Chain Replication for Supporting High Throughput and Availability

Robbert van Renesse
rvr@cs.cornell.edu

Fred B. Schneider
fbs@cs.cornell.edu

FAST Search & Transfer ASA
Tromsø, Norway
and
Department of Computer Science
Cornell University
Ithaca, New York 14853

Abstract

Chain replication is a new approach to coordinating clusters of fail-stop storage servers. The approach is intended for supporting large-scale storage services that exhibit high throughput and availability without sacrificing strong consistency guarantees. Besides outlining the chain replication protocols themselves, simulation experiments explore the performance characteristics of a prototype implementation. Throughput, availability, and several object-placement strategies (including schemes based on distributed hash table routing) are discussed.

1 Introduction

A *storage system* typically implements operations so that clients can store, retrieve, and/or change data. File systems and database systems are perhaps the best known examples. With a file system, operations (read and write) access a single file and are idempotent; with a database system, operations (transactions) may each access multiple objects and are serializable.

This paper is concerned with storage systems that sit somewhere between file systems and database systems. In particular, we are concerned with storage systems, henceforth called *storage services*, that

- store *objects* (of an unspecified nature),
- support *query* operations to return a value derived from a single object, and
- support *update* operations to atomically change the state of a single object according to some pre-programmed, possibly non-deterministic, computation involving the prior state of that object.

A file system write is thus a special case of our storage service update which, in turn, is a special case of a database transaction.

Increasingly, we see on-line vendors (like Amazon.com), search engines (like Google's and FAST's), and a host of other information-intensive services provide value by connecting large-scale storage systems to networks. A storage service is the appropriate compromise for such applications, when a database system would be too expensive and a file system lacks rich enough semantics.

One challenge when building a large-scale storage service is maintaining high availability and high throughput despite failures and concomitant changes to the storage service's configuration, as faulty components are detected and replaced.

Consistency guarantees also can be crucial. But even when they are not, the construction of an application that fronts a storage service is often simplified given *strong consistency guarantees*, which assert that (i) operations to query and update individual objects are executed in some sequential order and (ii) the effects of update operations are necessarily reflected in results returned by subsequent query operations.

Strong consistency guarantees are often thought to be in tension with achieving high throughput and high availability. So system designers, reluctant to sacrifice system throughput or availability, regularly decline to support strong consistency guarantees. The Google File System (GFS) illustrates this thinking [11]. In fact, strong consistency guarantees in a large-scale storage service are not incompatible with high throughput and availability. And the new *chain replication* approach to coordinating fail-stop servers, which is the subject of this paper, simultaneously supports high throughput, availability, and strong consistency.

We proceed as follows. The interface to a generic storage service is specified in §2. In §3, we explain how query and update operations are implemented using chain replication. Chain replication can be viewed as an instance of the primary/backup approach, so §4 compares them. Then, §5 summarizes experiments to analyze throughput and availability using our prototype implementation of chain replication and a simulated network. Some of these simulations compare chain replication with storage systems (like CFS [7] and PAST [19]) based on distributed hash table (DHT) routing; other simulations reveal surprising behaviors when a system employing chain replication recovers from server failures. Chain replication is compared in §6 to other work on scalable storage systems, trading consistency for availability, and replica placement. Concluding remarks appear in §7, followed by endnotes.

2 A Storage Service Interface

Clients of a storage service issue *requests* for query and update operations. While it would be possible to ensure that each request reaching the storage service is guaranteed to be performed, the end-to-end argument [20] suggests there is little point in doing so. Clients are better off if the storage service simply generates a reply for each request it receives and completes, because this allows lost requests and lost replies to be handled as well: a client re-issues a request if too much time has elapsed without receiving a reply.

- The reply for query($objId$, $opts$) is derived from the value of object *objId*; options *opts* characterizes what parts of *objId* are returned. The value of *objId* remains unchanged.

- The reply for update($objId$, $newVal$, $opts$) depends on options *opts* and, in the general case, can be a value V produced in some nondeterministic pre-programmed way involving the current value of *objId* and/or value *newVal*; V then becomes the new value of *objId*.[1]

Query operations are idempotent, but update operations need not be. A client that re-issues a non-idempotent update request must therefore take precautions to ensure the update has not already been performed. The client might, for example, first issue a query to determine whether the current value of the object already reflects the update.

```
State is:
  $Hist_{objID}$ : update request sequence
  $Pending_{objID}$ : request set

Transitions are:
  T1: Client request $r$ arrives:
        $Pending_{objID} := Pending_{objID} \cup \{r\}$
  T2: Client request $r \in Pending_{objID}$ ignored:
        $Pending_{objID} := Pending_{objID} - \{r\}$
  T3: Client request $r \in Pending_{objID}$ processed:
        $Pending_{objID} := Pending_{objID} - \{r\}$
        if $r$ = query($objId, opts$) then
          reply according options $opts$ based
            on $Hist_{objID}$
        else if $r$ = update($objId, newVal, opts$) then
          $Hist_{objID} := Hist_{objID} \cdot r$
          reply according options $opts$ based
            on $Hist_{objID}$
```

Figure 1: Client's View of an Object.

A client request that is lost before reaching the storage service is indistinguishable to that client from one that is ignored by the storage service. This means that clients would not be exposed to a new failure mode when a storage server exhibits transient outages during which client requests are ignored. Of course, acceptable client performance likely would depend on limiting the frequency and duration of transient outages.

With chain replication, the duration of each transient outage is far shorter than the time required to remove a faulty host or to add a new host. So, client request processing proceeds with minimal disruption in the face of failure, recovery, and other reconfiguration. Most other replica-management protocols either block some operations or sacrifice consistency guarantees following failures and during reconfigurations.

We specify the functionality of our storage service by giving the client view of an object's state and of that object's state transitions in response to query and update requests. Figure 1 uses pseudo-code to give such a specification for an object *objID*.

The figure defines the state of *objID* in terms of two variables: the sequence[2] $Hist_{objID}$ of updates that have been performed on *objID* and a set $Pending_{objID}$ of unprocessed requests.

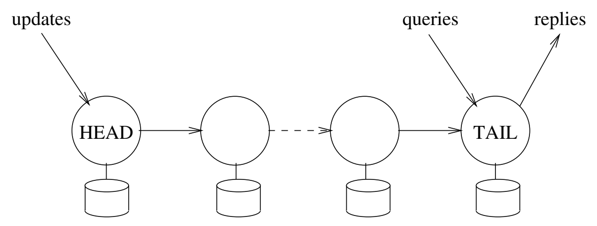


Figure 2: A chain.

