Cheap Recovery: A Key to Self-Managing State

ANDREW C. HUANG and ARMANDO FOX
Stanford University

Cluster hash tables (CHTs) are key components of many large-scale Internet services due to their highly-scalable performance and the prevalence of the type of data they store. Another advantage of CHTs is that they can be designed to be as self-managing as a cluster of stateless servers. One key to achieving this extreme manageability is reboot-based recovery that is predictably fast and has modest impact on system performance and availability. This "cheap" recovery mechanism simplifies management in two ways. First, it simplifies failure detection by lowering the cost of acting on false positives. This enables one to use statistical techniques to turn hard-to-catch failures, such as node degradation, into failure, followed by recovery. Second, cheap recovery simplifies capacity planning by recasting repartitioning as failure plus recovery to achieve zero-downtime incremental scaling. These low-cost recovery and scaling mechanisms make it possible for the system to be continuously self-adjusting, a key property of self-managing systems.

Categories and Subject Descriptors: C.0 [**Computer Systems Organization**]: General—*System architectures*; C.4 [**Computer Systems Organization**]: Performance of Systems—*Reliability, availability, and serviceability*

General Terms: Design, Measurement, Reliability

Additional Key Words and Phrases: Cluster hash table, manageability, quourum replication, storage systems design

1. INTRODUCTION

In current computer systems, the cost of administration dwarfs the cost of hardware and software. With typical companies requiring one administrator per 1-10 terabytes and data demands growing to the petabyte range, simplifying state management is increasingly important [Gray 2003]; this is especially true for Internet services, which must deliver content from massive datasets for fractions of a penny per access. In recent years, to address the system administration challenges, large research and development efforts have been

This work was funded by grants from the National Science Foundation, gifts from Microsoft Corporation and Hewlett-Packard Corporation, and the IBM Faculty Award.
Author's address: A. C. Huang, Stanford University, Gate 252, Stanford, CA; email: andy.huang@stanford.edu.

undertaken to design systems that can be managed with little or no human intervention [Gibbs 2002; Microsoft Corporation 2004; IDC 2003; Sun Microsystems 2002; Ganger et al. 2003]. This article addresses a slice of the larger "self-management" problem and focuses on designing extremely easy-to-administer state stores for large-scale Internet services.

A practical illustration of a self-managing system is a cluster of stateless HTTP frontends, which can be managed with a few simple mechanisms that lend themselves to automation. First, proactive rolling reboots and reboots of potentially anomalous nodes are used to resolve transient failure and performance degradation, or "fail-stutter," caused by factors like memory leaks and other software aging effects [Arpaci-Dusseau and Arpaci-Dusseau 2001; Garg et al. 1998]. Second, adding and removing nodes in a "plug-and-play" manner enables one to reactively scale the system to match current load [Fox et al. 1997]. What makes these simple failure-handling and load-provisioning policies so effective is the existence of low-cost mechanisms for recovery and scaling. The key property that makes these mechanisms low cost is interchangeability. Since any node can serve any request, node shutdown, recovery, and scaling can occur independently of other nodes.

In contrast, many stateful systems have higher recovery and scaling costs. Relational databases and network file systems can take minutes to recover from a failure, and scaling performance capacity requires administrator intervention and nontrivial downtime [PASS Consulting Group 2001; Rowe 2002]. Even in systems that mask failures with failover nodes, when read-one-write-all (ROWA) and primary-secondary replication schemes are used, the failover process can take over a minute [Rowe 2002] and recovery may involve freezing writes while missed updates are copied to the recovering node [Liskov et al. 1991; Gribble et al. 2000].

When recovery and scaling incur significant downtime and administration cost, the burden of ensuring high availability falls on the accuracy of failure detection and capacity planning. In the presence of all types of transient failures, including fail-stutter, *fast* failure detection is at odds with *accurate* failure detection—reacting quickly to potential failures may lead to false positives, while waiting to collect enough observations to accurately identify failures may results in a higher mean-time-to-repair. In Internet services where load can vary by an order of magnitude or more, resource provisioning involves a difficult tradeoff—heavy overprovisioning results in unnecessary operating costs, while allocating too few nodes results in higher administration cost and system downtime due to having to scale the system repeatedly.

These failure detection and capacity planning challenges would largely vanish if stateful systems could be managed with the same set of simple mechanisms and policies that have been found to be so effective for stateless frontends. This article provides some insight on how this can be accomplished by examining the design of two "crash-only" state stores in which components are designed to crash safely and recover quickly [Candea and Fox 2003]. First, we demonstrate that by decoupling nodes, we can make recovery extremely "cheap" in that it is predictably fast and has predictably small impact on system

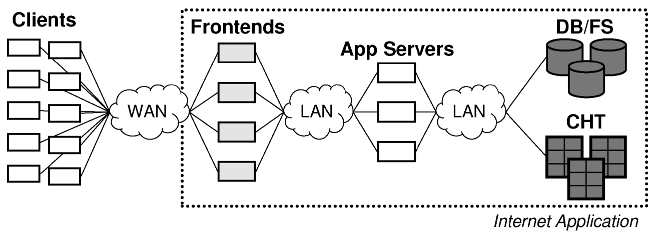


Fig. 1. Tiered Internet application that uses cluster hash tables (CHTs) as part of its overall storage solution.

availability and performance. In contrast, maintaining strict consistency among replicas in ROWA causes coupling of performance and availability between nodes [Gribble et al. 2000]. Second, we demonstrate that cheap recovery, by lowering the cost of recovery and scaling, helps address the failure detection and capacity planning challenges. More specifically:

(1) We simplify failure-handling by leveraging cheap recovery and aggressive anomaly-detection to turn node degradation into failure followed by recovery.
(2) We simplify capacity planning by recasting repartitioning as failure plus recovery to achieve zero-downtime incremental scaling.
(3) We implement a decoupled persistent state store to serve as a testbed for exploring the benefits of cheap recovery and developing self-management principles.

2. THE EMERGENCE OF CLUSTER HASH TABLES

In large-scale Internet services, *cluster hash tables* (CHTs) have emerged as a critical component in the overall state-storage solution (see Figure 1). One primary advantage of a CHT is its ability to scale linearly to achieve high performance [Gribble et al. 2000]. For this reason, single-key-lookup data such as Yahoo! user profiles and metadata for Amazon catalog items is stored in CHTs [Pal 2003; Vermeulen 2003].[1] Another common design pattern involves using a CHT as a base storage layer and placing more complex query logic in the application. Inktomi's search engine accesses several CHTs on each query, the largest of which is a 10^{12}-entry table that maps a word's MD5 hash to a list of document IDs for pages containing that word [Brewer 2004]. In Ebay, although databases make up its storage layer, complex queries involving cross-node joins and foreign-key constraints are performed in the application to achieve greater scalability [Grossfeld 2003]. These examples show that certain types of data do not require the full generality of databases and can instead be stored in a CHT for improved scalability and performance.

[1]Although "personal communication" does not carry the same weight as published articles, since the information typically comes from senior technical principals, we felt it was valuable to include the citations as examples of existing successful practices in large-scale production services for which there are rarely any publicly-available articles.

To reveal the important principles for building self-managing systems, we designed DStore (Decoupled Storage), a CHT that stores persistent data such as user profiles, catalog metadata, and shared workflow data. This article shows that just as CHTs have been built for immense performance and scalability, by focusing on a particular class of data, we can design CHTs for extreme manageability as well. In the rest of this article, we present DStore's design, implementation, and evaluation, followed by philosophical thoughts on building self-managing systems in general.

3. ACHIEVING CHEAP RECOVERY

To make recovery cheap in DStore, we follow two design principles based on the idea of decoupling. The first principle is to tolerate replica inconsistency by using quorum-based replication [Gifford 1979]. In the basic quorum scheme, reads and writes are performed on a majority of the replicas. Since the *read set* and *write set* necessarily intersect, when we use timestamps to compare the values returned on a read, the most up-to-date value is returned. Thus, quorums allow some replicas to store stale data while the system returns up-to-date data. What this means in practice is that a failed replica does not need to execute special-case recovery code to freeze writes and copy missed updates.

Although a wealth of prior work uses quorums to maintain availability under network partitions and Byzantine failures [Davidson et al. 1985; Pierce 2000], few real-life systems do this, perhaps because these failure modes are too rare [Wool 1998]. Instead, we use quorums to simplify the mechanisms for adding new nodes and rebooting failed nodes, which are frequent occurrences in Internet services. The main cost of quorum-based replication is storage overhead, which we address in the next section.

The second principle is to avoid locking and transactional logging by using single-phase operations for updates. In replicated state stores that use two-phase commit, recovery involves reading the log and completing in-progress transactions by contacting other replicas. Meanwhile, replicas holding data locks for in-progress transactions may be forced to block, causing data unavailability during both failure and recovery. Although the quantity of data affected by blocked transactions is likely to be small, since users are currently accessing the data, the impact of making in-progress data unavailable is high compared to taking an arbitrary portion of the dataset offline. By using single-phase operations, we avoid locking data during failures and cleaning up those locks on recovery. The main cost of single-phase operations are extra consistency mechanisms that incur extra cost to certain read requests and to the application logic.

Our design principles point to two forms of coupling—the strict consistency in ROWA and the locking in two-phase commit. To completely eliminate coupling, replicas must be completely interchangeable so that any request can be serviced by any node. This is difficult in practice when persistent state is involved. We, therefore, approximate this goal by making certain that a request never depends on the availability of any *specific* replicas, but rather on some minimum-sized *subset* of replicas.

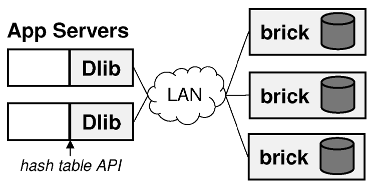


Fig. 2. DStore architecture: Dlibs are libraries, which expose a hash table API and service requests from the application by issuing distributed quorum operations on bricks, which store persistent data.

4. CHEAP RECOVERY TRADEOFFS

To understand the tradeoffs involved in using quorums with single-phase writes, it helps to have a basic understanding of DStore's architecture, shown in Figure 2. *Dlibs* (DStore libraries) expose a hash table API and service requests by acting as the *coordinator* for distributed quorum operations on *bricks*, which store persistent data. Based on typical uses of CHTs, we assume the following usage model: when a user issues a request to the Internet service, the request is forwarded to a random application server, which performs one or more hash table operations on the Dlib to fulfill the request.[2] For the discussion on consistency, we consider consistency from the point of view of individual Dlibs making multiple requests, as well as from the point of the view of the end-user. However, if a single end-user issues requests from two separate browser windows, we consider each window to be a separate user.

To handle a request, the Dlib first identifies the bricks responsible for storing the given key (*replica group*). For writes, the Dlib issues the write to all bricks in the replica group, and waits for a majority to respond. As is typical with hash tables, writes completely overwrite the current value. For reads, the Dlib queries a majority of bricks in the replica group and uses timestamps to determine which value to return. As in Phalanx [Malkhi and Reiter 1998], before returning, if the timestamps do not match, the Dlib issues *read-repair* operations with the up-to-date value-timestamp pair to bricks that returned stale values. This ensures that a majority of the bricks have the up-to-date value upon returning from the read.

