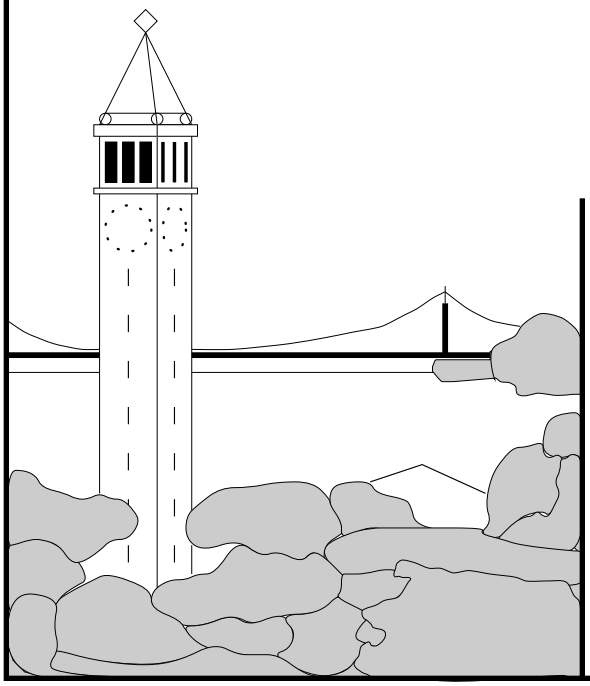
Data Replication in OceanStore

Dennis Geels
University of California, Berkeley
geels@cs.berkeley.edu



Report No. UCB/CSD-02-1217

November 2002

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Data Replication in OceanStore

Dennis Geels
University of California, Berkeley
geels@cs.berkeley.edu

November 2002

Abstract

We present the design and implementation of the data replication subsystem of OceanStore, a global-scale storage system. Our system automatically replicates data on or near the the client machines where the data is accessed, in order to improve locality, scalability, and availability. These replicas cooperate to share data and disseminate updates securely and efficiently.

Replicas also provide a flexible read interface that allows applications to relax data consistency in exchange for improved performance and availability. This interface also supports retrieval of arbitrarily old versions of client data ("time travel").

1 Introduction

Wide-area, distributed storage systems can provide much higher data accessibility and durability than more localized alternatives. Their distributed nature allows mobile users to access their files while traveling and facilitates sharing among remote collaborators. To increase durability, the storage system can place copies of data objects[1] (files) on geographically distant machines, improving resistance to natural disasters and power outages, in addition to normal failures and attacks.

These benefits are diminished by the problems introduced by distributed storage: security, complexity, and performance. This paper presents a data replication system designed to mitigate these problems. Our system, implemented as part of the OceanStore global-scale storage system [27], automatically creates and maintains soft-state replicas of data near client machines. Users then interact securely with these local replicas, avoiding the high latency of wide-area network communication.

[1] On the granularity of entire objects, blocks, or erasure-coded fragments[9].

1.1 OceanStore

This section briefly summarizes some key features of OceanStore, the storage system in which our replica system was built.

OceanStore is a persistent data store designed to support billions of users and exabytes (10^{18} bytes) of data. Reaching this somewhat ambitious goal requires the cooperation of millions of servers. OceanStore makes two core assumptions to address this issue.

First, machines participate in a utility model; no single entity must own or control all the machines in the network. Consumers pay a monthly fee to a single service provider, who buys and sells capacity with other providers to ensure the guaranteed levels of durability and availability. Each provider's machines form an autonomous administrative domain; cooperative data replication across domains must be driven by mutual profitability.

Second, OceanStore never entrusts the privacy or integrity of user data to any server. All objects are encrypted by the client, and servers operate only on the resulting cyphertext. Updates to an object are processed by an *inner ring* of machines participating in a Byzantine fault tolerant protocol. The inner ring timestamps and digitally signs the object, allowing clients to later verify its authenticity. Thus *any* server can store copies of any data object, and clients can access their copies securely.

1.2 Need for Replication

Such a storage system could function with only clients and inner ring servers. Unfortunately, servers have limited CPU, storage, and network resources. As demand for a data object increases, the object's inner ring becomes a bottleneck.

Operation on a global-scale network presents further problems. Clients unlucky enough to be far from any servers in the ring will *always* see high access latency. And clients lose all access to their data if the inner ring becomes unreachable due to a network failure or due to the basic connectivity problems common in mobile

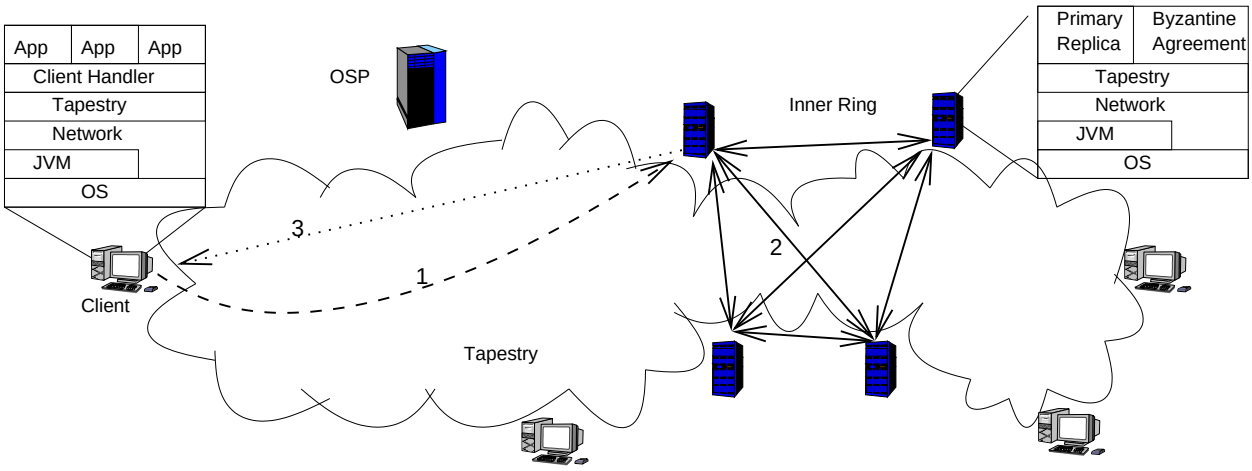


Figure 1: *A Client Request in OceanStore* A client forwards a request (1) to the inner ring servers. They agree on a response (2) which one server returns to the client (3).

networks.

1.3 Our Approach

We attack these performance and availability problems by automatically distributing copies of data objects to servers near client machines, or possibly onto the client machines themselves. We call these additional copies *replicas* or *secondary replicas*, to distinguish them from the *primary replicas* on the inner ring servers.

Secondary replicas function as local access points for the data, absorbing most of the read traffic and possibly reducing the update traffic as well. They can provide faster data access than the primary replicas, due to shorter, more stable network connections and less heavily loaded servers. They are also purely soft-state, so that the system may quickly create or destroy replicas as needed.