Then, the figure lists possible state transitions. Transition T1 asserts that an arriving client request is added to $Pending_{objID}$. That some pending requests are ignored is specified by transition T2—this transition is presumably not taken too frequently. Transition T3 gives a high-level view of request processing: the request r is first removed from $Pending_{objID}$; query then causes a suitable reply to be produced whereas update also appends r (denoted by $\cdot$) to $Hist_{objID}$.[3]

3 Chain Replication Protocol

Servers are assumed to be fail-stop [21]:

- each server halts in response to a failure rather than making erroneous state transitions, and
- a server's halted state can be detected by the environment.

With an object replicated on t servers, as many as $t-1$ of the servers can fail without compromising the object's availability. The object's availability is thus increased to the probability that all servers hosting that object have failed; simulations in §5.4 explore this probability for typical storage systems. Henceforth, we assume that at most $t-1$ of the servers replicating an object fail concurrently.

In chain replication, the servers replicating a given object *objID* are linearly ordered to form a *chain*. (See Figure 2.) The first server in the chain is called the *head*, the last server is called the *tail*, and request processing is implemented by the servers roughly as follows:

Reply Generation. The reply for every request is generated and sent by the tail.

Query Processing. Each query request is directed to the tail of the chain and processed there atomically using the replica of *objID* stored at the tail.

Update Processing. Each update request is directed to the head of the chain. The request is processed there atomically using replica of *objID* at the head, then state changes are forwarded along a reliable FIFO link to the next element of the chain (where it is handled and forwarded), and so on until the request is handled by the tail.

Strong consistency thus follows because query requests and update requests are all processed serially at a single server (the tail).

Processing a query request involves only a single server, and that means query is a relatively cheap operation. But when an update request is processed, computation done at $t-1$ of the t servers does not contribute to producing the reply and, arguably, is redundant. The redundant servers do increase the fault-tolerance, though.

Note that some redundant computation associated with the $t-1$ servers is avoided in chain replication because the new value is computed once by the head and then forwarded down the chain, so each replica has only to perform a write. This forwarding of state changes also means update can be a non-deterministic operation—the non-deterministic choice is made once, by the head.

3.1 Protocol Details

Clients do not directly read or write variables $Hist_{objID}$ and $Pending_{objID}$ of Figure 1, so we are free to implement them in any way that is convenient. When chain replication is used to implement the specification of Figure 1:

- $Hist_{objID}$ is defined to be $Hist^{T}_{objID}$, the value of $Hist_{objID}$ stored by tail T of the chain, and
- $Pending_{objID}$ is defined to be the set of client requests received by any server in the chain and not yet processed by the tail.

The chain replication protocols for query processing and update processing are then shown to satisfy the specification of Figure 1 by demonstrating how each state transition made by any server in the chain is equivalent either to a no-op or to allowed transitions T1, T2, or T3.

Given the descriptions above for how $Hist_{objID}$ and $Pending_{objID}$ are implemented by a chain (and assuming for the moment that failures do not occur), we observe that the only server transitions affecting $Hist_{objID}$ and $Pending_{objID}$ are: (i) a server in the chain receiving a request from a client (which affects $Pending_{objID}$), and (ii) the tail processing a client request (which affects $Hist_{objID}$). Since other server transitions are equivalent to no-ops, it suffices to show that transitions (i) and (ii) are consistent with T1 through T3.

Client Request Arrives at Chain. Clients send requests to either the head (update) or the tail (query). Receipt of a request r by either adds r to the set of requests received by a server but not yet processed by the tail. Thus, receipt of r by either adds r to $Pending_{objID}$ (as defined above for a chain), and this is consistent with T1.

Request Processed by Tail. Execution causes the request to be removed from the set of requests received by any replica that have not yet been processed by the tail, and therefore it deletes the request from $Pending_{objID}$ (as defined above for a chain)—the first step of T3. Moreover, the processing of that request by tail T uses replica $Hist^T_{objID}$ which, as defined above, implements $Hist_{objID}$—and this is exactly what the remaining steps of T3 specify.

Coping with Server Failures

In response to detecting the failure of a server that is part of a chain (and, by the fail-stop assumption, all such failures are detected), the chain is reconfigured to eliminate the failed server. For this purpose, we employ a service, called the *master*, that

- detects failures of servers,
- informs each server in the chain of its new predecessor or new successor in the new chain obtained by deleting the failed server,
- informs clients which server is the head and which is the tail of the chain.

In what follows, we assume the master is a single process that never fails. This simplifies the exposition but is not a realistic assumption; our prototype implementation of chain replication actually replicates a master process on multiple hosts, using Paxos [16] to coordinate those replicas so they behave in aggregate like a single process that does not fail.

The master distinguishes three cases: (i) failure of the head, (ii) failure of the tail, and (iii) failure of some other server in the chain. The handling of each, however, depends on the following insight about how updates are propagated in a chain.

Let the server at the head of the chain be labeled H, the next server be labeled $H+1$, *etc.*, through the tail, which is given label T. Define

$$Hist^i_{objID} \preceq Hist^j_{objID}$$

to hold if sequence[4] of requests $Hist^i_{objID}$ at the server with label i is a prefix of sequence $Hist^j_{objID}$ at the server with label j. Because updates are sent between elements of a chain over reliable FIFO links, the sequence of updates received by each server is a prefix of those received by its successor. So we have:

Update Propagation Invariant. For servers labeled i and j such that $i \leq j$ holds (i.e., i is a predecessor of j in the chain) then:

$$Hist^j_{objID} \preceq Hist^i_{objID}.$$

Failure of the Head. This case is handled by the master removing H from the chain and making the successor to H the new head of the chain. Such a successor must exist if our assumption holds that at most $t-1$ servers are faulty.

Changing the chain by deleting H is a transition and, as such, must be shown to be either a no-op or consistent with T1, T2, and/or T3 of Figure 1. This is easily done. Altering the set of servers in the chain could change the contents of $Pending_{objID}$—recall, $Pending_{objID}$ is defined as the set of requests received by any server in the chain and not yet processed by the tail, so deleting server H from the chain has the effect of removing from $Pending_{objID}$ those requests received by H but not yet forwarded to a successor. Removing a request from $Pending_{objID}$ is consistent with transition T2, so deleting H from the chain is consistent with the specification in Figure 1.

Failure of the Tail. This case is handled by removing tail T from the chain and making predecessor T^- of T the new tail of the chain. As before, such a predecessor must exist given our assumption that at most $t-1$ server replicas are faulty.