4.1 Quorums and Storage Overhead

Quorums require a higher degree of replication than ROWA to achieve an equivalent level of fault tolerance because tolerating N failures requires $N+1$ bricks with ROWA, and $2N+1$ bricks with quorums. To capture the entire cost, however, we must consider common failure scenarios and clarify what it means to "tolerate failures." When considering overall availability, one must take into account availability during recovery. In ROWA, bringing the failed brick up to date causes a big dip in write availability [Gribble et al. 2000], whereas in quorums with read-repair, the cost of recovery is spread out over time as on-demand repair operations cause a slight dip in write throughput and an increase in read

[2]Although session affinity is often used to route all of a user's requests to the same application server, we do not rely on this mechanism.

latency for some reads. Therefore, if one needs to meet a certain minimum level of service, using quorums can actually make provisioning for failures simpler and less costly.

Quorums also require a greater number of replica groups to match the read performance of ROWA. On a read, 1 brick is queried in ROWA, while $\lceil (N+1)/2 \rceil$ bricks are queried in quorums. To alleviate this, we use a *read-timestamp* optimization in which we read the value-timestamp pair from one replica and treat the remaining replicas like *witnesses*, reading only timestamps even though they also store the data [Paris 1986]. This technique is effective if the value returned is up-to-date and if reading a timestamp is faster than reading the actual value. Since writes are issued to all bricks, the value returned is usually up-to-date, which avoids having to issue a second request to obtain an up-to-date value. Furthermore, when the value size is large, compared to the 8-byte timestamp size, most timestamps in the working set can be cached in the brick's RAM. Under these conditions and assuming that extra communication cost can be absorbed by the network, the overhead of reading timestamps from an in-memory cache is insignificant compared to the cost of reading a value from disk. If, instead, timestamps do not fit in memory, the resulting performance penalty can be resolved by adding more replica groups. However, since administration costs make up a large fraction of the overall storage cost, the cost increase is offset by the simplified management that cheap recovery provides.

4.2 Single-Phase Writes and Consistency

The main challenge in using single-phase operations is ensuring consistency. Two-phase commit guarantees sequential consistency, a concept first defined for shared-memory multiprocessor (SMP) systems:

> [A system is sequentially consistent if] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by its program. [Lamport 1979]

A "processor" in the above definition is analogous to a Dlib, which issues load-and-store requests to execute the program logic. Meanwhile, an SMP memory is analogous to a DStore brick, which executes load-and-store operations. Therefore, sufficient conditions for sequential consistency in DStore are as follows:

C1 All Dlibs issue brick requests in program order.

C2 If a Dlib issues a `get`, the Dlib may not issue another request until the value which is to be read has both been bound to the `get` operation and become accessible to all other Dlibs.

C3 If a Dlib issues a `put`, the Dlib may not issue another request until the value written has become accessible by all other Dlibs [Scheurich and Dubois 1987].

Since Dlibs do not reorder requests, C1 is satisfied. The rest of this section describes the mechanisms DStore uses to meet C2 and C3 to achieve sequential consistency.

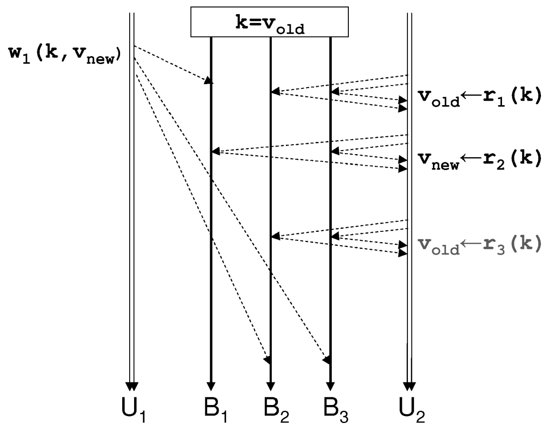


Fig. 3. Partial write: Delayed requests or Dlib failures can result in an inconsistent read.

4.2.1 *Ordering Concurrent Writes.* First, we describe how DStore orders concurrent writes. Rather than locking data, Dlibs determine the global update order by generating a globally-unique physical timestamp (local time, IP address) on each update. Following the Thomas Write Rule [Thomas 1979], bricks execute the update only if the new timestamp is more recent than the current timestamp. Thus, bricks "agree" on the update order without explicit coordination. However, since timestamps are generated from local clocks, we must be careful to avoid *lost writes*, in which a past write is effectively overwritten by a more recent write because the clocks are out-of-sync:

```
U1:  w1(k,a,ts1)
U2:                    a←r1(k)   w2(k,b,ts2)   a←r2(k)
     ─────────────────────────────────────────────────→ time.
```

Since $r_2(k)$ returns the previously-written value a, it must be the case that $ts_2 < ts_1$, resulting in $w_2(k, b, ts_2)$ being lost. However, $w_2 \to w_1$ is inconsistent with the order as seen by U_2. To prevent inconsistency, bricks return a *timestamp error* for w_2 along with the current timestamp ts_1. This allows the Dlib to update its clock, generate a new timestamp, and retry the request. As an optimization, clocks can be synchronized to reduce the occurrence of timestamp errors.

4.2.2 *Read Repair Satisfies C2.* For a given key, if `get(k)` returns v, a subsequent request by any Dlib must return a value no older than v. As long as the quorum majority invariant is maintained, the most recent value is returned on all `get` requests, satisfying C2. However, as shown in Figure 3, a delayed request, or a Dlib failing while processing a `put`, can cause a *partial write*, where the write has reached some, but not a majority, of the bricks.[3] When this occurs, depending on which bricks are queried on a `get`, different values are returned.

Figure 4 shows how the read-repair mechanism described earlier resolves partial writes by committing the partially-written value v_{new} before the `get` request $r_2(k)$ returns. Once v_{new} is committed, all future `get` requests return v_{new}. It follows that `get` requests issued prior to the commit point returned v_{old};

[3]Note that the unlabeled timeline next to the user's timeline represents a random Dlib that may be different for each user request.

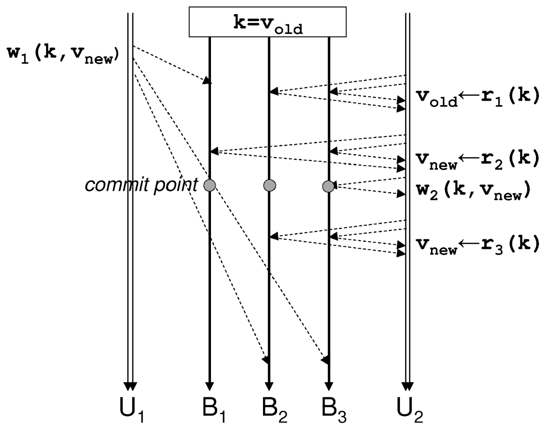


Fig. 4. Read repair: Once v_{new} is returned, all subsequent requests return a value at least as recent as v_{new}.

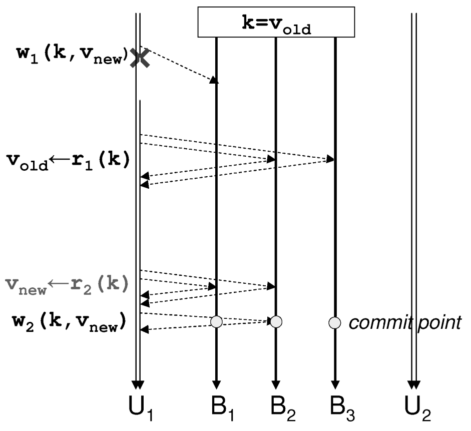


Fig. 5. Inconsistent orsering: Read repairs delayed-commit semantics causes, U_1 to see inconsistent ordering.

otherwise, v_{new} would have been committed already. In summary, before each get request returns, read repair ensures that the quorum invariant is upheld, thus satisfying C2. Another way to look at it is that read repair binds all get requests to a fixed value between commit points.

4.2.3 *Write-In-Progress Cookie Satisfies C3.* In Figure 4, as far as U_2 can tell, the write appears to have been executed atomically at the commit point. However, as shown in Figure 5, since U_1 knows when partial write w_1 was issued, seeing the update being committed at a later time results in inconsistency. In general, if a user issues a put and issues another request before the write is commited, C3 is violated.

When the application server a Dlib resides on fails, the Web server tier can retry the request on a different application server, or an application-level error can cause the user to resubmit the request via HTTP Retry-After. In either case, any partial write that occurred is overwritten and U_1 sees consistent

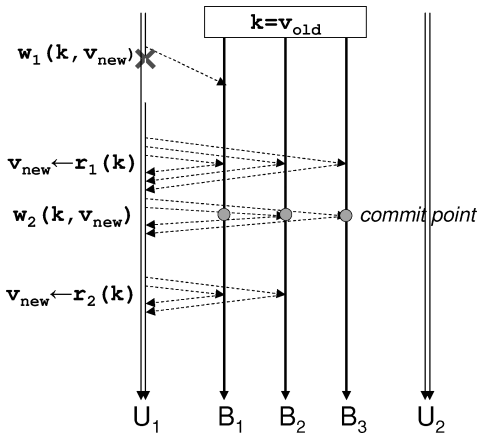


Fig. 6. Write-in-progress cookie: Before U_1 issues another request, any partial writes are committed or aborted, thus ensuring consistent ordering.

event ordering. If instead, U_1 issues a get before retrying, we must commit any partial writes made by U_1, before processing any other requests. To accomplish this, we must record the fact that a write is issued but has not completed. Rather than adding another phase to the write protocol to keep a write history on the bricks, we effectively store the "state" associated with two-phase commit at the client. When U_1 clicks *submit*, client-side JavaScript code writes an *in-progress* cookie on the client before the request is submitted. Upon success, the server returns a replacement cookie with the in-progress flag cleared. On a subsequent read, the cookie is sent along with U_1's request. Shown in Figure 6, if the cookie's in-progress flag is not cleared, the Dlib reads the values from all bricks to find the most recent value and writes that value to a majority of the bricks, reestablishing the quorum invariant. Thus, C3 is satisified because U_1 is not allowed to issue another request until U_1's previous put is committed. As an alternative to using the write-in-progress cookie, Internet services can implement other application-level techniques to force a user to reissue the put request before performing another operation.

4.2.4 *Recovery Tradeoffs Summary.* In summary, quorums can be used to simplify recovery and keep data available throughout recovery at the cost of extra storage and communication overhead. Although two-phase commit can be used on top of quorums, we use single-phase operations to further simplify recovery and keep all data available during failure. Together, these techniques allow us to research the extreme design point of cheap recovery. Table I summarizes the practical tradeoffs between two-phase commit and single-phase operations.

