bricks. Define WQ as the write quota, which is the minimum number of bricks that must return "success" to the stub before the stub returns to the caller. We use the term quota to avoid confusion with the term quorum; quorums are discussed in section on related work. $WQ - 1$ is the number of **simultaneous brick failures** that the system can tolerate before losing data. Note that $1 \le WQ \le W$, $1 \le R$, and $R \le W$. Lastly, call t the timeout interval, an amount of time that the stub waits for a brick to reply to an individual request. Note that t is different from the session expiration, which is the lifetime of a session state object.

Basic Read/Write Algorithm:

The basic write algorithm can be described as "write to many, wait for a few to reply." Conceptually, the stub

PROCEEDINGS OF THE AUTONOMIC COMPUTING WORKSHOP FIFTH ANNUAL INTERNATIONAL WORKSHOP ON ACTIVE MIDDLEWARE SERVICES (AMS'03) ISBN 0-7695-1983-0/01 $17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY

writes to more bricks than are necessary, 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. The algorithm is described below:

1. Calculate checksum for object and expiration time.
2. Create a list of bricks *L,* initially the empty set.
3. Choose *W* **random** bricks, and issue the write of {object, checksum, expiry} to each brick.
4. Wait for *WQ* of the bricks to return with success messages, or until *t* elapsed. As each brick replies, add its identifier to the set *L*.
5. If *t* has elapsed and the size of L is less than *WQ*, throw an exception indicating that the system is temporarily overloaded. Otherwise, continue.
6. Create a cookie consisting of *H*, the identifiers of the *WQ* bricks that acknowledged the write, and the expiry, and calculate a checksum for the cookie.
7. Return the cookie to the caller.

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

1. Verify the checksum on the cookie.
2. Issue the read to *R* random bricks chosen from the list of *WQ* bricks contained in the cookie.
3. Wait for 1 of the bricks to return, or until *t* elapses.
4. If the timeout has elapsed and no response has been returned, throw an exception indicating that the system is temporarily overloaded. Otherwise, continue.
5. Verify checksum and expiration. If checksum is invalid, repeat step 2. Otherwise continue.
6. Return the object to the caller.

For garbage collection of bricks, we use a method seen in generational garbage collectors [10]. Earlier we described each brick as having one hash table, for simplicity. 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, the stub also sends the key's expiration time, so 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.

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 TCP-like algorithm for maintaining the window; when a request is successfully acked, the window size is additively increased, and when a request times out, the window size is multiplicatively decreased. The stub assumes that whenever a request to a brick times out, it is because the brick is unable to process the request in a timely manner, and therefore reduces its sending 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 rejected until the number of in-flight messages 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 only temporary state for 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 to the caller indicating that the system is temporarily overloaded.

Each brick also performs admission control; when a request arrives at the brick, it is put in a queue. If the 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.

Recovery

If a node cannot communicate with another, we assume it is because the other node has stopped executing.

On failure of a client, the user perceives the session as lost, i.e. if the browser crashes, a user does not expect to be able to resume her interaction with a web application.

On failure of an application server, a simple restart of the server is sufficient since it is stateless. The stub on the server detects existing bricks from the beacons and can reconstruct the table of bricks that are alive. The stub can immediately begin handling both read and

IEEE COMPUTER SOCIETY

write requests; to service a read request, the necessary metadata is provided by the client in the cookie, and 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 necessary. All state on that brick is lost; however, the state is replicated on *(WQ – 1)* other bricks, and so no data is lost. Furthermore, because session state is frequently accessed by the end user with the read/write pattern discussed earlier, on a subsequent user request, a write will create *WQ* new copies\, and the system can once again tolerate *(WQ – 1)* faults without losing data.

An elegant 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 is a candidate for a write. Over time, the new brick will receive an equal load of read/write traffic as the existing bricks.

Recovery Philosophy

Previous work has argued that rebooting is an appealing recovery strategy in cases where it can be made to work [19]: 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. In fact, [19] argues that a principal challenge is to quantify the cost of making a mistake. 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 [20] or other characteristics associated with software aging [21]—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, this makes SSM a *crash-only* subsystem [22].