This change to the chain alters the values of both $Pending_{objID}$ and $Hist_{objID}$, but does so in a manner consistent with repeated T3 transitions: $Pending_{objID}$ decreases in size because $Hist^T_{objID} \preceq Hist^{T^-}_{objID}$ (due to the Update Propagation Invariant, since $T^- < T$ holds), so changing the tail from T to T^- potentially increases the set of requests completed by the tail which, by definition, decreases the set of requests in $Pending_{objID}$. Moreover, as required by T3, those update requests completed by T^- but not completed by T do now appear in $Hist_{objID}$ because with T^- now the tail, $Hist_{objID}$ is defined as $Hist^{T^-}_{objID}$.

Failure of Other Servers. Failure of a server S internal to the chain is handled by deleting S from the chain. The master first informs S's successor S^+ of the new chain configuration and then informs S's predecessor S^-. This, however, could cause the Update Propagation Invariant to be invalidated unless some means is employed to ensure update requests that S received before failing will still be forwarded along the chain (since those update requests already do appear in $Hist^i_{objID}$ for any predecessor i of S). The obvious candidate to perform this forwarding is S^-, but some bookkeeping and coordination are now required.

Let U be a set of requests and let $<_U$ be a total ordering on requests in that set. Define a request sequence $\overline{r}$ to be *consistent* with $(U, <_U)$ if (i) all requests in $\overline{r}$ appear in U and (ii) requests are arranged in $\overline{r}$ in ascending order according to $<_U$. Finally, for request sequences $\overline{r}$ and $\overline{r'}$ consistent with $(U, <_U)$, define $\overline{r} \oplus \overline{r'}$ to be a sequence of all requests appearing in $\overline{r}$ or in $\overline{r'}$ such that $\overline{r} \oplus \overline{r'}$ is consistent with $(U, <_U)$ (and therefore requests in sequence $\overline{r} \oplus \overline{r'}$ are ordered according to $<_U$).

The Update Propagation Invariant is preserved by requiring that the first thing a replica S^- connecting to a new successor S^+ does is: send to S^+ (using the FIFO link that connects them) those requests in $Hist^{S^-}_{objID}$ that might not have reached S^+; only after those have been sent may S^- process and forward requests that it receives subsequent to assuming its new chain position.

To this end, each server i maintains a list $Sent_i$ of update requests that i has forwarded to some successor but that might not have been processed by the tail. The rules for adding and deleting elements on this list are straightforward: Whenever server i forwards an update request r to its successor, server i also appends r to $Sent_i$. The tail sends an acknowledgement $ack(r)$ to its predecessor when it completes the processing of update request r. And upon receipt $ack(r)$, a server i deletes r from $Sent_i$ and forwards $ack(r)$ to its predecessor.

A request received by the tail must have been received by all of its predecessors in the chain, so we can conclude:

Inprocess Requests Invariant. If $i \leq j$ then

$$Hist^i_{objID} = Hist^j_{objID} \oplus Sent_i.$$

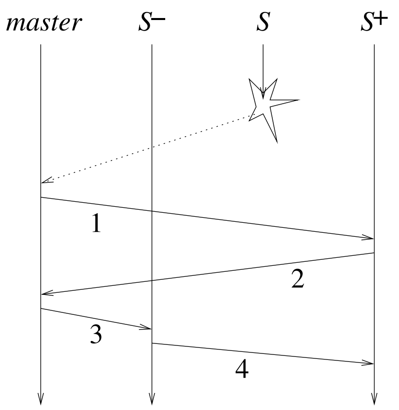


Figure 3: Space-time diagram for deletion of internal replica.

Thus, the Update Propagation Invariant will be maintained if S^-, upon receiving notification from the master that S^+ is its new successor, first forwards the sequence of requests in $Sent_{S^-}$ to S^+. Moreover, there is no need for S^- to forward the prefix of $Sent_{S^-}$ that already appears in $Hist^{S^+}_{objID}$.

The protocol whose execution is depicted in Figure 3 embodies this approach (including the optimization of not sending more of the prefix than necessary). Message 1 informs S^+ of its new role; message 2 acknowledges and informs the master what is the sequence number sn of the last update request S^+ has received; message 3 informs S^- of its new role and of sn so S^- can compute the suffix of $Sent_{S^-}$ to send to S^+; and message 4 carries that suffix.

Extending a Chain. Failed servers are removed from chains. But shorter chains tolerate fewer failures, and object availability ultimately could be compromised if ever there are too many server failures. The solution is to add new servers when chains get short. Provided the rate at which servers fail is not too high and adding a new server does not take too long, then chain length can be kept close to the desired t servers (so $t-1$ further failures are needed to compromise object availability).

A new server could, in theory, be added anywhere in a chain. In practice, adding a server T^+ to the very end of a chain seems simplist. For a tail T^+, the value of $Sent_{T^+}$ is always the empty list, so initializing $Sent_{T^+}$ is trivial. All that remains is to initialize local object replica $Hist^{T^+}_{objID}$ in a way that satisfies the Update Propagation Invariant.

The initialization of $Hist^{T^+}_{objID}$ can be accomplished by having the chain's current tail T forward the object replica $Hist^{T}_{objID}$ it stores to T^{+}. The forwarding (which may take some time if the object is large) can be concurrent with T's processing query requests from clients and processing updates from its predecessor, provided each update is also appended to $Sent_T$. Since $Hist^{T^{+}}_{objID} \preceq Hist^{T}_{objID}$ holds throughout this forwarding, Update Propagation Invariant holds. Therefore, once

$$Hist^{T}_{objID} = Hist^{T^{+}}_{objID} \oplus Sent_T$$

holds, Inprocess Requests Invariant is established and T^{+} can begin serving as the chain's tail:

- T is notified that it no longer is the tail. T is thereafter free to discard query requests it receives from clients, but a more sensible policy is for T to forward such requests to new tail T^{+}.
- Requests in $Sent_T$ are sent (in sequence) to T^{+}.
- The master is notified that T^{+} is the new tail.
- Clients are notified that query requests should be directed to T^{+}.

4 Primary/Backup Protocols

Chain replication is a form of primary/backup approach [3], which itself is an instance of the state machine approach [22] to replica management. In the primary/backup approach, one server, designated the *primary*

- imposes a sequencing on client requests (and thereby ensures strong consistency holds),
- distributes (in sequence) to other servers, known as *backups*, the client requests or resulting updates,
- awaits acknowledgements from all non-faulty backups, and
- after receiving those acknowledgements then sends a reply to the client.

If the primary fails, one of the back-ups is promoted into that role.