To make the consistency discussion more concrete, we conclude this section by listing some applications for which DStore's API and consistency guarantees would and would not be appropriate:

—Single-user data *(yes)*: user profiles, shopping carts, workflow data for tax returns,

Table I. Practical Tradeoffs Between Two-Phase Commit and Single-Phase Operations

Property	2-Phase Commit	Single-Phase Operations
Recovery	Read log to complete in-progress transactions	No special-case recovery
Availability	Locking may cause requests to block during failures	No locking
Performance	2 synchronous log writes 2 roundtrips	1 synchronous update 1 roundtrip
Other costs and benefits	Supports multi-operation transactions	Read repair causes slight performance degradation Relies on client to store write-in-progress cookie

—Single-producer multi-consumer *(yes)*: catalog metadata, search engine data (retailers/crawlers make updates, which are reflected on the site)

—Multi-producer multi-consumer *(probable)*: auction bids (users submit bids, which other users see after some delay)—requires application-level checks that may require atomic compare-and-swap [Gribble et al. 2001],

—Non-overwriting objects *(no)*: workflow data for insurance claims (multiple users update separate portions of a single document)—since updating stale data on a quorum replica has undefined results, use ROWA instead.

5. IMPLEMENTATION DETAILS

In this section, we discuss details of the DStore implementation. Recall that DStore is composed of two components, Dlibs and bricks. A Dlib is a Java class that presents a single-system image with the consistency model detailed in the previous section. The Dlib API exposes `put(key,value)` and `get(key)` methods where keys are 32-bit integers and values are byte arrays. Dlibs service requests by issuing read/write requests to bricks via TCP/IP socket connections. In order to act as the *coordinator* for these distributed operations, Dlibs maintain soft state metadata about how data is partitioned and replicated. For each brick, Dlibs also maintain TCP-style (additive-increase, multiplicative-decrease) sending windows to help prevent overload. Finally, Dlibs maintain request latency statistics, which are used in making repartitioning decisions.

Bricks store persistent data accessed via the brick API—`write(key,value, ts)`, `read_val(key)`, and `read_ts(key)`. For reading data, `read_val` returns a value-timestamp pair, while `read_ts` returns only the timestamp. On a `write`, the brick checksums the key-value-timestamp object and writes it synchronously to disk. Bricks also cache timestamps in an in-memory Java Hashtable, and use the file system buffer cache to cache values. Since the timestamp cache is updated after the value is written to disk, if a brick fails while processing a write, on a read, it conservatively returns a timestamp that is no newer than the timestamp of the value on disk.

Currently, data is stored in files as fixed-length records where the record size is configured at table-creation time. Although this scheme is constrained by its fixed-length nature, it is appropriate for datasets with low value-size variance, such as Amazon's catalog metadata database, which has fixed-size entries [Vermeulen 2003]. Since the underlying storage scheme is orthogonal to DStore's design techniques and is not critical for evaluating the system's recovery and manageability, simplicity is the main reason this scheme is used. Nevertheless, bricks are designed so that it is easy to plug in different storage schemes. For example, implementing wrapper code for Berkeley DB [Olson et al. 1999] took less than an hour and changing storage schemes takes a matter of seconds.

Bricks maintain two open TCP/IP socket connections (send and receive) per Dlib, one control channel used for initiating recovery and repartitioning, and three message queues (read, put, and ts) serviced by individual thread pools. Like router QoS queues, differentiating requests enables administrators to provision resources and maintain a minimum level of service for each request type. Furthermore, differentiating longer-running write requests from read requests reduces the service time variance, and subsequently, the average queuing delay [Trivedi 2002].

5.1 Keyspace Partitioning

Storage and throughput capacity is scaled by horizontally partitioning the hash table across bricks. Each partition is replicated on a set of bricks to form a *replica group*, the unit upon which quorum operations are performed. Keys are partitioned across replica groups based on the keys' least-significant bits, called the *replica group ID* (*RGID*). Later we show how the keyspace can be dynamically repartitioned without loss of availability to compensate for an uneven workload or heterogeneous brick performance.

Upon startup, each brick is configured with an RGID-mask pair, which it beacons periodically to distribute metadata and indicate liveness. For example, if brick B beacons $(1, 11)$, this tells Dlib's to use the last two bits of the key to find B's RGID, 01. Using this information, Dlibs build a soft-state RGID-to-brick mapping (*RGID map*). To find the replica group storing a given key, Dlibs find the entry with the longest matching suffix.

The beaconing period is a system-configurable parameter (currently set to two seconds), so in between beacons, a Dlib's RGID map may become stale. Since each brick is the ultimate authority of its own RGID, if a Dlib sends a request to the wrong brick, the brick returns a `WRONG_REPLICA_GROUP` error. Once the Dlib's RGID map is updated on the next beaconing period, the request is sent to the correct set of bricks. Finally, a brick can be a part of more than one replica group by announcing multiple RGIDs. This feature is used to spread load across a heterogeneous set of bricks and in the repartitioning algorithm described later.

5.2 Detailed Algorithms

DStore's `put` and `get` algorithms, which were outlined earlier, are described here in full detail. For this discussion, let N be the number of bricks in a replica group, let WT (*write threshold*) be the minimum number of bricks that must reply before a `put` returns, and let RT (*read threshold*) be the minimum number of bricks that are queried on a `get`. In DStore, we choose *WT* and *RT* to be a majority—$\lceil (N + 1)/2 \rceil$. However, in general, *WT* and *RT* can be chosen such that the read and write sets intersect: $WT + RT > N$.[4]

Dlib put: On a `put`, the Dlib generates a physical timestamp from its local clock and appends its IP address. The timestamped value is sent to the entire replica group, but the Dlib only waits for the first *WT* responses to ensure the quorum majority invariant holds, and ignores any subsequent responses. Pseudo-code for all algorithms are provided in the appendix.

Brick write: A brick overwrites a value only if the new value has a more recent timestamp. Otherwise, the brick returns `TIMESTAMP_ERROR`.

Dlib get: On a `get`, the Dlib selects *RT* bricks from the replica group, and issues a `read_val` request to one brick and `read_ts` requests to the remaining $RT-1$ bricks. Since the relative sending window capacities reflect relative brick load, Dlibs use this measure to perform simple load-balancing for read requests. After the value and all the timestamps are returned, the Dlib calls the `check` method to confirm that the value is up to date.

Dlib check: On each `get`, two checks are performed on the timestamps. First, the Dlib checks whether the timestamp for `value` is the most recent one returned. If it is, `value` is returned—Otherwise, the value is read from the brick that returned the most recent timestamp. Second, the Dlib checks whether at least *WT* bricks have the most recent timestamp. If not, the most recent value and timestamp are written back to ensure that enough bricks contain an up-to-date value. This repair mechanism is used to repair partial writes arising from Dlib failures.

5.3 Recovery Mechanism

The final implementation detail we discuss is the brick recovery mechanism, which is used by the failure detection mechanisms described later. To restart a failed brick B_f, a Dlib scans its RGID map for the brick with the next-highest IP address B_r, and sends a `RESTART_BRICK` message to B_r's control channel. If B_r does not respond because it too has failed, the brick with the next-highest IP address is asked to restart both bricks. The Dlib also removes B_f from its RGID map so that requests are not sent to B_f until it recovers.

To restart the brick, B_r runs a script that first sends a `kill -9` to B_f's brick processes. Next, the script attempts to restart the brick processes. If instead, B_f's machine does not respond at all, the node can be restarted using an IP-addressable power source. Since multiple Dlibs are likely to detect B_f's failure, B_r does not perform recovery more than once every four seconds (two times the beaconing interval). Further, since restarting a brick only cures transient failures, if the brick has been restarted more than a threshold number of times in a given period, it is assumed the brick has a persistent fault and should be taken offline. Automatically reconstructing data after disk failures and allowing failed components to be replaced en-masse, say on a weekly basis, are next steps in our future work.

[4]Quorum systems generally also require that $2 * WT > N$ so that simultaneous writes can be ordered. However, we use physical timestamps, which eliminates this requirement.

When a brick recovers, it does not freeze writes to copy missed updates or read a log to complete in-progress transactions. Rather, the recovered brick is incrementally updated using on-demand read-repair. When a recovered brick starts up, the timestamp cache must be loaded before system capacity is fully restored. Timestamp loading can be configured in one of two ways. If cache preloading is on, capacity is restored once the brick comes online. Otherwise, timestamps are loaded after start up in an on-demand fashion.

6. RECOVERY BENCHMARKS

In this section, we evaluate how well DStore achieves its goal of cheap recovery. We first measure basic performance and scalability, and then show system behavior during failure and recovery.

6.1 Benchmark Details

All benchmarks are run on the UC Berkeley Millennium cluster, which has 40 PCs with dual 1GHz Pentium III CPUs, 1.5GB RAM, and dual 36GB IBM UltraStar 36LZX hard drives. Nodes are connected by a 100Mb/s switched Ethernet. A single instance of a brick or client application is run on each node using Sun's JDK 1.4.1 on top of Linux 2.4.18. DStore is configured with three bricks per replica group ($N = 3$), and 1-KByte records. The number of threads allocated to service the write, read_val, and read_ts queues is 14, 32, and 10, respectively, which produces a workload mix of around 2.5 to 5 percent writes. To generate load, clients perform closed-loop operations[5] on random keys and 1-KByte values. This 1-KByte value size falls within the range of typical hash table object sizes in Internet services [Pal 2003]. Enough clients are run to saturate the bricks in steady-state with writes accounting for approximately 2% of the workload.

6.2 Steady-State Performance

Although designed for manageability, DStore retains the high performance and scalability of CHTs. First, we show that under reasonable cache hit rates, the read-timestamp optimization gives quorums read performance comparable to ROWA. Second, we show that DStore throughput scales linearly with the number of bricks.

Timestamp read overhead. To evaluate the effectiveness of DStore's read-timestamp optimization, we compare the read performance of ROWA versus quorums under different cache hit rates. We run a three-brick DStore with 2GB of data and induce approximate cache hit rates between 60 and 100 percent. The client induces a hit rate c by generating a random integer i, between 1 and 100, on each request. If $i \leq c$, the key k is chosen to fall within the 1.2GB working set: $0 \leq k < 1.2M$. If $i > c$, k is chosen to induce a disk access: $1.2M \leq k < 2M$.

Figure 7 shows that although ROWA outperforms DStore when the working set fits in memory, the disk quickly becomes the bottleneck for lower, more realistic cache hit rates and performing the extra timestamp read

[5]closed-loop = a new request is immediately issued after the previous request returns.

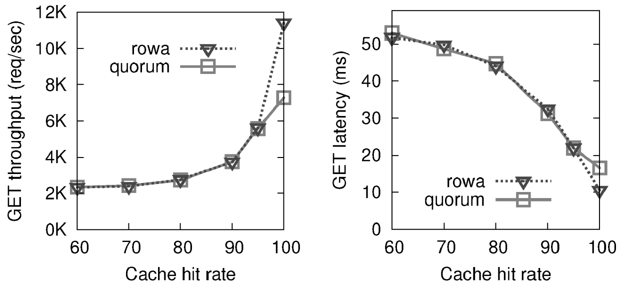


Fig. 7. ROWA vs. Quorums: In quorums, under realistic cache hit rates, reading extra timestamps from cache has little effect on performance because the disk is the bottleneck.