SSM is being integrated into a larger Web application server called JAGR, many of whose components are designed to have the crash-only property. We anticipate that the ease and speed of recovery of SSM as a subsystem will make it easier to develop system-wide recovery policies for JAGR.

5. Evaluation and Results

We present several micro-benchmarks that suggest that SSM satisfies the previously set forth design requirements. Our load generator is composed of multiple machines. The keyspace is partitioned and a partition is given to each of the load generating machines. Each sending machine then partitions the keyspace further and hands off the partition to a sending thread. Each thread makes a pair of write and read requests for session state through the stub. This models the worst-case behavior where a user writes and reads his state exactly once; this causes the hash tables to continually increase in size. We also conducted benchmarks where a small set of users read and wrote their own state repeatedly; the findings were similar and are not presented here.

In this paper, we assume a fail-stop model, and simulate such failures by killing a process manually. We leave the more general case to future work. We vary the number of users and bricks; results are presented in Figures 2, 3 and 4. In all experiments, *W* is set to 3, *WQ* to 2, *R* to 1, and *t* to 60 ms. The size of the state is 8K.

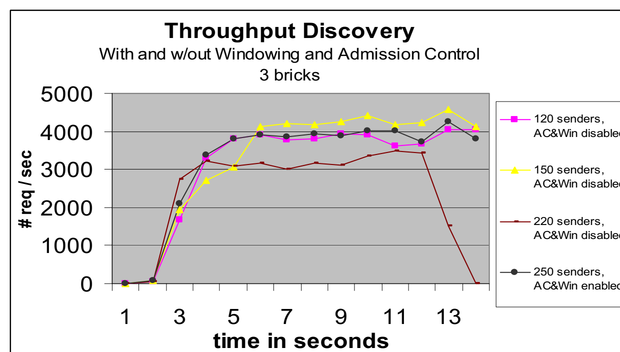

Figure 2. Load capacity discovery

- Self-tuning: the implemented windowing mechanism allows for usage of bricks at full capacity. Figure 2 shows that SSM discovers the throughput capacity of the system. The y-axis labels the number of requests successfully serviced, while the x-axis measures time. Figure 2 shows the throughput for 3 bricks for 120, 150, and 220 sending threads, without any admission control at the stubs or at the

IEEE COMPUTER SOCIETY

bricks. Note that the maximum throughput capacity of the 3 bricks is around 4000 requests per second. As the number of users increases to 220, the system has reached the point of saturation and can no longer serve any requests successfully because the incoming requests are being enqueued at a faster rate than they can be serviced. However, as shown in Figure 2, SSM with windowing and request rejection discovers the maximum throughput capacity of the system, even with 250 sender threads.

- Self-protecting from overload: Using the windowing mechanism, a stub knows when particular bricks are overloaded, and knows when to respond to the caller that the system is overloaded and to try back later. Figure 2 shows that without admission control, system collapse occurs with 220 senders; even with 250 senders, SSM with windowing and request rejection allows for maximum system throughput without collapse.
- Self-healing:
  - SSM can tolerate *WQ*-1 faults without data loss. Furthermore, assuming faults are independent, since the data is replicated in WQ bricks, if a brick crashes, recovery is unnecessary. Data is "rejuvenated" to WQ bricks on the subsequent write request.
  - It is possible to kill at most WQ-1 bricks and restart them without data loss. Furthermore, the system can tolerate W-WQ performance faults without degrading response time.
  - Figure 3 shows that killing a brick reduces the number of requests serviced per second (since the load generated is higher than what 3 bricks can sustain), and that SSM recovers to the appropriate load capacity. Figure 4 shows that when a brick is restarted after a failure, total system capacity is restored. Although SSM has not explored an appropriate policy for killing and restarting bricks, the potential of stopping and restarting bricks so to heal the system is clearly demonstrated by our initial experiments.

If we assume a user "think time" or inter-request interval of 20 seconds, SSM, with 4 bricks, is able to handle approximately 100,000 users with the system parameters described earlier.