With chain replication, the primary's role in sequencing requests is shared by two replicas. The head sequences update requests; the tail extends that sequence by interleaving query requests. This sharing of responsibility not only partitions the sequencing task but also enables lower-latency and lower-overhead processing for query requests, because only a single server (the tail) is involved in processing a query and that processing is never delayed by activity elsewhere in the chain. Compare that to the primary backup approach, where the primary, before responding to a query, must await acknowledgements from backups for prior updates.

In both chain replication and in the primary/backup approach, update requests must be disseminated to all servers replicating an object or else the replicas will diverge. Chain replication does this dissemination serially, resulting in higher latency than the primary/backup approach where requests were distributed to backups in parallel. With parallel dissemination, the time needed to generate a reply is proportional to the maximum latency of any non-faulty backup; with serial dissemination, it is proportional to the sum of those latencies.

Simulations reported in §5 quantify all of these performance differences, including variants of chain replication and the primary/backup approach in which query requests are sent to any server (with expectations of trading increased performance for the strong consistency guarantee).

Simulations are not necessary for understanding the differences in how server failures are handled by the two approaches, though. The central concern here is the duration of any transient outage experienced by clients when the service reconfigures in response to a server failure; a second concern is the added latency that server failures introduce.

The delay to detect a server failure is by far the dominant cost, and this cost is identical for both chain replication and the primary/backup approach. What follows, then, is an analysis of the recovery costs for each approach assuming that a server failure has been detected; message delays are presumed to be the dominant source of protocol latency.

For chain replication, there are three cases to consider: failure of the head, failure of a middle server, and failure of the tail.

- **Head Failure.** Query processing continues uninterrupted. Update processing is unavailable for 2 message delivery delays while the master broadcasts a message to the new head and its successor, and then it notifies all clients of the new head using a broadcast.
- **Middle Server Failure.** Query processing continues uninterrupted. Update processing can be delayed but update requests are not lost, hence no transient outage is experienced, provided some server in a prefix of the chain that has received the request remains operating.

Failure of a middle server can lead to a delay in processing an update request—the protocol of Figure 3 involves 4 message delivery delays.

- **Tail Failure.** Query and update processing are both unavailable for 2 message delivery delays while the master sends a message to the new tail and then notifies all clients of the new tail using a broadcast.

With the primary/backup approach, there are two cases to consider: failure of the primary and failure of a backup. Query and update requests are affected the same way for each.

- **Primary Failure.** A transient outage of 5 message delays is experienced, as follows. The master detects the failure and broadcasts a message to all backups, requesting the number of updates each has processed and telling them to suspend processing requests. Each backup replies to the master. The master then broadcasts the identity of the new primary to all backups. The new primary is the one having processed the largest number of updates, and it must then forward to the backups any updates that they are missing. Finally, the master broadcasts a message notifying all clients of the new primary.

- **Backup Failure.** Query processing continues uninterrupted provided no update requests are in progress. If an update request is in progress then a transient outage of at most 1 message delay is experienced while the master sends a message to the primary indicating that acknowledgements will not be forthcoming from the faulty backup and requests should not subsequently be sent there.

So the worst case outage for chain replication (tail failure) is never as long as the worst case outage for primary/backup (primary failure); and the best case for chain replication (middle server failure) is shorter than the best case outage for primary/backup (backup failure). Still, if duration of transient outage is the dominant consideration in designing a storage service then choosing between chain replication and the primary/backup approach requires information about the mix of request types and about the chances of various servers failing.

5 Simulation Experiments

To better understand throughput and availability for chain replication, we performed a series of experiments in a simulated network. These involve prototype implementations of chain replication as well as some of the alternatives. Because we are mostly interested in delays intrinsic to the processing and communications that chain replication entails, we simulated a network with infinite bandwidth but with latencies of 1 ms per message.

5.1 Single Chain, No Failures

First, we consider the simple case when there is only one chain, no failures, and replication factor t is 2, 3, and 10. We compare throughput for four different replication management alternatives:

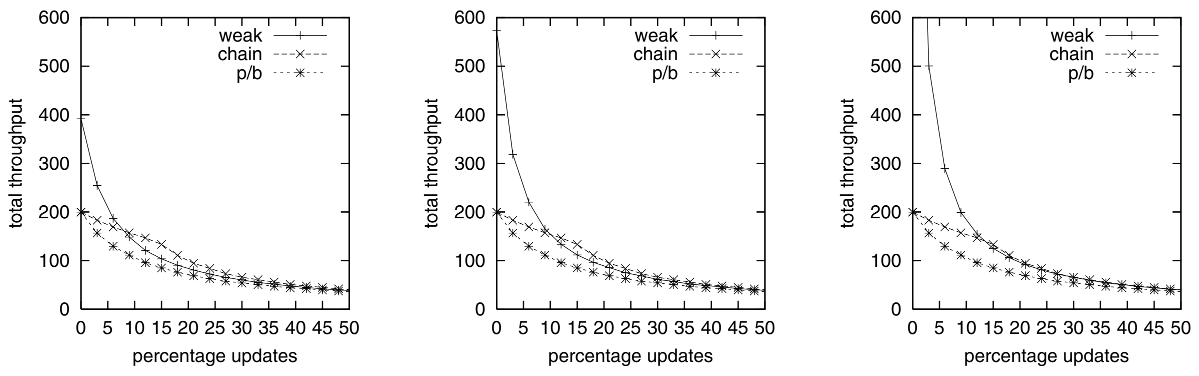
- **chain**: Chain replication.
- **p/b**: Primary/backup.
- **weak-chain**: Chain replication modified so query requests go to any random server.
- **weak-p/b**: Primary/backup modified so query requests go to any random server.

Note, **weak-chain** and **weak-p/b** do not implement the strong consistency guarantees that **chain** and **p/b** do.

We fix the query latency at a server to be 5 ms and fix the update latency to be 50 ms. (These numbers are based on actual values for querying or updating a web search index.) We assume each update entails some initial processing involving a disk read, and that it is cheaper to forward object-differences for storage than to repeat the update processing anew at each replica; we expect that the latency for a replica to process an object-difference message would be 20 ms (corresponding to a couple of disk accesses and a modest computation).

So, for example, if a chain comprises three servers, the total latency to perform an update is 94 ms: 1 ms for the message from the client to the head, 50 ms for an update latency at the head, 20 ms to process the object difference message at each of the two other servers, and three additional 1 ms forwarding latencies. Query latency is only 7 ms, however.