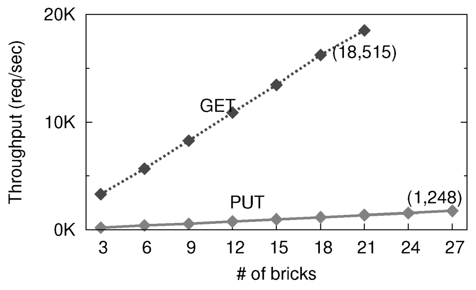


Fig. 8. Linear scaling: GET and PUT throughput scale linearly with the number of bricks (cache hit rate = 85%).

becomes largely insignificant. If timestamps do not completely fit in memory, the 100 percent cache hit rate measurements can be seen as the worst-case overhead.

Linear performance scaling. To evaluate DStore's scaling properties, we induce an 85 percent cache hit rate as described above and measure GET and PUT throughput separately. Figure 8 shows throughput scaling linearly up to 21 bricks, servicing over 18,000 request per second for `gets`, and 27 bricks, servicing approximate 1,200 request per second for `puts`, after which, we did not have enough nodes to scale the benchmark further.

6.3 Fast, Nonintrusive Recovery

Next, we show that recovery is fast and leaves data available for reads and writes throughout failure and recovery. For these benchmarks, we run a three-brick DStore, and induce a single brick failure at time $t = 5$ min. To show the system behavior during failure and recovery, we turn off all failure-detection mechanisms and manually restart the failed brick at $t = 10$ min.

6.3.1 *Basic Recovery Behavior.*

The graphs in Figures 9 and 10 show `get`/`put` throughput and repair operations per second. Each point represents a single measurement (taken once per second) as seen from the clients, and the shaded areas highlight the brick-recovery period. Figure 9 shows recovery behavior when bricks have a 100 percent cache hit rate. As expected, `get` throughput drops by one third during failure, and `put` throughput remains steady because

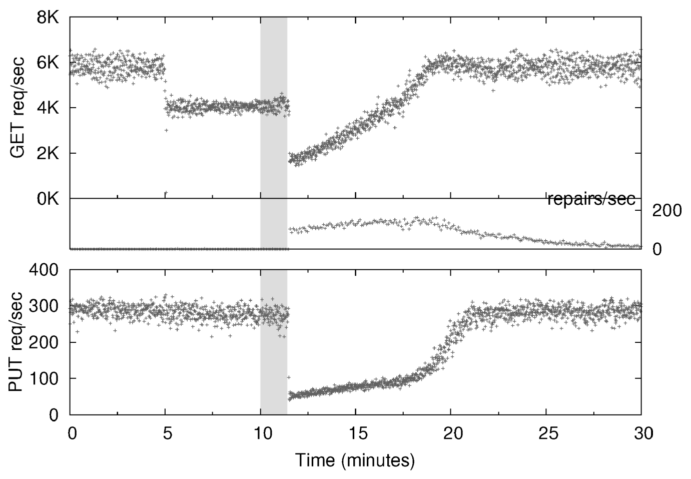


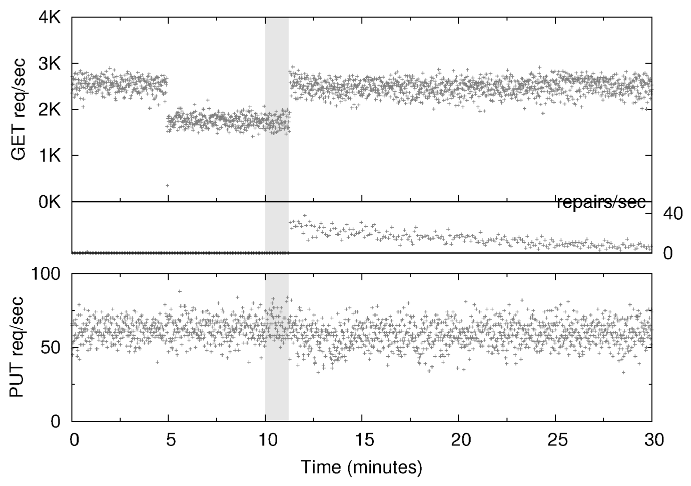
Fig. 9. Unrealistic recovery behavior: Under an unrealistic cache hit rate of 100%, the disparity between disk-bound recovery performance and in-core steady-state performance causes poor recovery behavior.

writes are issued to the entire replica group causing bricks to see roughly the same load no matter how many bricks are available. After recovery, the increase in read-repair operations causes get and put throughput to drop dramatically. Although the percentage of repairs is small, the high cost of disk writes causes reads to become disk-bound. The sudden contention for disk causes put throughput to fall as well. Only after ten minutes does throughput return to normal. Therefore, under an unrealistic cache hit rate, recovery behavior is neither fast nor nonintrusive. However, under more realistic cache hit rates, the recovery behavior differs significantly.

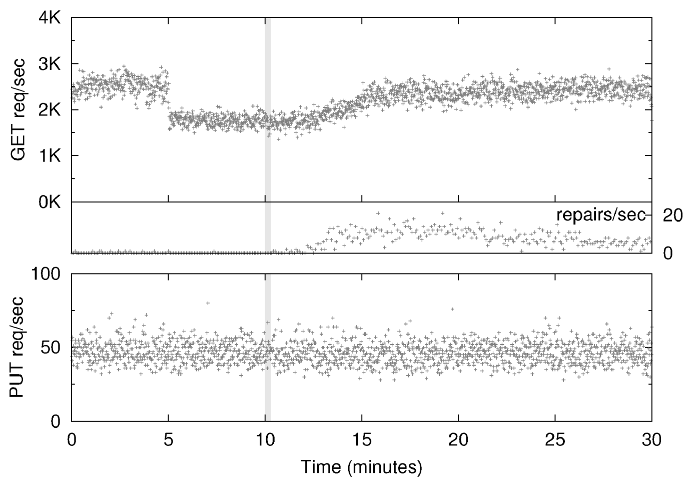
In Internet services, workloads for large data sets are typically disk-bound with cache hit rates ranging from 60 to 90 percent [Pal 2003]. Figure 10 shows recovery behavior under a cache hit rate of 85 percent using two different timestamp-loading schemes. In 10(a), after the brick preloads its timestamp cache and starts up, full system capacity is restored immediately. In 10(b), the brick starts up immediately with capacity gradually ramping up as the brick warms its cache. During cache-warming, load balancing prevents a dip in overall system performance.

The timestamp-loading scheme one chooses depends on the number of put requests that are likely to be missed during recovery. In general, the longer it takes to recover, the more repair operations will be required after start up. If preloading incurs too much downtime, the incremental-loading scheme should be used. In either case, Figure 10 shows that even after five minutes of downtime, get throughput quickly returns to normal and put throughput is barely affected because the small number of repair operations adds little to the disk contention that already exists due to cache misses.

6.3.2 *DStore Versus ROWA Recovery.* Restarting a failed DStore brick takes approximately 12 seconds, which includes the time to detect the failure



(a) Preloading timestamps



(b) Incremental loading

Fig. 10. Expected recovery behavior: With an 85% hit rate, repair operations have little impact on the already disk-bound workload. Timestamp preloading in (a), or window-based load balancing in (b), prevents a cache-warming-induced dip in performance.

and initiate restart (<1 second), start the brick process (~1.5 seconds), and receive beacons from the recovered brick (≤10 seconds). Note that since the beaconing period and the frequency with which Dlibs update their RGID tables are configurable, the duration of the last step can be reduced as needed.

Drawing a fair comparison between recovery in DStore and ROWA is difficult because there are many factors involved. As one point of comparison, recovery in DDS, a ROWA-based CHT, takes 80 seconds, during which writes are frozen while the entire 100-megabyte dataset is copied to the recovered brick [Gribble et al. 2000]. Adding mechanisms to log all updates can drastically reduce the amount of data that needs to be copied. We next provide a quantitative and qualitative analysis comparing DStore to ROWA with log-based recovery.

Let N be the number of updates missed due to failure and c be the cost of a single read repair operation. If every missed update eventually results in a repair operation, the performance cost of data repair for DStore is cN. It is often the case, however, that some updates are overwritten before the recovered brick is queried, making repair unnecessary. If δ denotes the percentage of updates that result in read repair, the data-repair cost is δcN.

For ROWA, let us assume the best case scenario where we know exactly when the brick failed and copy only the N missed updates from the update log and two-phase-commit logs. If we also assume that copying data in bulk is ρ percent cheaper than issuing individual repair operations, the data-repair cost in ROWA is ρcN. Therefore, based on performance alone, data-repair in DStore is cheaper than in ROWA, if $\delta < \rho$.

Since it is unlikely that $\delta \ll \rho$ or $\delta \gg \rho$, one might conclude that DStore and ROWA have comparable recovery costs. However, one should also consider the qualitative differences in recovery behavior. First, recall that during ROWA recovery, data is unavailable for writes. Second, during failure, data that was being updated may become unavailable for both reads and writes. Although blocking in two-phase commit affects a relatively small amount of data, it can have a significant impact on end users because all the data affected is currently in use. Third, although repairing data in bulk may take less time overall, it is likely to have an adverse effect on system performance. In contrast, DStore maintains data availability throughout failure and recovery for both reads and writes, and a predictable level of performance by spreading repairs over a longer period of time.

6.3.3 *Recovery With Overprovisioning.* In practice, cluster operators significantly overprovision their clusters to prevent overload [Duvur 2004; Messinger 2004]. Thus, for the last set of recovery benchmarks, we present an offered load approximately 65 percent of full capacity and induce the same type of failure as in the previous benchmarks. Note that for these and all subsequent benchmarks, we induce an 85 percent cache hit rate and configure bricks to load timestamps incrementally after start up.

As shown in the previous set of benchmarks, losing one brick results in a one-third reduction in throughput capacity. Therefore, in an overprovisioned system, as Figure 11 shows, there is little effect in throughput capacity during brick failure. Latency, however, is affected, because the nonfailed bricks must handle a higher load. Figure 12 shows that latency increases during failure and for awhile after recovery due to the increased number of read-repair operations.

To analyze latency further, Figure 13 shows latency distributions before, during, and after failure. First, notice that the get distribution is bimodal; the large peak results from file-cache hits and the small peak results from cache misses and requests requiring read-repair. During failure, the size of the large peak does not change because the cache hit rate and number read-repair operations remains roughly the same. However, increased disk load increases

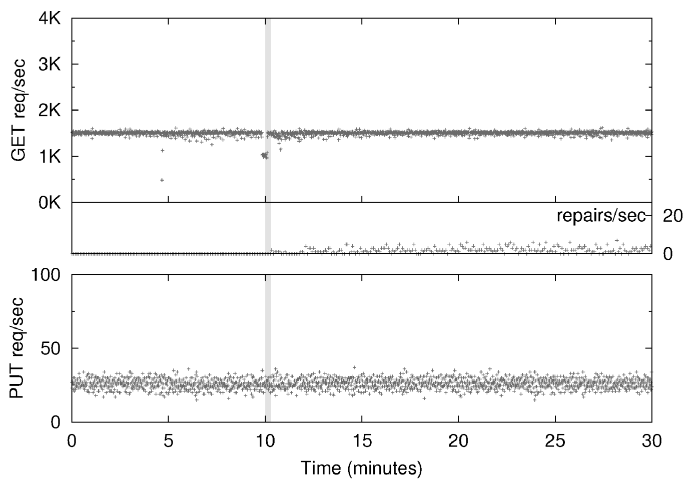


Fig. 11. Overprovisioned throughput: There is little effect in throughput capacity during brick failure and recovery when the system is overprovisioned.