Replicas automatically organize themselves into multicast trees rooted at inner ring servers. They use these trees to disseminate updates as quickly and efficiently as possible. Conversely, client requests are passed up the tree when they require more or fresher data than the closest replica can provide.

Although an important locality optimization, this second tier of replicas does not weaken the security or consistency guarantees of the underlying storage system. In OceanStore, as discussed above, all data is encrypted, and updates are signed, with timestamps, by the inner ring. The read interface exported by replicas include predicates that ensure the freshness of replicated data. Clients can verify both that their updates are processed correctly and that the data they read is valid and consistent.

Due to the high latency and network failures common in a wide-area system, clients may wish to relax their consistency model in order to reduce communication with remote servers. We extended the read interface so that clients may explicitly allow slightly stale data in exchange for faster access and higher availability.

We further extended the read interface to provide access to arbitrary old versions of data. This "time travel" ability is not directly related to the performance or availability of the replication system;[2] however, it is a natural extension to our read interface that takes advantage of the versioning nature of OceanStore, which we discuss in Section 3.2.

The rest of this paper is organized as follows: in the next section we frame our system against related work. In Section 3 we present OceanStore in more detail, concentrating on a client's view of the base system. We then describe the design and implementation of our replication system in Sections 4 and 5, respectively. We evaluate its performance in Section 6. In Sections 7 and 8 we discuss the results and outline additional work planned for the future. Finally, we conclude in Section 9.

2 Related Work

We have built a replication system that dynamically places and maintains replicas on or near client machines. The replicas share data and disseminated updates using a self-organizing multicast tree. They export a flexible read interface that allows clients to ac-

[2]Time travel indirectly improves availability by eliminating data inaccessibility due to user error. Clients can easily examine and restore data after accidental overwriting or deletion.

cept loose consistency semantics in exchange for faster performance. The interface also enables "time travel": the ability to read the data as it existed at earlier points in time. Finally, the authenticity and timeliness of all data is verified using strong cryptography, so clients can securely access their data without trusting remote replicas.

We believe that our system is the first to combine these characteristics; however, many previous storage systems have provided one or more of them. In this section we outline the related contributions from these projects.

2.1 File Systems

Early distributed file systems such as AFS [32], NFS [29], and Sprite [24] allowed multiple file servers to cooperatively export a partitioned namespace. A second generation, including Amoeba [23], Coda [31], Echo [8], Ficus [25], Frangipani [35], and Harp [21], replicated files across servers to improve availability.

This server-only replication is relatively static; client machines connect directly to one of a small number of servers for each file, and client caches are not shared. The relatively small size and slow rate of replica migration obviate the need for sophisticated security models and update propagation; clients and servers authenticate each other, rather than the data, and replicas broadcast updates (or at least notifications) to all other replicas, whose locations are known.

Later some work on cooperative caching [14], as realized in xFS [7], adopted a decentralized storage model. Cooperative caching allowed clients to capitalize on available storage on their peers by sharing file caches. These systems extended the trust relationship to peer machines, and did not verify the validity of data retrieved from a client cache.

Most of these file systems have simple, inflexible consistency models, designed to be approximations of normal UNIX single-copy semantics. Coda and Ficus both support optimistic concurrency, which allows clients to modify replicated data while disconnected. Neither supports control of relaxed consistency in between these two extremes. That aspect of our system draws instead from recent work by Yu and Vahdat [38].

These distributed file systems may retain periodic checkpoints of previous system state, but they do not retain versions of individual files at a fine granularity, nor do they provide a simple interface to automatically retrieve these versions. OceanStore, like EFS [30], does keep all (or most) versions of files. Our system was designed to make time travel in OceanStore as simple as that in EFS.

2.2 Databases

Although less related to OceanStore superficially, replicated database management systems (DBMSs) have developed many of the same ideas. Franklin et al. [19] give a survey and taxonomy of replica control (cache consistency) mechanisms in the simpler, client-server replicated systems.

Bayou [15] was a revolutionary distributed system; it made some progress developing useful levels of relaxed consistency, using merge predicates and session guarantees. Bayou replicas are more lightweight and transient than normal DBMS servers, although they are restricted to replicate full copies of the database. Updates propagated to all replicas eventually using pairwise anti-entropy. Security was not addressed, and time travel was not possible.

POSTGRES [34] explored versioning as a simplifying design decision. The interface exported to retrieve data from the past, like that later adopted by EFS, was the first incarnation of accessible time travel.

Mariposa [33] built a market-based replication system atop individual POSTGRES-based servers. Like our system, Mariposa automatically constructed a dissemination tree for update streams. It also used the flexibility of the underlying POSTGRES system to relaxed the consistency model by allowing a controlled amount of staleness in the data. Unlike our system, Mariposa used replication for load sharing, not for client locality and availability. It also failed to address the security issues raised by inter-domain data sharing.

2.3 Web Caches

Cooperative web caches, such as Harvest [10] and the Summary Cache [18], serve web content from nearby cached copies. Content Delivery Networks (CDN), exemplified by Akamai [2], actively distribute and maintain replicas throughout a large, often proprietary, network of servers.

Like our replication system, both types of systems operate in the wide area and rely on locality for performance and availability much more than traditional distributed file systems. In fact, there is a large, interesting body of work in the field of *replica management*, which deals with algorithms for selecting locations for replicas that maximizes the performance and/or availability of the system. Our work contributes little to replica management research.

Although web content is usually read-only, web caches and CDNs do provide some support for controlling the staleness of caches and informing clients (insecurely) of the freshness of the data.

Unlike our system, web caches and CDNs do not allow clients to authenticate the cached data they read,

nor do they provide a time travel interface. Both of these weaknesses follow directly from a lack of support in the underlying storage medium: web pages are usually neither versioned nor verifiable.

Recently the Internet Archive [4] introduced a time travel interface for the world wide web called the "Wayback Machine". This interface provides access to a collection of recent snapshots of a significant portion of the web. Although an intriguing and well-engineered tool, the Wayback Machine cannot provide the same granularity and quality of time travel as our system due to the lack of versioning support in the underlying web content.

2.4 P2P File Sharing

Finally, our replication system perhaps has most in common with recent peer-to-peer (P2P) file sharing networks. These systems first appeared as distributed file-swapping systems like Napster [1], Gnutella [3], and Mojo Nation [5]. They were not designed for security or locality.

Several P2P file sharing systems have been designed to enable anonymous storage and censorship-resistant publishing. This set includes the Eternity Service [6], Freenet [12], FreeHaven [16], Publius [22], and Mnemosyne [20]. They each provide various guarantees of privacy and sometimes also data integrity.

These systems all assume that their data is immutable, or at most modifiable by a single author, and even then only rarely. Consistency and time travel are therefore not meaningful topics. Locality and efficiency are also not high priorities.