In Figure 4 we graph total throughput as a function of the percentage of requests that are updates for $t = 2$, $t = 3$ and $t = 10$. There are 25 clients, each doing a mix of requests split between queries and updates consistent with the given percentage. Each client submits one request at a time, delaying between requests only long enough to receive the response for the previous request. So the clients together can have as many as 25 concurrent requests outstanding. Throughput for **weak-chain**



(a) $t = 2$ (b) $t = 3$ (c) $t = 10$

Figure 4: Request throughput as a function of the percentage of updates for various replication management alternatives **chain**, **p/b**, and **weak** (denoting **weak-chain**, and **weak-p/b**) and for replication factors t.

and **weak-p/b** was found to be virtually identical, so Figure 4 has only a single curve—labeled **weak**—rather than separate curves for **weak-chain** and **weak-p/b**.

Observe that chain replication (**chain**) has equal or superior performance to primary-backup (**p/b**) for all percentages of updates and each replication factor investigated. This is consistent with our expectations, because the head and the tail in chain replication share a load that, with the primary/backup approach, is handled solely by the primary.

The curves for the weak variant of chain replication are perhaps surprising, as these weak variants are seen to perform worse than chain replication (with its strong consistency) when there are more than 15% update requests. Two factors are involved:

- The weak variants of chain replication and primary/backup outperform pure chain replication for query-heavy loads by distributing the query load over all servers, an advantage that increases with replication factor.
- Once the percentage of update requests increases, ordinary chain replication outperforms its weak variant—since all updates are done at the head. In particular, under pure chain replication (i) queries are not delayed at the head awaiting completion of update requests (which are relatively time consuming) and (ii) there is more capacity available at the head for update request processing if query requests are not also being handled there.

Since **weak-chain** and **weak-p/b** do not implement strong consistency guarantees, there would seem to be surprisingly few settings where these replication management schemes would be preferred.

Finally, note that the throughput of both chain replication and primary backup is not affected by replication factor provided there are sufficient concurrent requests so that multiple requests can be pipelined.

5.2 Multiple Chains, No Failures

If each object is managed by a separate chain and objects are large, then adding a new replica could involve considerable delay because of the time required for transferring an object's state to that new replica. If, on the other hand, objects are small, then a large storage service will involve many objects. Each processor in the system is now likely to host servers from multiple chains—the costs of multiplexing the processors and communications channels may become prohibitive. Moreover, the failure of a single processor now affects multiple chains.

A set of objects can always be grouped into a single *volume*, itself something that could be considered an object for purposes of chain replication, so a designer has considerable latitude in deciding object size.

For the next set of experiments, we assume

- a constant number of volumes,
- a hash function maps each object to a volume, hence to a unique chain, and
- each chain comprises servers hosted by processors selected from among those implementing the storage service.

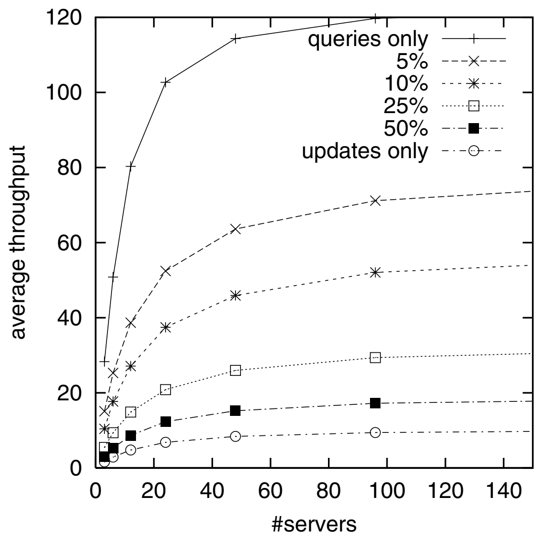


Figure 5: Average request throughput per client as a function of the number of servers for various percentages of updates.

Clients are assumed to send their requests to a *dispatcher* which (i) computes the hash to determine the volume, hence chain, storing the object of concern and then (ii) forwards that request to the corresponding chain. (The master sends configuration information for each volume to the dispatcher, avoiding the need for the master to communicate directly with clients. Interposing a dispatcher adds a 1ms delay to updates and queries, but doesn't affect throughput.) The reply produced by the chain is sent directly to the client and not by way of the dispatcher.

There are 25 clients in our experiments, each submitting queries and updates at random, uniformly distributed over the chains. The clients send requests as fast as they can, subject to the restriction that each client can have only one request outstanding at a time.

To facilitate comparisons with the GFS experiments [11], we assume 5000 volumes each replicated three times, and we vary the number of servers. We found little or no difference among **chain**, **p/b**, **weak chain**, and **weak p/b** alternatives, so Figure 5 shows the average request throughput per client for one—chain replication—as a function of the number of servers, for varying percentages of update requests.

5.3 Effects of Failures on Throughput

With chain replication, each server failure causes a three-stage process to start:

1. Some time (we conservatively assume 10 seconds in our experiments) elapses before the master detects the server failure.

2. The offending server is then deleted from the chain.

3. The master ultimately adds a new server to that chain and initiates a *data recovery* process, which takes time proportional to (i) how much data was being stored on the faulty server and (ii) the available network bandwidth.

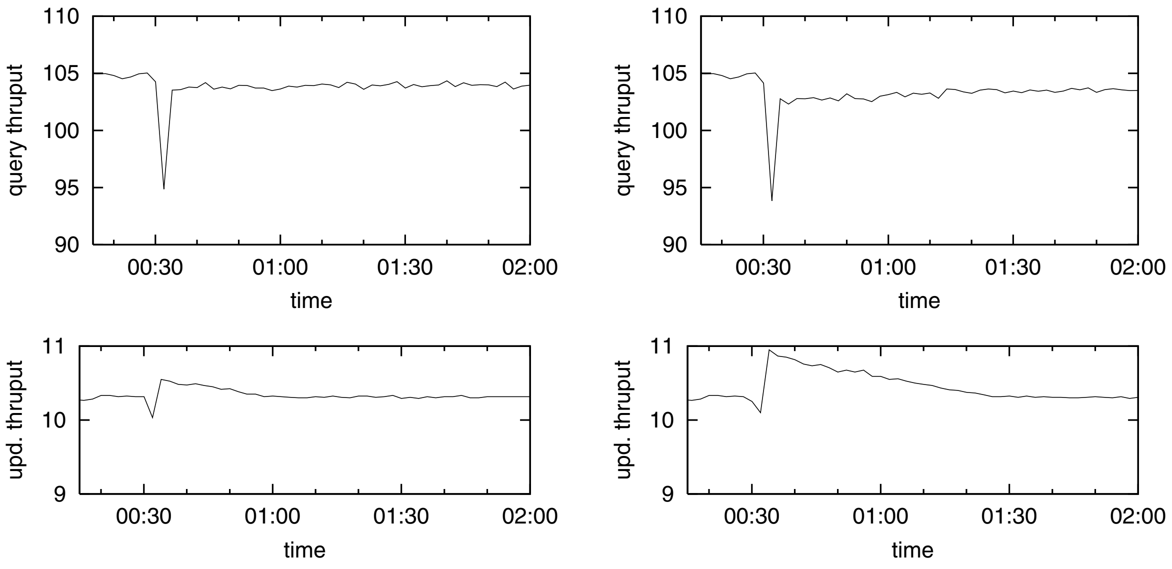
Delays in detecting a failure or in deleting a faulty server from a chain can increase request processing latency and can increase transient outage duration. The experiments in this section explore this.