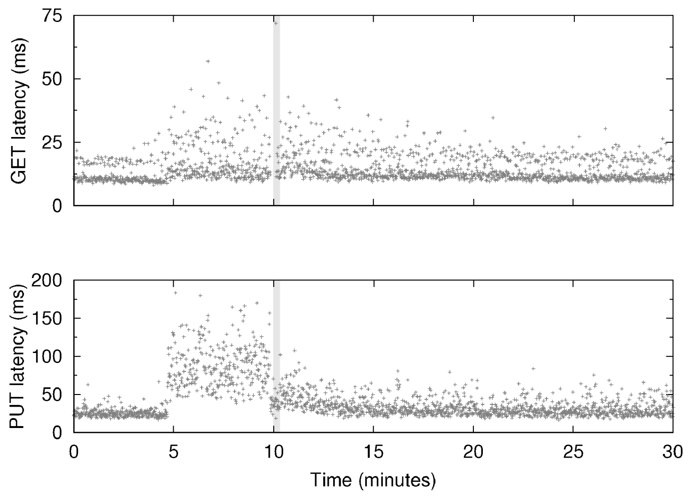


Fig. 12. Overprovisioned latency: Extra read load on the nonfailed bricks causes latency to increase. After recovery, latency gradually decreases as the cache is warmed.

the latency for cache misses, causing the small peak to spread out towards the right. After recovery, an increase in the number of read-repair operations causes a decrease in the large peak's size and an increase in the small peak's size.

For put requests, during failure, the number of requests remains roughly constant because requests are always sent to all available bricks. The increase in latency is due to increased disk contention from the extra get requests. After recovery, the extra get load is removed and is replaced by increased read-repair load. However, the number of repair operations is relatively small, making the latency after recovery roughly equivalent to the distribution before failure.

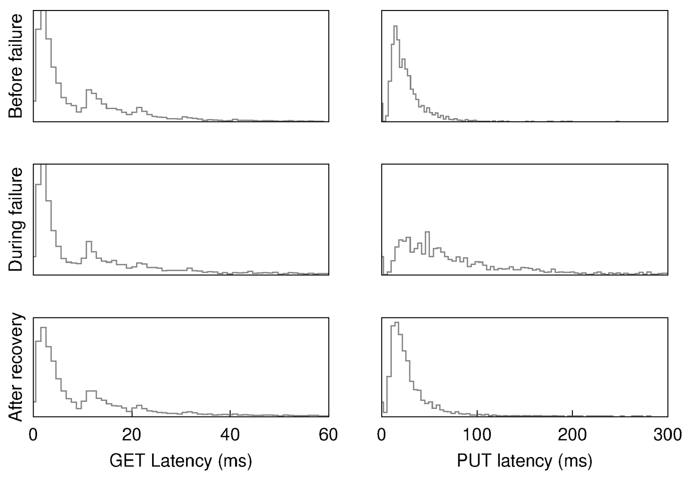


Fig. 13. Latency distributions: Histograms show the latency distributions before, during, and after brick failure. For both get and put requests, the relatively small shifts in distribution are due mainly to varying I/O requirements during the different periods of operation.

6.3.4 *Recovery From Hard Failures.* Thus far, we have only discussed transient failures. We conclude this section with a discussion on how we intend to handle hard failures, a next step in our future work. When data is not lost due to a hard failure, data can simply be copied to a replacement node. If instead, a disk fails, data must be reconstructed from data on the remaining bricks. In RAID, reconstruction [Chen et al. 1994] involves issuing low-priority reads to rebuild the lost data. Similarly, DStore can issue quorum reads to rebuild an up-to-date dataset. A second option that we are exploring involves first copying the entire dataset from one brick, then reading the timestamp tables from $RT - 1$ other bricks, and finally, issuing read requests for the values in the copied dataset that are stale. As with online repartitioning described later in Section 8, we can use quorums to take bricks offline while copying data and still maintain data availability. This technique allows us to recast reconstruction as failure plus recovery.

7. AGGRESSIVE FAILURE DETECTION

Two failure detection mechanisms can cause DStore to initiate brick recovery as described earlier in Section 5. First, Dlibs initiate recovery if a brick misses two consecutive RGID beaconing periods. This simple beaconing mechanism is usually sufficient to detect stopping failures like node crashes, or faults that can be mapped to stopping failures using fault-model enforcement, like certain types of memory corruption and bit flips [Nagaraja et al. 2002].

Whereas stopping failures are relatively easy to detect, fail-stutter is difficult to detect reliably because it is hard to determine if degradation is due to a faulty component or some other factor, like garbage collection or cache warming. To detect fail-stutter behavior and other anomalous behaviors that may be indicative of a current or pending failure, a brick periodically reports operating statistics which are compared to the operating statistics of other bricks to detect

Table II. Operating Statistics That Are Monitored to Detect Anomalies in Brick Behavior

Statistic	Description
CPU load	Load average as reported by `uptime`
Memory usage	Memory usage as reported by `free`
Queue length	Current GET, PUT, TS queue lengths
Success	GET, PUT, TS requests processed since last report
Dropped	GET, PUT, TS requests dropped since last report
Queue delay	Exponential moving average (EMA, $\alpha = .0002$) delay for each queue type
Latency	Request processing time EMA ($\alpha = .0002$) for each request type

deviant behavior. If a brick's operating statistic deviates by more than a certain deviation threshold, an anomaly is raised. If the number of anomalies for one brick exceeds a certain anomaly threshold, the brick is restarted. The set of operating statistics bricks report is listed in Table II.

We use two statistical methods for generating a model of "good behavior" and detecting anomalies. The first method is median absolute deviation, which compares one brick's current behavior with that of the majority of the system. This metric was chosen for its robustness to outliers in small populations, which is important for small DStore installations. The second method is the Tarzan algorithm [Keogh et al. 2002] for analyzing time series, which incorporates a brick's past behavior and compares it with that of the rest of the bricks. For every statistic of each brick, we keep an N-length history, or time-series, of the state and discretize it 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 an abnormally high or low frequency of some substring as compared to the other bricks, an anomaly is raised. We set $k = 3$, $N = 100$, deviation threshold to 0.5, and we vary the anomaly threshold.

We note that although such methods are potentially powerful tools for identifying deviant behavior, identifying the "best" algorithms to use is a *nongoal* of this work. Rather, our goal is to demonstrate that DStore's cheap recovery allows us to take advantage of these relatively simple, application-generic techniques. Even though anomalous conditions may be false positives that predict an eventual brick failure, the low cost of recovery enables us to act on some false positives without serious adverse effects. In fact, at times, it may be beneficial to reboot a brick even if an anomaly is transient.

7.1 Failure Detection Benchmarks

In this section, we show how DStore takes advantage of cheap recovery to resolve hard-to-catch slowdown failures using statistical anomaly detection. In these benchmarks, we run a six-brick DStore, and at $t = 5$ min gradually degrade one brick's throughput capacity by increasing the request-processing latency. We model degradation as lost CPU cycles, which is reasonable for slow downs due to memory leaks and virtual memory thrashing. We simulate degradation by performing extraneous floating point operations before processing each request, the number of which is increased every ten seconds.

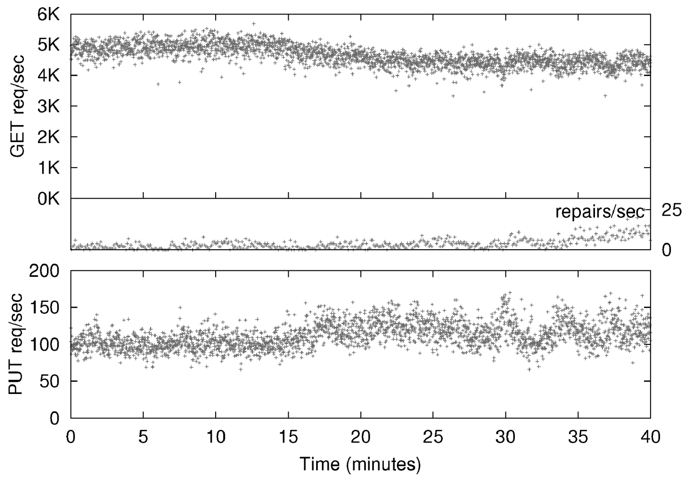


Fig. 14. Beacon-based detection: With beacons alone, DStore does not detect fail-stutter behavior.

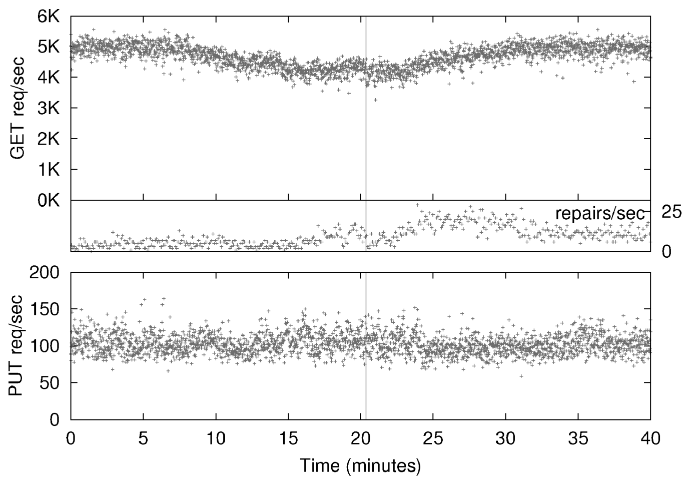


Fig. 15. Anomaly-based detection: Statistical techniques detect the degradation after a slight dip in system performance.

Figure 14 shows the result of injecting a performance degradation when only beacon-based detection is used. The degradation is not caught because the degraded brick continues sending beacons. However, overall performance degrades to the point where the brick performs almost no useful work. Figures 15 and 16 show the same degradation when statistical anomaly detection is applied with varying degrees of aggressiveness. Recall that the anomaly threshold corresponds to the number of brick statistics that must indicate deviant behavior before a brick is rebooted. By trial-and-error, we selected two thresholds, one that showed some system degradation before recovery was initiated, and another that caused spurious reboots in bricks in which we did not inject faults. With an anomaly threshold of 11, DStore detects the injected slowdown failure

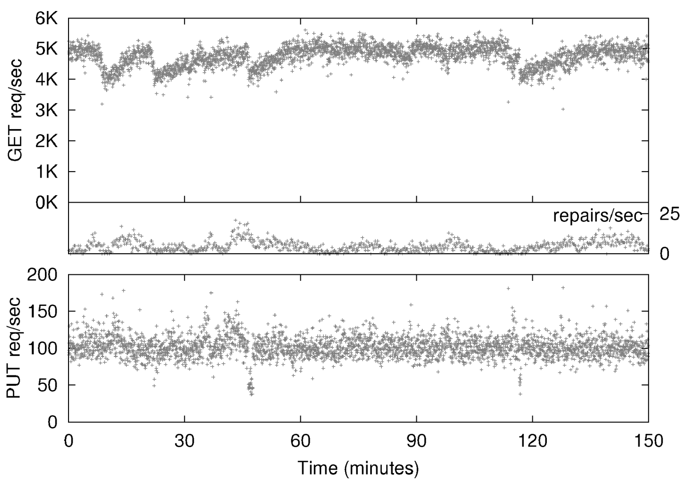


Fig. 16. Low cost of false positives: With aggressive thresholds, degradation is detected almost immediately; although false positives are also raised, the cost of acting on those false positives is small compared to prolonged degradation.

and reboots the node after a small, but noticeable, degradation in system performance. With a more aggressive policy where the threshold is set to 8, the degrading brick is caught more quickly, but DStore also reboots other bricks in which no faults were injected. However, the cost of acting on those false positives is small compared to prolonged degradation.