Recently the research community has explored P2P file systems in great depth. Some projects, like CFS [13] and PAST [17], provide both security and reasonable efficiency. Like the anonymous publishing systems, these projects deal only with immutable data, and therefore have a very simple consistency model and no support or need for time travel.

Pangaea [28], like our base system, OceanStore [26], does allow write sharing in a wide-area storage system. They use an optimistic consistency model, with support for single-copy consistency when required. Like our system, they replicate data where it is accessed, and automatically form a multicast tree for disseminating updates. Unlike our system, they do not provide any security checks.

3 Background

We have implemented data replication as a subsystem of OceanStore, a global-scale storage system. In this section we describe the portions of OceanStore relevant to data replication, in order to provide enough background for later sections.

3.1 System Overview

OceanStore is a storage system composed of computers that communicate over the internet using an overlay network named Tapestry[40]. Each machine in Tapestry is named by a 160-bit Globally Unique Identifier (GUID)[3]. Messages sent to a machine are forwarded across the overlay network "towards" its GUID.

Machines may also *publish* the names (GUIDs) of objects that they store. They publish an object GUID label by sending a message addressed to this label through Tapestry. The machines whose GUIDs closely match the object GUID remember the publishing machine and forward subsequent messages for that label back to the publisher. Clients can send requests to an object, without knowing where the object is stored. Moreover, if multiple machines publish the same label, Tapestry usually forwards messages to one that is nearby in the underlying network. The details of Tapestry's implementation are not particularly relevant to this paper; we will treat it as a magical routing black box.

Client applications access OceanStore through a local daemon process. This client-side software exports a file system interface with NFS semantics in addition to its richer, native API.

Client machines interact with a group of servers known as the *inner ring*. These servers serialize and process client requests using a Byzantine-fault-tolerant agreement protocol that guarantees that a group of $3f + 1$ servers can continue to operate correctly in spite of arbitrary failures (including attack) by f of the group. In practice, each inner ring consists of 4 powerful machines ($f = 1$) with high-bandwidth connections to the internet backbone.

Each data object may reside on a distinct inner ring, so the system can scale to a large number of objects. Each server in a ring publishes itself using the data object's name as a label. Clients then address messages with that label, and Tapestry routes them to one of the servers which is closest.

A third important entity is the client's OceanStore Service Provider (OSP). The OSP chooses servers to participate in its clients' inner rings, and is responsible for ensuring that a sufficiently large portion of each ring ($> 2/3$) remains uncorrupted. Beyond this task the OSP does not participate in client operation, and is not important to data replication.

[3]We take the SHA1 hash of a machine's public key to form its GUID. The result is globally unique with extremely high probability.

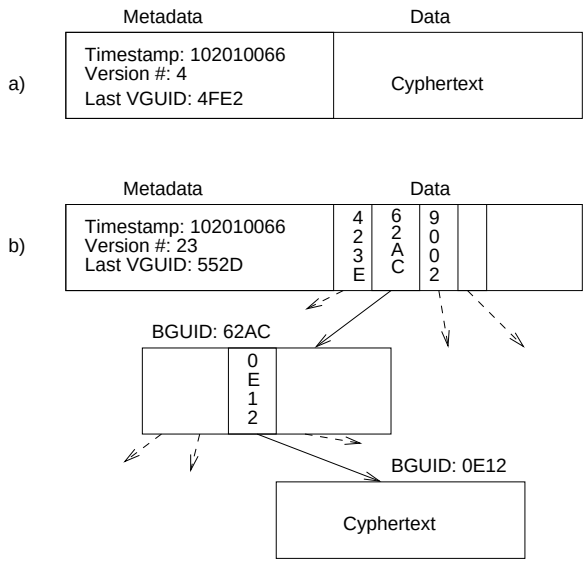


Figure 2: *Data Object Versions* (a) A simple, small data block. (b) The client data stored as a btree; the main data block references other blocks by their BGUID.

Figure 1 illustrates the various OceanStore components interacting to satisfy a client request.

3.2 Data Object Format

OceanStore is a versioning storage system. At a basic level it simply stores immutable blocks of data; mutable data objects are constructed by securely mapping a single object name to a chain of versions.

3.2.1 Read-only Versions

The name of each immutable version, called its *VGUID*, is a cryptographically-secure hash of its contents. This naming convention allows clients to verify that they have correctly received the data that they requested, as we describe later.

Each version contains a metadata hashtable and a byte array containing the client-encrypted data. The metadata contains the version's sequence number, timestamp, and the name of the previous version, to allow chaining.

For large versions the client data is stored in a btree, and only the top block of the btree is included in the data object itself. Other blocks are stored independently and are referenced by their own name, a *BGUID*, which is also a secure hash of the blocks contents[4]. This btree format is also available to clients who wish to index an object with explicit keys, rather than using the standard byte-array interface. Figure 3.2 illustrates both version formats.

Storing btree blocks independently allows sharing across objects and among multiple versions of a single object. When the inner ring applies an update to an object, producing a new version, it need only store new copies of blocks which were added or modified. This copy-on-write optimization allows OceanStore to save old versions of an object efficiently.

3.2.2 Mutability Through Heartbeats

The set of versions comprising a single data object are given another name, an *active GUID* (AGUID), equal to a secure hash over the object's human-readable name and the public key of the owner. This AGUID is the label published by servers in the inner ring, as mentioned above.

The inner ring produces signed, timestamped mappings from an object's AGUID to the VGUID of its most recent version. These mappings, which we call *heartbeats*, are produced after each update or on a client's request.

3.2.3 Verifiability

A client can verify the authenticity of a heartbeat by validating the inner ring's public key, then checking the signature on the heartbeat. The first step requires the data object's certificate (see Figure 4), and need not be rechecked for each new heartbeat.

Given a valid heartbeat, the client can verify the data of the corresponding read-only version by hashing the data and comparing the result to the expected VGUID. For large objects the client iteratively verifies the BGUIDs at each level of the tree.

In fact, a client can verify a portion of a large btree without reading the entire version. She only needs the parent blocks of the data blocks she wants to verify. Consequently, a client can store and verify very sparse copies of a data object. We use this ability in our replication system to avoid the overhead of storing entire data objects when only part is needed.

3.2.4 Temporary and Permanent Names

In theory, the OceanStore stores each version of every data object permanently. To increase the durability and availability of data blocks, OceanStore complements the data format described above with a second, archival format. The latter is a set of fragments produced by erasure-coding the data block[36]. A client can later reconstruct the data block given only a small fraction of these fragments.

The name (BGUID) of each block is actually a hash over two components: the secure hash of the online

[4]Note that a VGUID is merely the BGUID of the top block of a btree, which happens to also contain the metadata table.

form, called a BHASH, and the secure hash of the archival fragments, called the FHASH. Each hash is then stored with the converse format, allowing clients to verify either form.

Unfortunately, the erasure-coding process is computationally expensive, requiring several milliseconds to encode each block. Until the BGUID is ready, servers and clients refer to blocks internally simply by their BHASH.