We assume a storage service characterized by the parameters in Table 1; these values are inspired by what is reported for GFS [11]. The assumption about network bandwidth is based on reserving for data recovery at most half the bandwidth in a 100 Mbit/second network; the time to copy the 150 Gigabytes stored on one server is now 6 hours and 40 minutes.

In order to measure the effects of a failures on the storage service, we apply a load. The exact details of the load do not matter greatly. Our experiments use eleven clients. Each client repeatedly chooses a random object, performs an operation, and awaits a reply; a watchdog timer causes the client to start the next loop iteration if 3 seconds elapse and no reply has been received. Ten of the clients exclusively submit query operations; the eleventh client exclusively submits update operations.

parameter	*value*
number of servers (N)	24
number of volumes	5000
chain length (t)	3
data stored per server	150 Gigabytes
maximum network bandwidth devoted to data recovery to/from any server	6.25 Megabytes/sec
server reboot time after a failure	10 minutes

Table 1: Simulated Storage Service Characteristics.



(a) one failure (b) two failures

Figure 6: Query and update throughput with one or two failures at time 00:30.

Each experiment described executes for 2 simulated hours. Thirty minutes into the experiment, the failure of one or two servers is simulated (as in the GFS experiments). The master detects that failure and deletes the failed server from all of the chains involving that server. For each chain that was shortened by the failure, the master then selects a new server to add. Data recovery to those servers is started.

Figure 6(a) shows aggregate query and update throughputs as a function of time in the case a single server F fails. Note the sudden drop in throughput when the simulated failure occurs 30 minutes into the experiment. The resolution of the x-axis is too coarse to see that the throughput is actually zero for about 10 seconds after the failure, since the master requires a bit more than 10 seconds to detect the server failure and then delete the failed server from all chains.

With the failed server deleted from all chains, processing now can proceed, albeit at a somewhat lower rate because fewer servers are operational (and the same request processing load must be shared among them) and because data recovery is consuming resources at various servers. Lower curves on the graph reflect this. After 10 minutes, failed server F becomes operational again, and it becomes a possible target for data recovery. Every time data recovery of some volume successfully completes at F, query throughput improves (as seen on the graph). This is because F, now the tail for another chain, is handling a growing proportion of the query load.

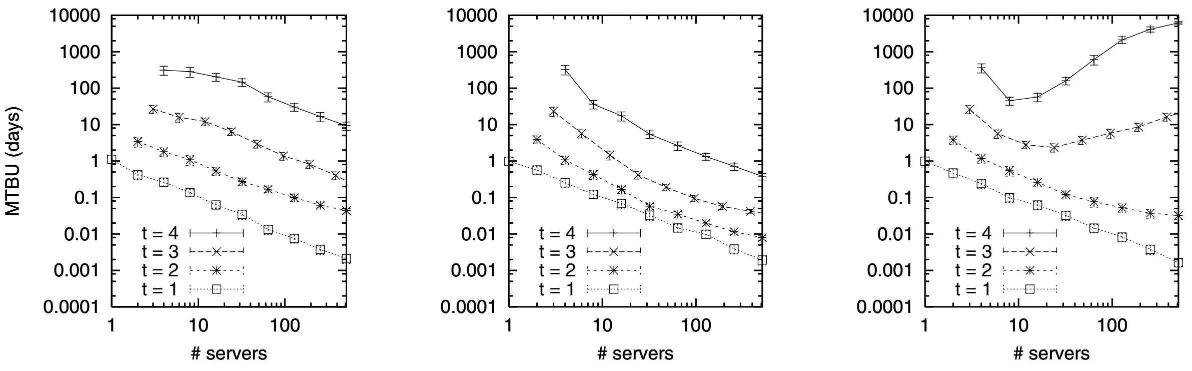
One might expect that after all data recovery concludes, the query throughput would be what it was at the start of the experiment. The reality is more subtle, because volumes are no longer uniformly distributed among the servers. In particular, server F will now participate in fewer chains than other servers but will be the tail of every chain in which it does participate. So the load is no longer well balanced over the servers, and aggregate query throughput is lower.

Update throughput decreases to 0 at the time of the server failure and then, once the master deletes the failed server from all chains, throughput is actually better than it was initially. This throughput improvement occurs because the server failure causes some chains to be length 2 (rather than 3), reducing the amount of work involved in performing an update.

The GFS experiments [11] consider the case where two servers fail, too, so Figure 6(b) depicts this for our chain replication protocol. Recovery is still smooth, although it takes additional time.

5.4 Large Scale Replication of Critical Data

As the number of servers increases, so should the aggregate rate of server failures. If too many servers fail, then a volume might become unavailable. The



(a) **ring** (b) **rndseq** (c) **rndpar**

Figure 7: The MTBU and 99% confidence intervals as a function of the number of servers and replication factor for three different placement strategies: (a) DHT-based placement with maximum possible parallel recovery; (b) random placement, but with parallel recovery limited to the same degree as is possible with DHTs; (c) random placement with maximum possible parallel recovery.

probability of this depends on how volumes are placed on servers and, in particular, the extent to which parallelism is possible during data recovery.

We have investigated three volume placement strategies:

- **ring**: Replicas of a volume are placed at consecutive servers on a ring, determined by a consistent hash of the volume identifier. This is the strategy used in CFS [7] and PAST [19]. The number of parallel data recoveries possible is limited by the chain length t.
- **rndpar**: Replicas of a volume are placed randomly on servers. This is essentially the strategy used in GFS.[5] Notice that, given enough servers, there is no limit on the number of parallel data recoveries possible.
- **rndseq**: Replicas of a volume are placed randomly on servers (as in **rndpar**), but the maximum number of parallel data recoveries is limited by t (as in **ring**). This strategy is not used in any system known to us but is a useful benchmark for quantifying the impacts of placement and parallel recovery.