Although finding the best detection algorithm is not a goal of this work, what a system like DStore provides is an operating environment on which new algorithms can be tested. These benchmarks show that failure detection need not be so accurate if recovery is cheap. Having this type of latitude to make mistakes, allows an operator or the system itself to try new algorithms and learn from experience which ones are most effective for the given application and workload.

7.2 Remarks On Rolling Reboots

Along with beacon-based and statistical anomaly-based failure detection, cheap recovery enables a third, complementary failure handling mechanism—rolling reboots. By proactively rebooting bricks, software rejuvenation [Huang et al. 1995] can prevent failures that arise due to aging effects like memory leaks. Although rolling reboots catch a superset of the problems we can detect using beacons and statistical anomaly-detection, DStore detects and deals with failures more quickly than if we had to wait for the reboot to cycle to the affected brick. Thus, all three are complementary mechanisms.

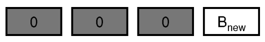
8. ZERO-DOWNTIME INCREMENTAL SCALING

Cheap recovery is the basis for our online repartitioning algorithm. Currently, repartitioning is initiated when an administrator issues a command to either add new bricks or remove existing ones. However, DStore is also well-suited for systems that monitor workload and predict resource utilization to automatically decide when to bring more resources online [Lassettre et al. 2003]. In either case, once the addition or removal process is initiated, DStore automatically decides which bricks to repartition and integrates new bricks or removes existing bricks.

8.1 Scaling Up By Splitting Bricks

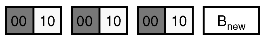
To increase the number of bricks used by the hash table, the current DStore implementation uses a linear hashing scheme for partitioning data [Litwin 1980]. The following repartitioning algorithm details how data is copied and bricks are split to implement this scheme. However, note that this algorithm can easily be modified to implement other hashing schemes.

(1) Discover RGID information: The new brick, B_{new}, constructs an RGID map by listening to RGID beacons.

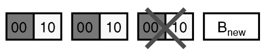


(2) Select the repartition brick: B_{new} listens for Dlib latency beacons and selects the most heavily-loaded brick, B_r.

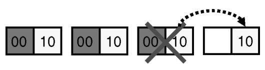
(3) Logically split B_r's replica group: A `SPLIT_RG` command is sent to each brick in B_r's replica group. Each brick in the replica group begins announcing two new RGIDs, one of which is obtained by prepending a 0 to the original RGID, the other, by prepending a 1. For example, if the original RGID is 0, a brick begins announcing both 00 and 10. Note that after this logical split, nothing changes in terms of which keys are sent to which bricks.



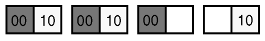
(4) Take B_r offline: An `OFFLINE` command is sent to B_r, causing B_r to prepend an "offline" flag to its RGID. When Dlibs see the change in RGID beacons, they stop sending requests to B_r. The result is similar to the failure of B_r, only Dlibs do not initiate recovery.



(5) Copy data: Data for half of B_r's keyspace is copied to B_{new}. In our example, the 10 partition is copied.



(6) Assign RGIDs and bring bricks online: The RGID for the copied partition, 10, is assigned to B_{new} and the complementary RGID, 00, is assigne,d to B_r. After both bricks begin announcing their new RGIDs, they are brought back online.



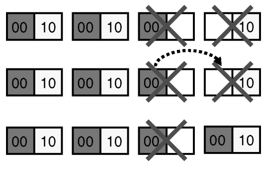
The repartitioning algorithm is both online and automatic. First, consider the online aspect. Taking B_r offline and bringing it back online has the same effect as if B_r had failed and recovered; the only difference is that upon "recovery," two physical bricks have taken B_r's place. Therefore, any updates that occur during repartitioning are executed by the online bricks with updates being propagated to the offline bricks via the read-repair mechanism. Since repartitioning has the same effect as a failure, when multiple bricks are added to the same replica group, they are integrated into the system one at a time. If a failure occurs while repartitioning and the number of online bricks falls below a majority, the repartitioning process is halted and B_r is brought back online. The resulting effect is no different than if the B_r had simply failed and recovered. Although it is possible to add bricks simultaneously, without taking bricks offline, by adjusting the read and write thresholds accordingly, we use this recovery-based mechanism because it makes the performance impact and time for adding new bricks more predictable.

Second, consider the automatic aspect of the algorithm. DStore's simple hash table API removes implicit data dependences, which enables our algorithm to select B_r based solely on load information, without worrying about splitting up data that is accessed together. The generalized form of the algorithm is to globally repartition the system by selecting the N-most loaded bricks and moving portions of the keyspace to the new brick as necessary. However, for simplicity, the current algorithm splits only the heaviest-loaded brick, approximated by the exponential-moving-average request latency with a smoothing constant $\alpha = 0.5$.

8.2 Scaling Down By Coalescing Bricks

Removing a brick is analogous to adding a new brick, and therefore, uses the same mechanisms of taking bricks offline, copying data, and bringing the remaining bricks back online. The basic idea is that before removing a brick, B_{remove}, its data must be coalesced with another brick, B_{keep}. The steps for scaling down are as follows:

(1) Select the bricks to coalesce: Identify the least-loaded brick B_{keep}, which will accept the load of the removed brick. Choose B_{remove} such that its RGID differs from B_{keep}'s RGID in only the most significant bit. To understand how this selection is made, recall that when splitting a brick, a high-order bit is added. Conversely, when coalescing bricks, the high order bit is removed to obtain B_{keep}'s new RGID.



(2) Take B_{remove} and B_{keep} offline:

(3) Copy data: Data in the partition that B_{remove} manages is copied to B_{keep}.

(4) Assign B_{keep}'s RGID, kill B_{remove}, and bring B_{keep} online: Before killing B_{remove} and bringing B_{keep} online, B_{keep} must first begin announcing both its original RGID and B_{remove}'s RGID.

When copying data from B_{remove} to B_{keep}, we must consider how the data layout affects the copy step. When a brick is split, in terms of correctness, B_{orig}'s entire data set can be copied to B_{new} even though only a portion of the data set is accessed. However, when two bricks are coalesced, only the data that B_{remove} is supposed to manage should be copied to B_{keep}.

We consider two file layouts shown in Figures 17 and 18: high-bit files and low-bit files. With high-bit files, the key's most significant bits determine the file

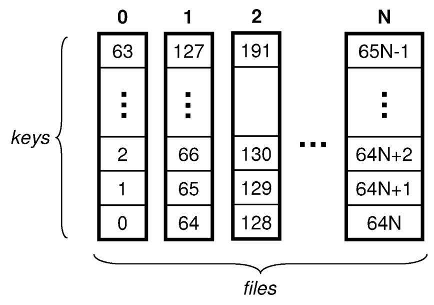


Fig. 17. High-bit files.

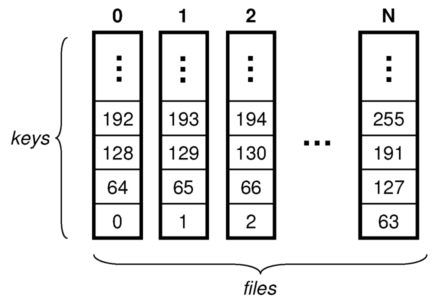


Fig. 18. Low-bit files.

name, and the least significant bits determine the index into the file. Since the current implementation of DStore partitions data based on the least significant bits, using high-bit files causes data from different partitions to be intermingled within each file. For example, data in the 0 partition (keys 0, 2, 4, etc.) resides on every file occupying every other entry. Therefore, when data is copied upon removal of a brick, every other entry must be copied from B_{remove} to B_{keep}. In contrast, when the least significant bits determine the file name as well as the partition, partitions occupy entire files. For example, the 0 partition is fully and exclusively stored in the even files making copying data simpler.

This same idea applies if, rather than relying on the file system, we stored data directly on disk blocks. By using a scheme in which data from only one partition is stored on any given disk block, copying can be done at the granularity of disk blocks rather than per key.

8.3 Scaling Benchmarks

The benchmarks in this section show that scaling, like recovery, has a predictably small impact on availability and performance. Before describing the benchmarks, we note that there are typically two dimensions along which a system can be scaled: storage- and load-capacity.

In our benchmarks, we do not add any data to the original data set. However, by taking advantage of the file layout scheme and the 4-K file block size, we are able to model both types of scaling. To scale in both storage- and load-capacity, we use high-bit files with 512-byte entries. With high-bit files, if the data set is split into N partitions, only every Nth entry in each file is relevant to a

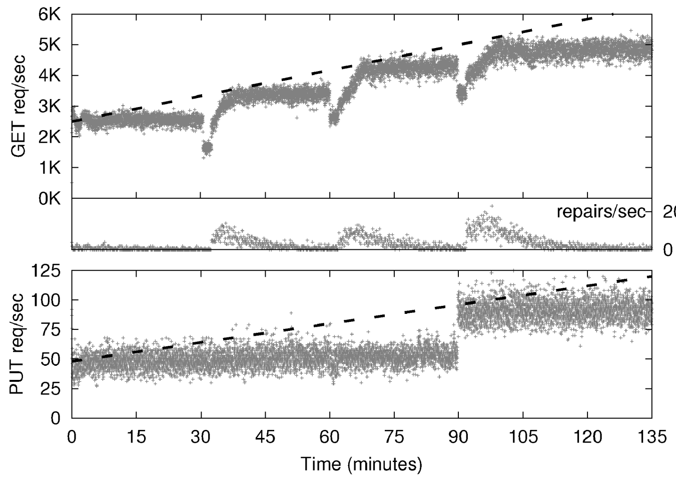


Fig. 19. Scaling in storage and load: Three new bricks are added to a three-brick DStore to double throughput without causing unavailability.

particular brick's partition. However, as long as $N < 8$, per-brick performance of disk and cache are not improved by repartitioning because an entire 4K-block has to be read on each access. In contrast, to measure performance scaling alone, we use low-bit files so that doubling the number of bricks halves the amount of data each brick stores. Note that in practice, DStore would likely be configured to use the low-bit file layout because it simplifies the data-copy step when scaling down, and it has better space utilization when scaling up.