These names are used only temporarily, yet provide the same security guarantees within the context of the object's inner ring and clients. For the rest of the paper we will ignore the distinction and simply use BGUID or VGUID to refer to a block's name.

3.3 Client Requests

There are three types of client requests.

3.3.1 Heartbeat Requests

The simplest request instructs the inner ring to produce and sign a new heartbeat. Clients use this request to learn the VGUID of the most recent version of an object.

3.3.2 Update Requests

OceanStore uses a predicate-based update interface, similar to the system pioneered in Bayou[15]. This update API allows great flexibility and supports a wide range of consistency semantics.

For example, one update predicate requires that the current version of a data object matches a specified VGUID. Using this predicate, an application can implement *optimistic concurrency control* (OCC), which provides transactions with ACID semantics.

Conversely, an update predicate may check nothing at all, if the client prefers to avoid conflict completely and can accept looser consistency semantics. The full set of available predicates is still under development.

Because all update responses and heartbeats must be signed by the inner ring, all update requests must naturally go to the inner ring. Each update contains a list of ⟨predicate,action⟩ pairs; the inner ring evaluates each predicate, in order, against the most recent version of the data object. It applies the action corresponding to the first successful predicate.

The response to an update request includes a heartbeat naming the new version.

3.3.3 Read Requests

The primary replica software provides a very minimal read interface. Clients may request individual blocks of a version using their BGUID.

Read responses need not be signed, because the client can verify that the block contents match the requested BGUID. Therefore, unlike update requests, read requests could be satisfied by any machine that happens to have a copy of the block.

In practice the inner ring machines also host the software components used in secondary replicas. They therefore support the same rich interface, which allows the client to read a data object by specifying its AGUID and a user-level description of the desired portion of the file. We describe this interface in detail, along with the way an AGUID is mapped to a specific version, in Section 4.6.

4 Design

We designed and implemented a data replication subsystem to complement the basic OceanStore system described in the previous section. Our system creates a secure, self-organizing second tier of soft-state replicas that sit between clients and the inner ring. This second tier can satisfy most client heartbeat- and read-requests, and may help reduce update traffic as well.

This work required two main additional components: a secondary replica to store data objects and process requests, and a dissemination tree to propagate updates and requests among those replicas. This section describes the structure and operation of these two components.

4.1 System Layout

Figure 3 shows how the second tier coordinates between the inner ring (the primary tier) and clients.

A replica will be created automatically on the client machine whenever possible. This collocation minimizes access latency and eliminates much of the network traffic generated by the client. Replicas can also be created on other available servers, in order to form a more efficient multicast tree or provide better support for weak clients.

4.2 Local Replica Design

The functionality of a secondary replica is provided by an OceanStore component (stage) that processes client requests using locally cached copies of the data object.

Figure 4 illustrates the replica data structure. These structures contain all the information required to map client requests to specific versions, including recent heartbeats and important version metadata such as sequence number and timestamp. Each replica consumes

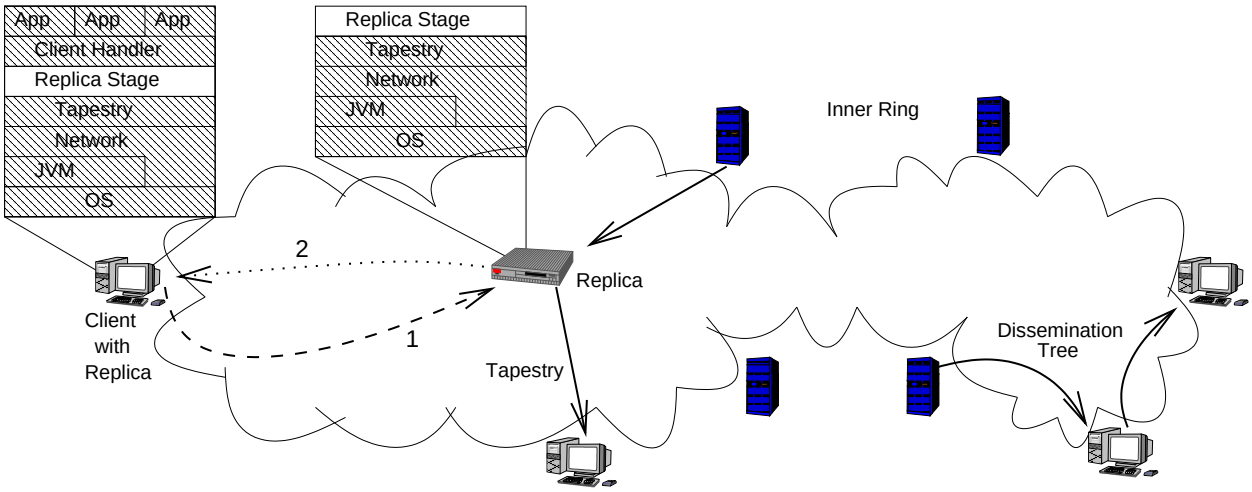


Figure 3: *Adding a Second Tier* Additional software components allow client machines and other servers to host soft-state replicas. Here a client asks to join the dissemination tree (1). A nearby replica receives the request and responds (2).

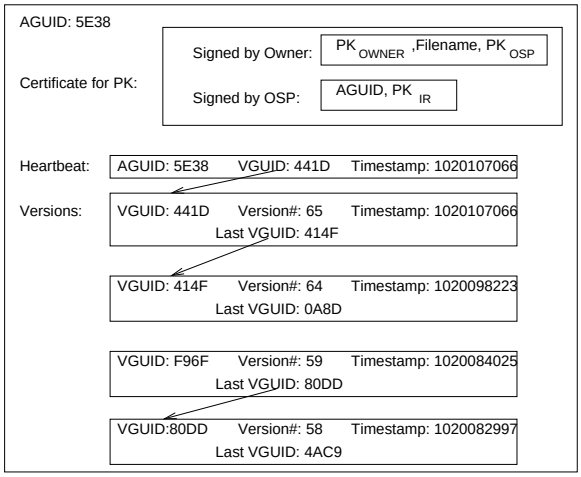


Figure 4: *A Secondary Replica* This data structure contains the information required to verify updates and map client requests to specific versions. A replica may store a small subset of a version chain; missing links are fetched if necessary.

less than 1.5KB[5] for certificates and heartbeats, plus 10 to 20 bytes per version in the history chain.

This data is purely soft-state; it is all available from the inner ring and/or the OceanStore archive, and its integrity is all verifiable, so local copies need not be stored carefully.

The data object's certificate verifies that the specified public key is the key used by the true inner ring for the object. It contains a signature from the owner's OSP verifying the inner ring's public key and a signature from the owner verifying the OSP's public key. The verification chain is complete by checking that the specified owner's key and filename hash correctly to form the AGUID.

The replica stage verifies this certificate chain and then uses the inner ring's public key to check the signatures on all later updates and heartbeats.