To understand the advantages of parallel data recovery, consider a server F that fails and was participating in chains $C_1, C_2, \ldots, C_n$. For each chain C_i, data recovery requires a source from which the volume data is fetched and a host that will become the new element of chain C_i. Given enough processors and no constraints on the placement of volumes, it is easy to ensure that the new elements are all disjoint. And with random placement of volumes, it is likely that the sources will be disjoint as well. With disjoint sources and new elements, data recovery for chains $C_1, C_2, \ldots, C_n$ can occur in parallel. And a shorter interval for data recovery of $C_1, C_2, \ldots, C_n$, implies that there is a shorter window of vulnerability during which a small number of concurrent failures would render some volume unavailable.

We seek to quantify the mean time between unavailability (MTBU) of any object as a function of the number of servers and the placement strategy. Each server is assumed to exhibit exponentially distributed failures with a MTBF (Mean Time Between Failures) of 24 hours.[6] As the number of servers in a storage system increases, so would the number of volumes (otherwise, why add servers). In our experiments, the number of volumes is defined to be 100 times the initial number of servers, with each server storing 100 volumes at time 0.

We postulate that the time it takes to copy all the data from one server to another is four hours, which corresponds to copying 100 Gigabytes across a 100 Mbit/sec network restricted so that only half bandwidth can be used for data recovery. As in the GFS experiments, the maximum number of parallel data recoveries on the network is limited to 40% of the servers, and the minimum transfer time is set to 10 seconds (the time it takes to copy an individual GFS object, which is 64 KBytes).

Figure 7(a) shows that the MTBU for the ring strategy appears to have an approximately Zipfian distribution as a function of the number of servers.

Thus, in order to maintain a particular MTBU, it is necessary to grow chain length t when increasing the number of servers. From the graph, it seems as though chain length needs to be increased as the logarithm of the number of servers.

Figure 7(b) shows the MTBU for **rndseq**. For $t > 1$, **rndseq** has lower MTBU than **ring**. Compared to **ring**, random placement is inferior because with random placement there are more sets of t servers that together store a copy of a chain, and therefore there is a higher probability of a chain getting lost due to failures.

However, random placement makes additional opportunities for parallel recovery possible if there are enough servers. Figure 7(c) shows the MTBU for **rndpar**. For few servers, **rndpar** performs the same as **rndseq**, but the increasing opportunity for parallel recovery with the number of servers improves the MTBU, and eventually **rndpar** outperforms **rndseq**, and more importantly, it outperforms **ring**.

6 Related Work

Scalability. Chain replication is an example of what Jimenéz-Peris and Patiño-Martínez [14] call a ROWAA (read one, write all available) approach. They report that ROWAA approaches provide superior scaling of availability to quorum techniques, claiming that availability of ROWAA approaches improves exponentially with the number of replicas. They also argue that non-ROWAA approaches to replication will necessarily be inferior. Because ROWAA approaches also exhibit better throughout than the best known quorum systems (except for nearly write-only applications) [14], ROWAA would seem to be the better choice for replication in most real settings.

Many file services trade consistency for performance and scalability. Examples include Bayou [17], Ficus [13], Coda [15], and Sprite [5]. Typically, these systems allow continued operation when a network partitions by offering tools to fix inconsistencies semi-automatically. Our chain replication does not offer graceful handling of partitioned operation, trading that instead for supporting all three of: high performance, scalability, and strong consistency.

Large-scale peer-to-peer reliable file systems are a relatively recent avenue of inquiry. OceanStore [6], FARSITE [2], and PAST [19] are examples. Of these, only OceanStore provides strong (in fact, transactional) consistency guarantees.

Google's File System (GFS) [11] is a large-scale cluster-based reliable file system intended for applications similar to those motivating the invention of chain replication. But in GFS, concurrent overwrites are not serialized and read operations are not synchronized with write operations. Consequently, different replicas can be left in different states, and content returned by read operations may appear to vanish spontaneously from GFS. Such weak semantics imposes a burden on programmers of applications that use GFS.

Availability versus Consistency. Yu and Vahdat [25] explore the trade-off between consistency and availability. They argue that even in relaxed consistency models, it is important to stay as close to strong consistency as possible if availability is to be maintained in the long run. On the other hand, Gray *et al.* [12] argue that systems with strong consistency have unstable behavior when scaled-up, and they propose the *tentative update transaction* for circumventing these scalability problems.

Amza *et al.* [4] present a one-copy serializable transaction protocol that is optimized for replication. As in chain replication, updates are sent to all replicas whereas queries are processed only by replicas known to store all completed updates. (In chain replication, the tail is the one replica known to store all completed updates.) The protocol of [4] performs as well as replication protocols that provide weak consistency, and it scales well in the number of replicas. No analysis is given for behavior in the face of failures.

Replica Placement. Previous work on replica placement has focussed on achieving high throughput and/or low latency rather than on supporting high availability. Acharya and Zdonik [1] advocate locating replicas according to predictions of future accesses (basing those predictions on past accesses). In the Mariposa project [23], a set of rules allows users to specify where to create replicas, whether to move data to the query or the query to the data, where to cache data, and more. Consistency is transactional, but no consideration is given to availability. Wolfson *et al.* consider strategies to optimize database replica placement in order to optimize performance [24]. The OceanStore project also considers replica placement [10, 6] but from the CDN (Content Distribution Network, such as Akamai) perspective of creating as few replicas as possible while supporting certain quality of service guarantees. There is a significant body of work (e.g., [18]) concerned with placement of web page replicas as well, all from the perspective of reducing latency and network load.

Douceur and Wattenhofer investigate how to maximize the worst-case availability of files in FARSITE [2], while spreading the storage load evenly across all servers [8, 9]. Servers are assumed to have varying availabilities. The algorithms they consider repeatedly swap files between machines if doing so improves file availability. The results are of a theoretical nature for simple scenarios; it is unclear how well these algorithms will work in a realistic storage system.

7 Concluding Remarks

Chain replication supports high throughput for query and update requests, high availability of data objects, and strong consistency guarantees. This is possible, in part, because storage services built using chain replication can and do exhibit transient outages but clients cannot distinguish such outages from lost messages. Thus, the transient outages that chain replication introduces do not expose clients to new failure modes—chain replication represents an interesting balance between what failures it hides from clients and what failures it doesn't.