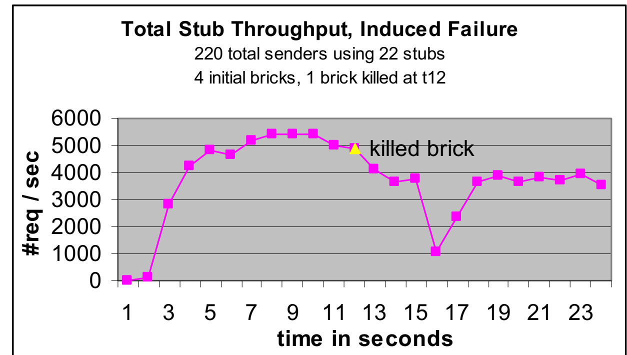

Figure 3. System reaction to brick failure

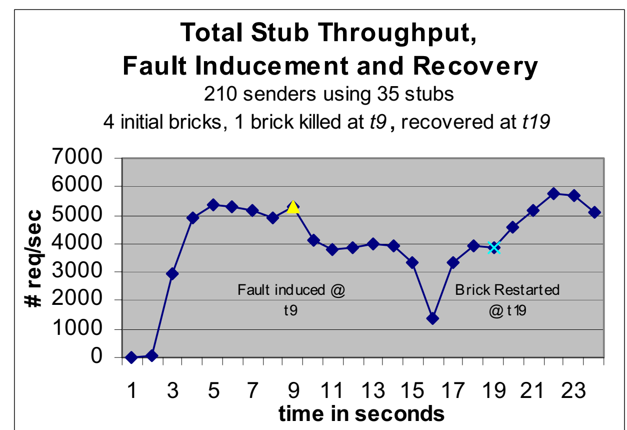

Figure 4. Failure and recovery of brick

6. Future Work

Our preliminary work has shown that SSM reacts well to spikes in workload, and allows requests to flow through the system at maximum capacity without causing system collapse. We hope to demonstrate this with detailed measurements using actual traces from applications. Garbage collection also remains to be implemented.

In this paper, we have assumed a fail-stop model. In the full version of the paper, we hope to explore the effect of faulty bricks (e.g. unusually slow response time, livelock) on the windowing system. We intend on integrating Pinpoint, which monitors systems for anomalous behavior, to detect failing or stuttering bricks.

7. Related Work

In related work, a similar mechanism is used in quorum-based systems [11, 12]. In quorum systems, writes must be propagated to *W* of the nodes in a

IEEE COMPUTER SOCIETY

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.

From distributed database research, Directory-Oriented Available Copies [15] utilizes a directory that must be consulted to determine what replicas store valid copies of an object. This involves a separate roundtrip, and the directory becomes a bottleneck. In our work, we distribute the directory by sending the directory entries to the browser, leveraging the fact that for a given key, there is a single reader/writer.

DDS [4] is very similar to SSM. However, one observed effect in DDS is performance coupling – a given key has a fixed replica group, and all nodes in the replica group must synchronously commit before a write completes. DDS also guarantees persistence, which is unnecessary for session state. Furthermore, DDS exhibits poor behavior in recovery. Even when DDS is lightly loaded, if a node in a replica group fails and then recovers, all data in the replica group is locked while it is copied to the recovered node, disabling write access to any data residing on the replica group. Furthermore, DDS exhibits negative cache warming effects; when a DDS brick is recovered, performance of the cluster first drops (because of cache warming) before it increases. This effect is not present in our work, since recovered/new bricks do not serve any read requests. The failure and recovery of bricks does lock any data and cause any data to be unavailable.

We share many of the same motivations as Berkeley DB [14], 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 [17]. The need to include explicit support for admission control and overload management at service design time was demonstrated in SEDA [18]; 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.