The replica will not have a full version history in general. Older versions will often be truncated, and the chain may have gaps if the replica becomes disconnected temporarily or if messages are dropped. These missing versions can easily be detected and fetched when necessary.

4.3 Dissemination Tree

Replicas tie themselves into multicast trees that are rooted at the primary replicas[6]. These trees distribute certificates, heartbeats, and updates to the second tier. They also provide a network up which replicas (and clients) can pass requests that they cannot process locally, due to resource limitations.

The dissemination tree is crucial to efficient write sharing. It keeps replicas loosely synchronized, which makes systems with relaxed consistency models very efficient. It also helps strict consistency models (like OCC) by invalidating stale data and exposing conflicts as early as possible. Earlier detection may even reduce

[5] Assuming 1024-bit RSA signatures. Certificate size scales linearly with signature length.

[6] In practice each data object therefore has a forest of four distinct trees. We refer to them as a single tree for clarity.

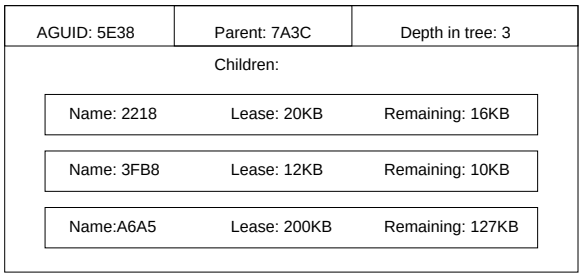


Figure 5: *Dissemination Tree State* A simple data structure which stores information required for communication across the second tier.

the amount of update traffic, by eliminating conflicting requests that are doomed to be rejected.

We chose to push updates down the dissemination tree so that the secondary replicas stay as consistent as possible. We could have adopted a lazier approach, in which a replica requested the versions it needs, on demand. That approach might consume less network resources, but adds latency to the critical read path. We look at this tradeoff in Section 6.

To create a dissemination tree, replica machines publish themselves using the data object's AGUID as a label, much like the inner ring servers. New replicas join the tree by sending a request addressed to this replica label. The message contains a lease that specifies the amount of traffic the new parent should send before disconnecting the new child.

If the machine that receives the join request has sufficient resources available, it records the new replica's GUID and sends it the certificate described in the previous section. The new parent will then answer requests from the child and forward to it all messages received from its own parent, until the child's lease expires.

If the child replica remains active it can renew the lease before that point and maintain continuous service. In practice, the child repeats the join process completely, with a more sophisticated request message.

This rejoin request may reach a different parent if new replicas have appeared since the first join. We include the distance of the previous parent from the inner ring in the request and ignore candidates which are farther away. Thus the process of periodically rejoining the tree may shrink the tree's height but will never increase it.

We discuss the overhead of maintaining these tree links in Section 6.2.1.

4.4 Creating Replicas

New replicas are created on a client machine when a client application first reads, or explicitly opens, a data object. The data structures described in Section 4.2 are initialized and the machine joins the dissemination tree as described above.

Replicas can also be created on nearby machines. Clients may send a request, similar to the join requests described above, but with a small Time-To-Live parameter. If the TTL expires before finding a replica, the request instructs the last machine it reaches to create a replica as proxy. This method of replica creation is discusses in Section 8.2.

4.5 Updating Replicas

As discussed in Section 3, all update requests must be processed by the inner ring. The primary replicas evaluate the update and, when successful, produce a new version of the data object and a signed heartbeat verifying the new version's VGUID.

The primary replicas then forward the heartbeat down the dissemination tree. It also sends the successful action from the update, as well as the VGUID for the previous version.

Each secondary replica, upon receipt of the update response, applies the update locally. The replica stage forwards the information from the response to a separate software component, which applies the update. It first checks that it has the correct initial version, fetching the required portions of that version if necessary. It then applies the action to produce the blocks of the new version. Finally, the replica stage verifies that the new VGUID matches the signed heartbeat.

Because update actions are generally implemented as logical, rather than physical changes, disseminating the update requires less bandwidth than forwarding the new data blocks would. Furthermore, if a replica is dormant or satisfied with older versions, it may request that its parent omit the update action completely and forward only the heartbeat. These truncated messages are only 400 bytes large.

Then the message serves as a signed invalidation message, signaling that the previous version is no longer the most recent. If the replica later requires fresh information it must block on requests up the tree. The tradeoff between full updates and invalidations is examined in Section 6.2.4.

When the new version has been archived, the inner ring distributes a second message, which contains a second heartbeat and the FHASHes required to produce the permanent names for the new data blocks. As with the first response, this message can be truncated to save bandwidth.

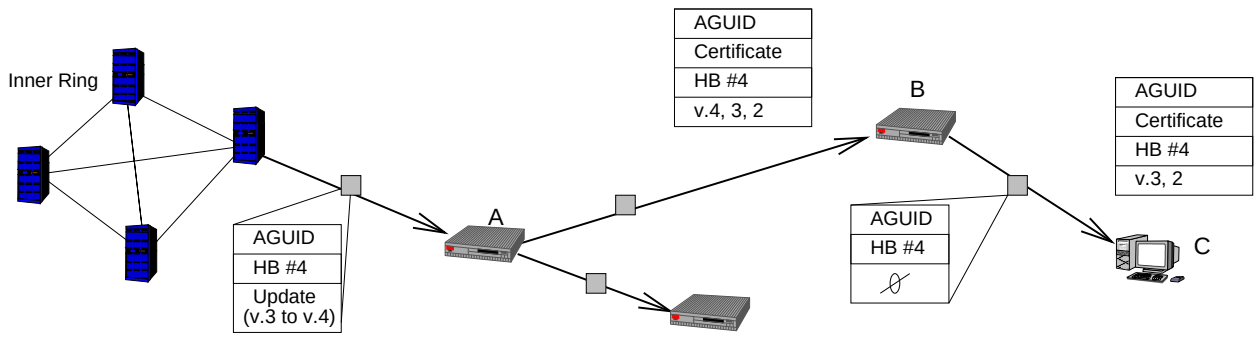


Figure 6: *Updating the Second Tier* The inner ring distributes an update response to replica A, which forwards it to its children. Replica B strips out the large update, forwarding only the heartbeat to C. This removal saves bandwidth to the small client C.

4.6 Reading a Replica

The true power of the second tier comes from its ability to satisfy client read requests.

4.6.1 Read API

Secondary replicas export a rich interface through which clients describe the data they wish to read. In addition to the AGUID of the data object, clients specify a *version predicate*, a *selection*, and a desired failure mode.

Version predicates restrict the set of versions of the data object. Possible predicates are:

1. Version's expiration date not yet reached
2. Version's sequence number in specified range or at least certain number
3. Version created in specified window or after certain time
4. Version was most recent as of certain time

There is also a trivial predicate which explicitly names a known VGUID as the only acceptable version.

One caveat is that the timestamps in heartbeats and version metadata were set by the primary replicas. If the secondary replica clocks are very skewed, the time-based predicates may be undesirable.