When chain replication is employed, high availability of data objects comes from carefully selecting a strategy for placement of volume replicas on servers. Our experiments demonstrated that with DHT-based placement strategies, availability is unlikely to scale with increases in the numbers of servers; but we also demonstrated that random placement of volumes does permit availability to scale with the number of servers if this placement strategy is used in concert with parallel data recovery, as introduced for GFS.

Our current prototype is intended primarily for use in relatively homogeneous LAN clusters. Were our prototype to be deployed in a heterogeneous wide-area setting, then uniform random placement of volume replicas would no longer make sense. Instead, replica placement would have to depend on access patterns, network proximity, and observed host reliability. Protocols to re-order the elements of a chain would likely become crucial in order to control load imbalances.

Our prototype chain replication implementation consists of 1500 lines of Java code, plus another 2300 lines of Java code for a Paxos library. The chain replication protocols are structured as a library that makes upcalls to a storage service (or other application). The experiments in this paper assumed a "null service" on a simulated network. But the library also runs over the Java socket library, so it could be used to support a variety of storage service-like applications.

Acknowledgements. Thanks to our colleagues Håkon Brugård, Kjetil Jacobsen, and Knut Omang at FAST who first brought this problem to our attention. Discussion with Mark Linderman and Sarah Chung were helpful in revising an earlier version of this paper. We are also grateful for the comments of the OSDI reviewers and shepherd Margo Seltzer. A grant from the Research Council of Norway to FAST ASA is noted and acknowledged.

Van Renesse and Schneider are supported, in part, by AFOSR grant F49620–03–1–0156 and DARPA/AFRL-IFGA grant F30602–99–1–0532, although the views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

Notes

[1]The case where $V = newVal$ yields a semantics for update that is simply a file system write operation; the case where $V = F(newVal, objID)$ amounts to support for atomic read-modify-write operations on objects. Though powerful, this semantics falls short of supporting transactions, which would allow a request to query and/or update multiple objects indivisibly.

[2]An actual implementation would probably store the current value of the object rather than storing the sequence of updates that produces this current value. We employ a sequence of updates representation here because it simplifies the task of arguing that strong consistency guarantees hold.

[3]If $Hist_{objID}$ stores the current value of $objID$ rather than its entire history then "$Hist_{objID} \cdot r$" should be interpreted to denote applying the update to the object.

[4]If $Hist^i_{objID}$ is the current state rather than a sequence of updates, then $\preceq$ is defined to be the "prior value" relation rather than the "prefix of" relation.

[5]Actually, the placement strategy is not discussed in [11]. GFS does some load balancing that results in an approximately even load across the servers, and in our simulations we expect that random placement is a good approximation of this strategy.

[6]An unrealistically short MTBF was selected here to facilitate running long-duration simulations.

References

[1] S. Acharya and S.B. Zdonik. An efficient scheme for dynamic data replication. Technical Report CS-93-43, Brown University, September 1993.

[2] A. Adya, W.J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J.R. Douceur, J. Howell, J.R. Lorch, M. Theimer, and R.P. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proc. of the 5th Symp. on Operating Systems Design and Implementation*, Boston, MA, December 2002. USENIX.

[3] P.A. Alsberg and J.D. Day. A principle for resilient sharing of distributed resources. In *Proc. of the 2nd Int. Conf. on Software Engineering*, pages 627–644, October 1976.

[4] C. Amza, A.L. Cox, and W. Zwaenepoel. Distributed Versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In *Proc. of Middleware'03*, pages 282–304, Rio de Janeiro, Brazil, June 2003.

[5] M.G. Baker and J.K. Ousterhout. Availability in the Sprite distributed file system. *Operating Systems Review*, 25(2):95–98, April 1991. Also appeared in the 4th ACM SIGOPS European Workshop – Fault Tolerance Support in Distributed Systems.

[6] Y. Chen, R.H. Katz, and J. Kubiatowicz. Dynamic replica placement for scalable content delivery. In *Proc. of the 1st Int. Workshop on Peer-To-Peer Systems*, Cambridge, MA, March 2002.

[7] F. Dabek, M.F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. of the 18th ACM Symp. on Operating Systems Principles*, Banff, Canada, October 2001.

[8] J.R. Douceur and R.P. Wattenhofer. Competitive hill-climbing strategies for replica placement in a distributed file system. In *Proc. of the 15th International Symposium on DIStributed Computing*, Lisbon, Portugal, October 2001.

[9] J.R. Douceur and R.P. Wattenhofer. Optimizing file availability in a secure serverless distributed file system. In *Proc. of the 20th Symp. on Reliable Distributed Systems*. IEEE, 2001.

[10] D. Geels and J. Kubiatowicz. Replica management should be a game. In *Proc. of the 10th European SIGOPS Workshop*, Saint-Emilion, France, September 2002. ACM.

[11] S. Ghermawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proc. of the 19th ACM Symp. on Operating Systems Principles*, Bolton Landing, NY, October 2003.

[12] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. of the International Conference on Management of Data (SIGMOD)*, pages 173–182. ACM, June 1996.

[13] J.S. Heidemann and G.J. Popek. File system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.

[14] R. Jimenéz-Peris and M. Patiño-Martínez. Are quorums an alternative for data replication? *ACM Transactions on Database Systems*, 28(3):257–294, September 2003.

[15] J. Kistler and M. Satyanarayanann. Disconnected operation in the Coda file system (preliminary version). *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

[16] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[17] K. Petersen, M.J. Spreitzer, D.B. Terry, M.M. Theimer, and A.J. Demers. Flexible update propagation for weakly consistent replication. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, pages 288–301, Saint-Malo, France, October 1997.

[18] L. Qiu, V.N. Padmanabhan, and G.M. Voelker. On the placement of web server replicas. In *Proc. of the 20th INFOCOM*, Anchorage, AK, March 2001. IEEE.

[19] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large scale, persistent peer-to-peer storage utility. In *Proc. of the 18th ACM Symp. on Operating Systems Principles*, Banff, Canada, October 2001.

[20] J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[21] F.B. Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Transactions on Computer Systems*, 2(2):145–154, May 1984.

[22] F.B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[23] M. Stonebraker, P.M. Aoki, R. Devine, W. Litwin, and M. Olson. Mariposa: A new architecture for distributed data. In *Proc. of the 10th Int. Conf. on Data Engineering*, Houston, TX, 1994.

[24] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM Transactions on Computer Systems*, 22(2):255–314, June 1997.

[25] H. Yu and A. Vahdat. The cost and limits of availability for replicated services. In *Proc. of the 18th ACM Symp. on Operating Systems Principles*, Banff, Canada, October 2001.