8. Conclusions

Several properties of session state enable us to greatly simplify the design of SSM. We rely on the fact that the cookie names which bricks store the state, and therefore never need to query a majority of the bricks on requests, as in quorum-based systems. Because session-state is frequently updated, SSM does not need to proactively maintain *WQ* copies of the data after a brick crash, since on the next update the data will be rejuvenated, and *WQ* new copies of the data will be written. Furthermore, recovery is simple because a new brick begins servicing new writes, and does not to copy over data from other bricks. Because session state is keyed to a particular user, SSM does not need a full query mechanism, and the standard Java hashtable is sufficient.

We believe SSM properly leverages the properties of session state, providing a self-protecting, self-tuning, self-healing, instantly-recovering session state management layer.

References

[1] Sun Microsystems. J2EE. http://java.sun.com/j2ee/.

[2] U. Singh. Personal communication. E.piphany, 2002.

[3] C.G. Gray and D. R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles,* pages 2002-210, Litchfield Park, AZ, 1989.

[4] S. Gribble, E. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for iIternet service construction. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation,* San Diego, CA, Oct. 2000.

[5] D. Jacobs. Distributed Computing with BEA WebLogic server. In *Proceedings of the Conference on Innovative Data Systems Research,* Asilomar, CA, Jan. 2003.

[6] D. Jacobs. Personal communication, BEA Systems, December 2002.

[7] J. Gray. The Transaction Concept, Virtues and Limitations. In *Proceedings of VLDB,* Cannes, France, Sept 1981.

[9] Network Appliance. http://www.networkappliance.com.

[10] William D. Clinger and Lars T. Hansen. Generational garbage collection and the radioactive decay model. SIGPLAN Notices, 32(5):97—108. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation,* May 1997.

[11] Robert H. Thomas: A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. TODS 4(2): 180-209(1979)

[12] D. Gifford: Weighted Voting for Replicated Data. Proceedings 7th Symposium on Operating Systems Principles: 150-162, 1979.

[14] M. Seltzer and M. Olson. Challenges in embedded database system administration. In Proceeding of the Embedded System Workshop, 1999. Cambridge, MA

[15] Concurrency Control and Recovery in Database Systems, by P.A. Bernstein, V. Hadzilacos and N. Goodman.

[16] Millennium Cluster. http://www.millennium.berkeley.edu

IEEE COMPUTER SOCIETY

[17] V. Jacobson. Congestion Avoidance and Control. In *Proceedings of SIGCOMM 1988* (Stanford, CA, Aug. 1988), ACM.

[18] Matt Welsh and David Culler. Overload Management as a Fundamental Service Design Primitive. In *Proceedings of Eighth Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Elmau, Germany, May 2001.

[19] G. Candea, J. Cutler, A. Fox, R. Doshi, P. Garg, R. Gowda. Reducing Recovery Time in a Small Recursively Restartable System. In *Proceedings of the International Conference on Dependable Systems and Networks* (DSN-2002), Washington, D.C., June 2002

[20] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau, Fail-Stutter Fault Tolerance. In Proceedings of Eighth Workshop on Hot Topics in Operating Systems (HotOS-VIII), Elmau, Germany, May 2001.

[21] Sachin Garg and Aard Van Moorsel and 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 (ISSRE 98)*, Paderborn, Germany, 1998.

[22] G. Candea and A. Fox. Crash-Only Software. In *Proceedings of the Ninth Workshop on Hot Topics in Operating Systems (HotOS-IX)*, Lihue, HI, May 2003 (to appear).

[23] David Oppenheimer and David A. Patterson. Why do Internet Services Fail, and What Can Be Done About It? In Proceedings of 4th USENIX Symposium on Internet Technologies and Systems (USITS '03), Seatttle, WA, March 2003.

[24] Kiran Nagaraja, Ricardo Bianchini, Richard P. Martin and Thu D. Nguyen. Using Fault Model Enforcement to Improve Availability. In Proceedings of the 2nd Workshop on Evaluating and Architecting System Dependability (EASY 2002), San Jose, CA, October 2002.

[22] M. Chen, E. Kiciman, *Using Runtime Path for Macro Analysis*. In *Proceedings of the Ninth Workshop on Hot Topics in Operating Systems (HotOS-IX)*, Lihue, HI, May 2003 (to appear).

IEEE COMPUTER SOCIETY