Selections specify the range or ranges of bytes to read, once a version is chosen. For keyed data objects, the selection specifies a set of fields and ranges within each field.

Clients may also specify a failure mode: how the replica should behave if the request cannot be fully satisfied locally. The replica may be instructed to fail immediately, to return a partial answer, to request missing information from the dissemination tree and wait to reply, or to return a partial answer and then request the missing information. Simpler clients will probably behave well only with complete answers or total failure; however, intelligent applications may prefer a fast, partial response to a slow one or none at all.

4.6.2 Consistency Semantics

The read interface just described enables clients to flexibly trade consistency guarantees for performance. Strict version predicates may be used when clients require the absolute latest version of an object. When freshness is less important, or when data objects change less frequently, a client could increase the acceptable range of versions to increase the ability of the replica stage to satisfy the request locally.

For example, we recently implemented an NFS emulation layer in the OceanStore prototype. Because NFS has very loose guarantees for the delay before clients see new versions, we used read requests that allowed versions that were current at any point in a recent window. The dissemination tree then successfully kept the replica fresh enough to handle all read requests locally.

4.6.3 Processing Requests

The replica stage searches the history chain for the endpoints that delimit the range of acceptable versions. This search requires at most two binary searches[7], which is fast. As an optimization, the replica stage first evaluates the predicate against the most recent known version. If that version is accepted, which should be a common case, the full history search is not performed.

If the version predicate is loose enough, like those which implement NFS semantics, it can usually be satisfied immediately. If not, or if the dissemination tree has not kept the replica sufficiently up-to-date, the

[7] Assuming that expiration dates are monotonic. If not, some acceptable versions may be ignored by the binary search.

replica may have to fetch a newer heartbeat or missing metadata blocks. To illustrate, see Figure 4. If a predicate required the version of that data object that was current at exactly timestamp 102009000, the replica stage would fetch missing versions 63 through 60 until it could satisfy the request.

These requests pass up the dissemination tree, giving parent replicas the opportunity to handle the request without involving the inner ring. A client machine can also send a heartbeat request directly to the inner ring, but those requests are avoided at all costs. They require long-distance communication and force the inner ring to agree and sign a new heartbeat, which is expensive.

Once the predicate is satisfied, the replica chooses the most recent version from the acceptable range and forwards its VGUID and the read selection to a separate software component, not described here, which walks the data object's btree and returns the bytes or blocks specified by the selection.

The replica then sends the data and VGUID back to the client. The client can verify that it received an acceptable version by re-applying its version predicate. If the client receives whole data blocks, rather than only byte ranges, it can also reform the btree and check that it hashes to the correct VGUID. In practice, the client skips these checks if the replica is running on the same machine.

In the future, the replica stage might not always choose the latest version in the acceptable range. For example, if little of that version is present locally, but a slightly older version is fully cached, the former has a higher chance of requiring off-node communication to complete the read. This option is discussed more in Section 8.4.

4.7 Reclaiming Resources

The second tier machines remove replicas when they fall out of use or when their resources are needed for more important tasks. Any remaining children in the dissemination tree are notified, so that they can reconnect themselves elsewhere in the tree. Then, because the local replica is only soft-state, it may simply be thrown away.

5 Implementation

We built our replication system as a software component for OceanStore servers. OceanStore is implemented in Java, for portability and ease of prototyping. Currently the same code base runs both server and client machines.

Internally, OceanStore servers employ a *staged event-driven architecture*[37]. Each major software component (e.g. network, cache management, update application) is a self-contained, event-driven *stage*. Stages communicate via an efficient publish/subscribe message dispatcher. Administrators can add new stages to servers to quickly add functionality, such as archival storage or internal monitoring.

We implemented our replication system as two stages, one to store the replicas and process requests, and a second to maintain and operate the dissemination tree. Each stage required a half-dozen request/response message pairs for communication with each other and other OceanStore stages. In total, the system consists of approximately 8000 lines of Java and a few additional files and scripts for testing. It was built in just under 6 months.

5.1 State Machine Structure

Each stage is organized as a large state machine. At a very high level, each event received by a stage advances its state and brings a request closer to completion.

In practice, the state for each data object is stored in a single data structure indexed by its AGUID, and they are kept almost completely independent. Each request or response references the AGUID (sometimes indirectly), and modifies the state of only that data object.

Programming for the event-driven model complicates the system somewhat. Asynchronous, message-passing code benefits throughput and makes stages easy to observe and control, but increases the number of possible failure conditions.

We found that we could simplify request processing by breaking a stage down into many small methods, each of which is responsible for only one type of event. A central event handler dispatches each request (or response) to a helper method for that type. The helper method process the event, adjusts the replica's state as appropriate, and sends off a request or response to continue the computation.

5.2 Difficulties

The most complex part of the code, by far, is the logic that evaluates version predicates. We had designed a fairly rich API for read requests, hoping to provide a powerful means to navigate our versioning file system. We also wanted to allow replicas to function properly without requiring the entire history of any data object.

Unfortunately these two design goals were somewhat at odds with each other, and complicated the implementation of the replica stage. The logic that searches the version chain needed to understand the concept of "gaps" in the chain, so that it could recognize cases in

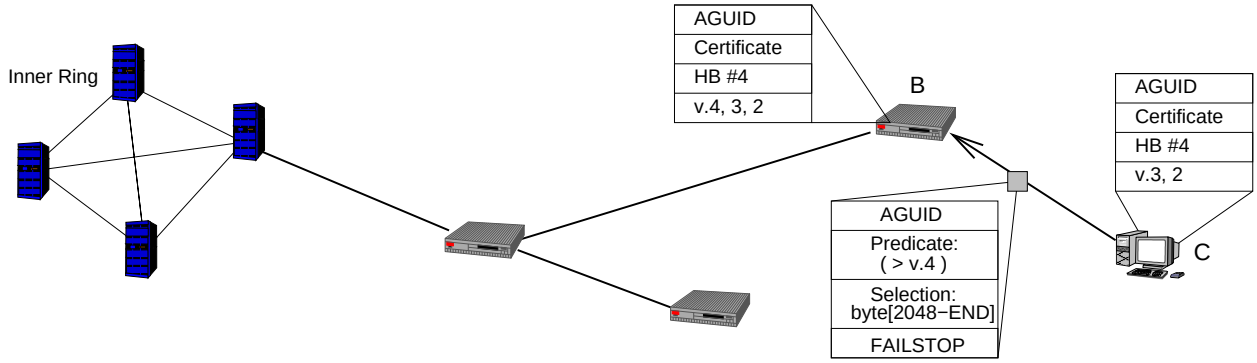


Figure 7: *Reading From the Second Tier* Replica C cannot handle a read request that requires the latest version of a data object. It therefore forwards the read request to its parent, Replica B.

which fetching more versions would or would not help satisfy a predicate.