We first show basic storage- and load-capacity scaling behavior by running a three-brick DStore and adding three new bricks, one every thirty minutes. Since recovery forms the basis for repartitioning, as Figure 19 shows, adding a brick looks very much like brick recovery. For each brick added, `get` throughput increases linearly, as denoted by the dotted line. However, `put` throughput does not increase until the entire replica group is repartitioned. Whereas `get` requests are load-balanced so that the repartitioned bricks receive more requests than the nonrepartitioned bricks, `put` requests are sent to the entire replica group, so throughput is limited by the nonpartitioned brick(s). Note that `put` throughput actually rises while the third brick is being repartitioned because the rate-limiting brick is removed, leaving the system with two replica groups, each with two bricks.

The second benchmark shows DStore's behavior as bricks are added and removed to scale load capacity while keeping storage capacity constant. We run a three-brick DStore, but instead, use a four-gigabyte dataset stored in low-bit files. A brick is added or removed once every thirty minutes, going from three bricks to six, and back down to three. Figure 20 shows that doubling the number of bricks more than doubles the `get` throughput. This is due to the fact that adding a brick halves the amount of data each brick handles, which increases the effective cache hit rate. Since writes are unaffected by the cache, `put` throughput increases linearly as it does when scaling in both load- and storage-capacity.

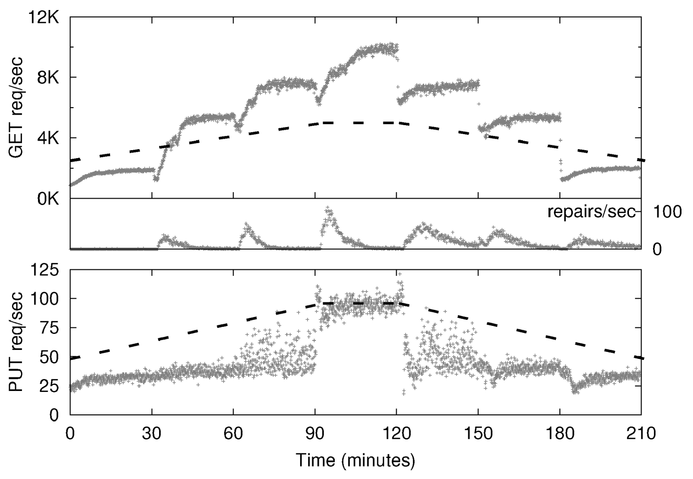


Fig. 20. Scaling in load: When scaling in load alone, throughput more than doubles because bricks split the dataset and achieve higher cache hit rates.

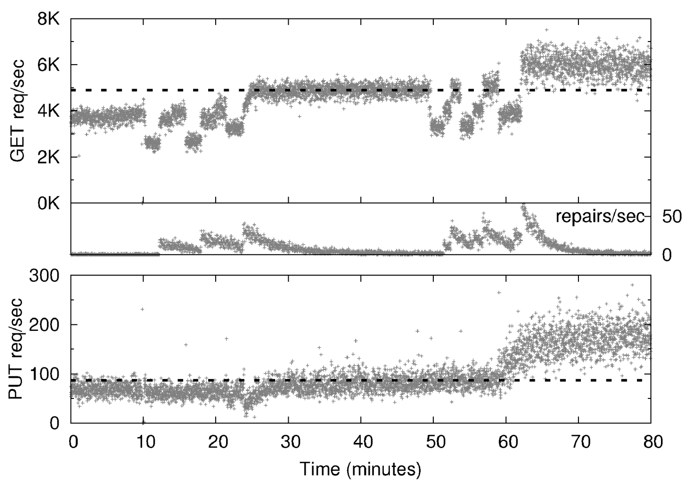


Fig. 21. Hot spot scaling: DStore repartitions hotspot bricks to achieve higher performance than a naive partitioning, which is shown by the dashed line.

The third benchmark shows that the latency-based brick-selection scheme handles hotspots by repartitioning the most heavily-loaded bricks. In Figure 21, we start with a six-brick DStore and two replica groups, $RGID = 0$ and $RGID = 1$. Rather than spreading requests evenly across the keyspace, clients request keys in a biased fashion, producing a data hotspot in keys ending in 00. Three bricks are added at $t = 10$ m to split the 0 partition. This addition brings the throughput to the same level as a naive partitioning in which the replica groups are split evenly across the entire keyspace while the 00 remains heavily overloaded (represented by the dashed line). Instead, when the second set of bricks is added at $t = 50$ m, DStore splits the 00 partition into 000 and 100 to achieve much higher performance.

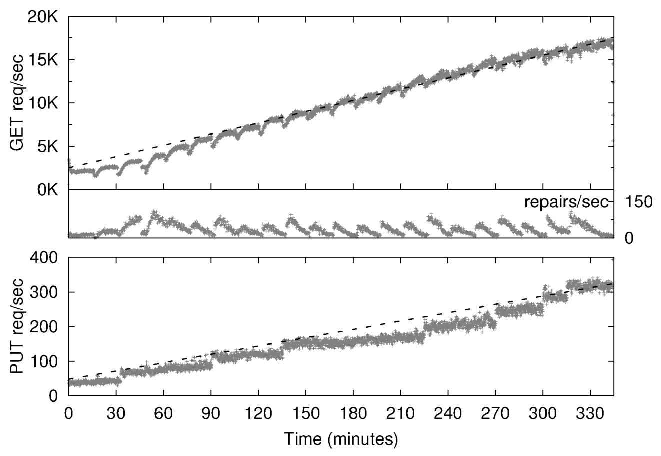


Fig. 22. Plug-and-play scaling: Repartitioning is used to scale DStore from 3 to 24 bricks in an automatic, fully online fashion.

In the final benchmark, we use online repartitioning to scale DStore up to 24 bricks, after which, we do not have enough nodes to scale the benchmark further. Figure 22 shows that DStore can be scaled linearly in a plug-and-play fashion with zero downtime and predictable performance.

Scaling is cheap because repartitioning is expressed in terms of failure and recovery, which are themselves cheap by design. Like with failure detection, having a low cost scaling mechanism means that load prediction need not be so accurate, which allows for experimentation with different algorithms. Furthermore, because bringing resources online takes a predictable amount of time and incurs a predictably small performance impact, DStore is amenable to the use of automatic resource provisioning algorithms developed for application servers [Lassettre et al. 2003].

9. DISCUSSION

In DStore, HTTP frontends served as a concrete example of a self-managing system. Having extracted principles aimed at developing low-cost mechanisms for failure handling and scaling, we now explore the more general question of what it means for a system to be self-managing. In this section, we use our experience with DStore to develop general design guidelines for building self-managing systems.

First, consider the three aspects of DStore's failure-handling shown in Figure 23. We start with an indicator of system health; bricks periodically beacon operating statistics such as memory usage and request latency. The second aspect is monitoring; statistical methods are applied to the operating statistics to detect when a brick is outside its "normal" operating window. Finally, there is a treatment; when a brick is flagged as anomalous, a reboot is triggered.

We can generalize the concept of indicators, monitoring, and treatments to include other dimensions of system health as shown in Figure 24. For example, monitoring overall system load can trigger the addition and removal of nodes as needed to meet current load demands. Further, monitoring disk failures can

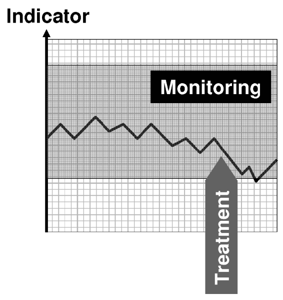


Fig. 23. Aspects of self-management.

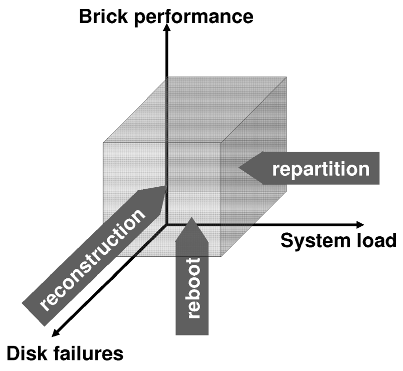


Fig. 24. System's operating envelope.

determine when disks are replaced and when data reconstruction is initiated. Combining these and other relevant indicators of system health produces an operating region within which the system's behavior is well understood. The resulting self-managing system comprises a set of monitors working together to detect when the system may be moving outside of its desired operating region and triggering the appropriate treatment when needed.

As is the case with recovery and scaling, low-cost treatments are critical for enabling simple monitoring policies which need not closely involve human operators. For example, in the area of human health care, treatments consisting of low-risk, low-side effect drugs encourage early prevention of more serious future problems. However, when surgery is the sole treatment option, doctors tend to be more conservative in diagnosing and prescribing treatment, sometimes waiting until it is too late to fully resolve the problem. Analogously, when management and recovery "treatments" are low-cost adjustments to the system behavior rather than heavyweight fixes, it is less critical that the monitoring system never make a mistake. This not only allows the monitoring to attempt early treatment, it also provides a testbed on which newly-developed treatments and monitoring techniques can be tried with minimal system impact.

10. RELATED WORK

The idea of achieving ease-of-manageability via cheap recovery has also been explored in the context of microreboots for application servers [Candea et al.

2004] and SSM [Ling et al. 2004], a RAM-only CHT for session state that ensures consistency by storing metadata about where state is replicated in a cookie on the user's Web client. SSM's design relies on the fact that session state is accessed by a single user and has a bounded lifetime, typically minutes or hours.

Designing state stores for single-key lookup data is well established. Providing motivation for our work, Distributed Data Structures [Gribble et al. 2000] is a scalable, high-performance CHT, which uses ROWA and two-phase commit. Berkeley DB [Olson et al. 1999], which can serve as underlying storage for DStore bricks, is a single-node hash table that can be replicated using a primary-secondary scheme, but does not provide a single-system image across a cluster.

Brick-based disk storage systems are low-cost alternatives to high-end RAID [Chen et al. 1994] disk arrays. Federated Array of Bricks [Frolund et al. 2003] uses quorums and a nonlocking two-phase protocol to ensure linearizability. RepStore [Zhang et al. 2004] optimizes for storage overhead through selective use of erasure-coding over replication. In contrast to distributed disk and file systems [Ganger et al. 2003; Thekkath et al. 1997] with similar self-management goals, DStore exploits specific workload characteristics for consistency management and exposes a higher-level interface with guarantees on variable-sized elements. Similarly, the Google File System [Ghemawat et al. 2003] is designed for its large-file, append-mostly workload to achieve scalability and manageability.

Consistency is traded for performance and availability [Brewer 2001; Yu and Vahdat 2000], and quorums [Gifford 1979] provide availability under network partitions [Davidson et al. 1985] and Byzantine faults [Malkhi and Reiter 1998; Pierce 2000]. However, DStore uses quorums and trades consistency for extremely simple persistent state management. Like DStore, Coda [Kistler and Satyanarayanan 1992] tolerates replica inconsistency, but in a nontransparent fashion. Bayou's [Terry et al. 1994] *Monotonic Reads* and *Read-Your-Writes* provide guarantees similar to those provided by DStore's read-repair and write-in-progress cookie mechanisms. However, Bayou makes guarantees for a single user session, not across users. The Porcupine Mail Server [Saito et al. 1999] has self-management goals, but its *eventual consistency* guarantees allow users to see data waver between old and new values before replicas eventually become consistent.