The gaps at the front of a version chain, where the most recent version and heartbeats lie, are even more troublesome. They may require fetching a new version, a new heartbeat, or both. Fetching a heartbeat is itself a two-phase process. First a replica asks other replicas in the tree for a better heartbeat. If that request fails, the replica must send a signed request to the inner ring, because the inner ring only responds to paying clients. Creating this signed request requires coordination with the client and adds even more failure modes.

The asynchronous model meant that all communication, with the buffer cache, local client, or remote replica, required storing the state of a pending request and reviving it upon receipt of the matching response. Restarting events also proved to be somewhat tricky, especially because each version predicate required slightly different evaluation logic.

5.3 Simplifying Assumptions

We made three main assumptions in order to simplify the implementation of our system.

First, as discussed in Section 4.3, we assume that Tapestry can provide enough locality for us to build reasonable dissemination trees. Developing our own discovery service and managing our own locality metrics would have greatly increased the time required to build this first prototype.

Second, much of our code ignores the naming duplicity discussed in Section 3.2.4. It uses only the "temporary" VHASHes, which are always present in the online form of data object blocks. We do remember permanent names (VGUIDs) when available and convenient, but we assume that the temporary names are never discarded.

Finally, our code assumes that local clients are not malicious. All messages from external machines is carefully checked, and all signatures are verified, but full error checking is not performed on local messages. We believe that this assumption is valid for now, because all clients currently operate within the same address space, and could probably damage the system more effectively than by sending ill-formed requests.

6 Evaluation

We evaluated our replication system using synthetic benchmarks and an emulated wide-area network. The results suggest that the secondary replicas effectively reduce client read latency. Also, the self-organizing dissemination trees propagate new data quickly and efficiently.

6.1 Experimental Setup

We used two synthetic benchmarks to drive a real OceanStore network with artificial client workloads.

6.1.1 Benchmarks

We designed the first benchmark to simulate data sharing among collaborating users. Several clients open the same data object, then concurrently submit random read and write requests for five minutes, waiting five seconds between requests. We model somewhat loose consistency semantics: clients see each other's writes as soon as possible, but may read old versions for up to five seconds without polling the server. In practice, updates were pushed out the replicas fast enough to avoid polling.

We used this benchmark mostly to measure the affect of local replication on client read latency. It allows us to vary the amount of write traffic in the mix, and

the think time between client requests gives the dissemination tree time to propagate updates. This last detail lets us examine the performance of read requests with some degree of isolation from the update stream bandwidth.

Our second benchmark simulated single-source streaming data. A single writer repeatedly overwrites portions of a data object, with zero think time between updates. Many readers continually query their local secondary replica for the latest version of the data object. When they detect a new version they reread the data object.

This benchmark provides an excellent stress test for the dissemination tree. We use it mainly to measure the latency of update propagation.

6.1.2 Measurement Framework

We instrumented the OceanStore software to log certain actions to a local, buffered file. After each experiment the logs were automatically collected and correlated to calculate the latency of cross-machine operations. This technique consumed very little resources during the experiment. It allows us, for example, to measure the time required for an update response to reach the last replica in the dissemination tree.

Measuring inter-machine communication on a granularity of fractions of a milliseconds is difficult when the clock skew across the network is more than ten milliseconds, and also when operating in Java, which does not export a microsecond-granulary timer.

We handled for the latter problem by taking the average across many operations. We compensated for clock skew during log correlation, but subtracting from all log timestamps a local estimate of the source machine's clock skew. These estimates were updated periodically using our own clock synchronization software.

6.1.3 Infrastructure

Our main experimental testbed consists of a local cluster of forty-two machines at Berkeley. Each machine in the cluster is a IBM xSeries 330 1U rackmount PC with two 1.0 GHz Pentium III CPUs, 1.5 GB ECC PC133 SDRAM, and two 36 GB IBM UltraStar 36LZX hard drives. The machines use a single Intel PRO/1000 XF gigabit Ethernet adaptor to connect to a Packet Engines PowerRail gigabit switch. The operating system on each node is Debian GNU/Linux 3.0 (woody), running the Linux 2.4.17 SMP kernel. The two disks run in software RAID 0 (striping) mode using md raidtools-0.90. During our experiments the cluster is only lightly loaded.

We simulated slightly larger systems by placing multiple virtual OceanStore machines on a each real machine. The memory, processing, and communication resources of a secondary replica were light enough to allow tens of machines to collocate in this method[8]. For our experiments we placed primary replicas on their own servers and grouped all other nodes eight per machine.

We simulated operation in a wide-area network using an artificial transit-stub network [39] of 495 nodes. The network had inter-domain latencies of approximately 150 ms and local-area latencies of 10-50 ms. The inner ring servers were placed on well-connected nodes in different domains in the interior of the network. We then distributed one hundred other nodes randomly throughout the network

6.2 Results

We hoped to show that our replication system reduces client read latency. Figure 8 shows that it can.

We ran the first (collaboration) benchmark in three different operating modes and varied the percentage of updates in the total requests traffic. In each case, 20 nodes (one fifth of the network) were given clients that participated in the benchmark; the other 80 served only as routing nodes in the overlay network.

The first set of bars shows the average read latency for a basic system that cached only data blocks. The second set shows the change when we also cache heartbeats. For the third set we also pushed updates down the dissemination tree and applied them automatically at each secondary replica.

We broke total access time down into three parts: first, the time required to verify the freshness of cached data; second, the time spent evaluating the request's version predicate; finally, the time consumed reading the data blocks out of the buffer cache.

The first set of bars shows the base time taken to download a new heartbeat from the inner ring and fetch a read-only version across the network. We believe that the small drop at the end is due to faster communication on an unloaded network; under heavy write traffic the inner ring's agreement protocol becomes a bottleneck, and does not seem to conflict with network traffic from clients. We are investigating this anomaly further.

The second set of bars shows that caching heartbeats on secondary replicas does improve their read latency slightly. The replicas are almost never have to fetch new heartbeats (exceptions explained below), which saves several hundred milliseconds. Unfortunately, these replicas spend a great deal of time fetching data blocks across the network, so the overall latency improvement is small.

The first few bars of the second two sets show that the replicas do still poll the inner ring for heartbeats

[8]The limiting factor was the large number of file descriptors required by the OceanStore communication layer.