Finally, DStore owes its statistical failure detection to Pinpoint [Chen et al. 2002], a comprehensive, ongoing investigation of anomaly-based failure detection.

11. CONCLUSION

We combined two techniques that reduce coupling—quorums to tolerate replica inconsistency, and single-phase operations to avoid locking—to make reboot-based recovery fast and nonintrusive in a persistent cluster hash table. One consequence of this cheap recovery is that we successfully applied aggressive statistical anomaly-based failure detection to automatically detect and recover from both fail-stop and fail-stutter transients. Furthermore, we used the same recovery mechanism to solve the distinct problem of incremental scaling without service interruption. The net result is a state store that can be managed with the same types of simple recovery and scaling mechanisms as used for stateless frontends.

Taking a larger view, we believe low-cost mechanisms like cheap recovery and zero-downtime scaling are important components of self-managing systems. When such mechanisms are predictably fast and have predictably small impact on system availability and performance, the line between "normal" and "recovery" operation becomes blurred. Thus, rather than relying on the monitoring system to be flawless, the resulting system can stay within its desired operating region by continuously self-adjusting, a key property of self-managing systems.

APPENDIX

A. ALGORITHM PSEUDO-CODE

```
put(key, value, ts)
if (ts == NULL)
   ts = generate_timestamp()
foreach b in <replica group bricks>
   send(b, WRITE, key, val, ts)
for (c = 0 to WT-1)
   err = receive(W_RESP, W_TIMEOUT)
   if (err == TIMEOUT)
      return TIMEOUT

write(key, value, ts)
current_ts = read_ts(key)
if (ts < current_ts)
   return TIMESTAMP_ERROR
else
   WRITE(key, value, ts)
   return SUCCESS
```

```
get(key)
rset[0...RT-1] = choose_bricks(RT)
send(rset[0], READ_VAL, key)
for (i = 1 to RT-1)
   send(rset[i], READ_TS, key)
for (i = 1 to RT-1)
   err = receive(TS_RESP, ts[i], R_TIMEOUT)
   if (err == TIMEOUT)
      return TIMEOUT
err = receive(VAL_RESP, ts[0], value, R_TIMEOUT)
if (err = TIMEOUT)
   return TIMEOUT
return check(key, value, ts[], rset[])

check(key, value, ts[], rset[])
{maxts, index} = findmax(ts[])
timeouts = 0
if (ts[0] != maxts)
   send(rset[index], READ_VAL, key)
   err = receive(VAL_RESP, maxts, value, R_TIMEOUT)
   if (err == TIMEOUT)
      return TIMEOUT
for (i = 0 to RT-1)
   if (ts[i] != maxts)
      send(rset[i], WRITE, key, value, maxts)
      err = receive(W_RESP, W_TIMEOUT)
      if (err != TIMEOUT)
         timeouts++
if (RT - timeouts < WT)
   put(key, value, maxts)
return value
```

REFERENCES

Arpaci-Dusseau, R. H. and Arpaci-Dusseau, A. C. 2001. Fail-stutter fault tolerance. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*. Elmau/Oberbayern, Germany. 33–38.

Brewer, E. 2001. Lessons from giant-scale services. *IEEE Internet Comput. 5*, 4 (July), 46–55.

Brewer, E. 2004. Combining systems and databases: A search engine retrospective. In *Readings in Database Systems*. 4th M. Stonebreaker, Ed. MIT Press, Cambridge, MA.

Candea, G. and Fox, A. 2003. Crash-only software. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*. Lihue, HI.

Candea, G., Kawamoto, S., Fujiki, Y., and Fox, A. 2004. A microrebootable system—design, implementation, and evaluation. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*. San Francisco, CA.

Chen, M., Kiciman, E., Fratkin, E., Brewer, E., and Fox, A. 2002. Pinpoint: Problem determination in large, dynamic, internet services. In *Proceedings of the International Conference on Dependable Systems and Networks*. Washington, DC, 595–604.

Chen, P. M., Lee, E. L., Gibson, G. A., Katz, R. H., and Patterson, D. A. 1994. RAID: High-performance, reliable secondary storage. *ACM Comput. Surv. 26*, 2 (June), 145–185.

Davidson, S. B., Garcia-Molina, H., and Skeen, D. 1985. Consistency in partitioned networks. *ACM Comput. Surv. 17*, 3.

Duvur, S. 2004. Sun Microsystems. Personal communication.

Fox, A., Gribble, S. D., Chawathe, Y., Brewer, E. A., and Gauthier, P. 1997. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*. Saint-Malo, France, 78–91.

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

Ganger, G. R., Strunk, J. D., and Klosterman, A. J. 2003. Self-* storage: Brick-based storage with automated administration. Carnegie Mellon University Tech. Rep. CMU-CS-03-178.

Garg, S., Moorsel, A. V., Vaidyanathan, K., and Trivedi, K. S. 1998. A methodology for detection and estimation of software aging. In *Proceedings of the 9th International Symposium on Software Reliability Engineering*. Paderborn, Germany, 282–292.

Ghemawat, S., Gobioff, H., and Leung, S.-T. 2003. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*. Bolton Landing, NY.

Gibbs, W. W. 2002. Autonomic Computing. *Scientific American*.

Gifford, D. K. 1979. Weighted voting for replicated data. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles*. Pacific Grove, CA.

Gray, J. 2003. A conversation with Jim Gray. *ACM Queue 1*, 4 (June).

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

Gribble, S. D., Welsh, M., von Behren, R., Brewer, E. A., Culler, D., Borisov, N., Czerwinski, S., Gummadi, R., Hill, J., Joseph, A., Katz, R., Mao, Z., Ross, S., and Zhao., B. 2001. The ninja architecture for robust internet-scale systems and service. *Spec. Issu. Compt. Netw. Pervas. Comput. 35*.

Grossfeld, L. 2003. EBay system architecture. Personal Communication.

Huang, Y., Kintala, C. M. R., Kolettis, N., and Fulton, N. D. 1995. Software rejuvenation: Analysis, module and applications. In *International Symposium on Fault-Tolerant Computing*. Pasadena, CA, 381–390.

IDC. 2003. HP Utility Data Center: Enabling enhanced datacenter agility.

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

Kistler, J. J. and Satyanarayanan, M. 1992. Disconnected Operation in the Coda file system. *ACM Trans. Comput. Syst. 10*, 1 (Feb), 3–25.

Lamport, L. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs In *IEEE Trans. Comput. 28*, 9 (Sept.), 690–691.

LASSETTRE, E., COLEMAN, D., DIAO, Y., FROELICH, S., HELLERSTEIN, J., HSIUNG, L., MUMMERT, T., RAGHAVACHARI, M., PARKER, G., RUSSELL, L., SURENDRA, M., TSENG, V., WADIA, N., AND YE, P. 2003. Dynamic Surge Protection: An approach to handling unexpected workload surges with resource actions that have lead times. In *Proceedings of the 1st Workshop on Algorithms and Architectures for Self-Managing Systems*. San Diego, CA.

LING, B. C., KICIMAN, E., AND FOX, A. 2004. Session state: Beyond soft state. In *Proceedings of the 1st USENIX/ACM Symposium on Networked Systems Design and Implementation*. San Francisco, CA.

LISKOV, B., GHEMAWAT, S., GRUBER, R., JOHNSON, P., SHRIRA, L., AND WILLIAMS, M. 1991. Replication in the Harp file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*. Pacific Grove, CA, 226–238.

LITWIN, W. 1980. Linear hashing: A new tool for file and table addressing. In *Proceedings of the 6th Conference on Very Large Databases*. Montreal, Canada, 212–223.

MALKHI, D. AND REITER, M. K. 1998. Secure and scalable replication in phalanx. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*. Purdue University, West Lafayette, IN.

MESSINGER, A. 2004. BEA Systems. Personal communication.

MICROSOFT CORPORATION. 2004. Microsoft Dynamic Systems Initiative Overview.

NAGARAJA, K., BIANCHINI, R., MARTIN, R. P., AND NGUYEN, T. D. 2002. Using fault model enforcement to improve availability. In *Proceedings of the 2nd Workshop on Evaluating and Architecting System Dependability*. San Jose, CA, 37–46.

OLSON, M., BOSTIC, K., AND SELZER, M. 1999. Berkeley DB. In *Proceedings of the 1999 USENIX Annual Technical Conference*. Monterey, CA, 183–192.

PAL, A. 2003. Yahoo! User Preferences Database. Personal Communication.

PARIS, J.-F. 1986. Voting with witnesses: A consistency scheme for replicated files. In *Proceedings of the 6th International Conference on Distributed Computing Systems*. Cambridge, MA, 606–612.

PASS CONSULTING GROUP. 2001. IBM DB2 UDB EEE v.7.2 and Oracle9i: A technical comparison.

PIERCE, E. T. 2000. Self-adjusting quorum systems for byzantine fault tolerance. Ph.D. thesis, University of Texas, Austin, TX.

ROWE, A. 2002. Network Appliance Filer. Personal Communication.

SAITO, Y., BERSHAD, B. N., AND LEVY, H. M. 1999. Manageability, availability and performance in porcupine: A highly scalable, cluster-based mail service. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*. Charleston, SC, 1–15.

SCHEURICH, C. AND DUBOIS, M. 1987. Correct memory operation of cache-based multiprocessors In *Proceedings of the 14th International Symposium on Computer Architecture*, Pittsburg, PA, 234–243.

SUN MICROSYSTEMS. 2002. N1—Introducting Just In Time Computing.

TERRY, D. B., DEMERS, A. J., PETERSEN, K., SPREITZER, M. J., THEIMER, M. M., AND WELCH, B. B. 1994. Session guarantees for weakly consistent replicated data. In *International Conference on Parallel and Distributed Information Systems*. Austin, TX, 140–149.

THEKKATH, C. A., MANN, T., AND LEE, E. K. 1997. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*. Saint-Malo, France.

THOMAS, R. 1979. A majority consensus approach to concurrency control for multiple copy database. *ACM Trans. Datab. Syst. 4*, 2(June), 180–209.

TRIVEDI, K. S. 2002. *Probability and Statistics with Reliability, Queueing and Computer Science Applications*. John Wiley & Sons, New York, NY.

VERMEULEN, A. 2003. Amazon.com. Personal Communication.

WOOL, A. 1998. Quorum systems in replicated datases: Science or fiction? *IEEE Data Engin. Bull. 21*, 4, 3–11.

YU, H. AND VAHDAT, A. 2000. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*. 305–318.

ZHANG, Z., LIN, S., LIAN, Q., AND JIN, C. 2004. RepStore: A self-managing and self-tuning storage backend with smart bricks. In *Proceedings of the 1st International Conference on Autonomic Computing (ICAC-04)*. New York, NY.

Received August 2004; revised September 2004; accepted September 2004