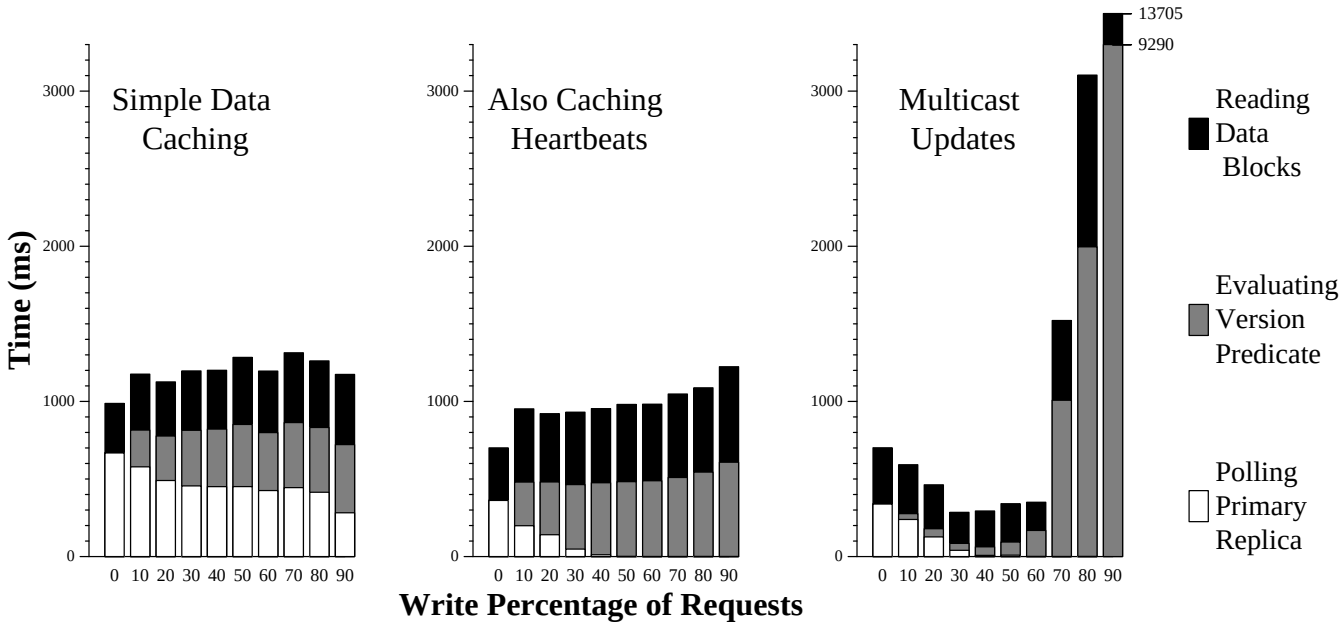


Figure 8: *The Big Picture.* Replication reduces access latency by handling more requests locally. Eagerly pushing updates to replicas helps until write traffic overloads their machines.

when the request traffic is mostly reads. With fewer (or no) new versions of the data object being pushed down the dissemination tree, the replicas were forced to explicitly request new ones to enforce their consistency guarantees. This problem could be fixed by instructing the inner ring to periodically broadcast new heartbeats (hence the term heartbeat), or by instructing the secondary replicas to proactively request them. We plan to implement both options.

The full replication system, shown by the rightmost bars, greatly reduces the latency of the all three read phases. Updates are pushed down the dissemination tree and applied locally, so replicas rarely have to request additional data blocks from the network.

When the request traffic mostly consists of updates, secondary replicas become CPU-bound as they try to apply each update from 19 other clients. All local processing slows, including the network layer responsible for fetching missing metadata required by the read requests. The resulting spike on the rightmost bars of the graph suggests that overall performance would benefit if the replicas avoided applying updates when under heavy load, and instead distributed only invalidation messages.

In the next few sections we discuss the performance of individual system components in greater detail.

6.2.1 Dissemination Tree Maintenance

Creating a new replica involves contacting a current replica, downloading the data object's certificate, and verifying that certificate. The first two phases are heavily network-dependent. On the artificial topology used for these experiments, they took on average approximately 400 ms, with a standard deviation of almost 300 ms. On a LAN we have rarely seen numbers over 50 ms. The last phase, certificate verification, is entirely CPU-bound. It takes 10-20 ms on our machines, but we believe this could be reduced by an order of magnitude using optimized cryptography code.

6.2.2 Version Predicate Evaluation

Version predicate evaluation is extremely fast in the common case, in which the replica stage can satisfy the predicate with locally cached information. To measure this speed we created a small client that repeatedly queried the replica stage for pre-cached information. Each predicate took 60 microseconds on average. That number falls to 20 microseconds if we ignore the slowest 1% of the queries; these outliers each took tens of milliseconds to complete, and were likely created by Java garbage collection stalls.

We also measured the latency of this version predicate evaluation during the experiment in Figure 8. Here evaluation required much longer because the replica stage often had to fetch new heartbeats or version metadata across the slow network.

Predicate evaluation slowed when we increased the amount of write traffic, because the number of versions to consider grew. The drastic increase in latency shown by the last few bars is caused by CPU overload, as

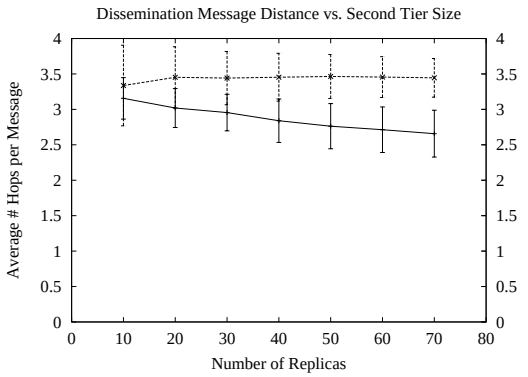


Figure 9: *Efficiency of dissemination tree.* The second tier multicast tree conserves network bandwidth by sending messages across shorter links. The error bars represent the standard deviation of the data.

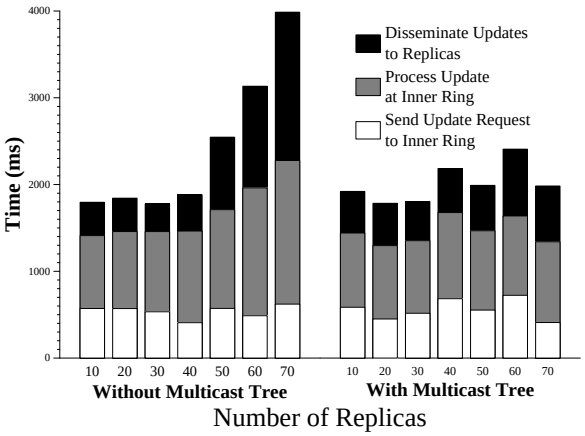


Figure 10: *The effect of the dissemination tree on system performance.* A small reduction in network usage can have a significant impact on perceived performance of the system. The dissemination tree shields the inner ring from load allowing the whole system to respond quicker.

discussed above, and is not representative of the system in isolation.

6.2.3 Buffer Cache Performance

The final component of read latency is the time required to fetch the data blocks from the version's btree out of the local buffer cache. Like other steps, the speed of reading a btree depends greatly on the number of blocks that must be requested across the network.

In our experiments reading a 4KB block from the network took over 400 ms on average, with a standard deviation of more than 300 ms. Reading from local memory is nearly instantaneous. Consequently, the timely application of updates can greatly reduce the latency of this phase of a read, as shown in Figure 8.

6.2.4 Replica Control

We next measured the efficiency of the dissemination tree. Our goal was to show that even a simple method of creating the trees could propagate updates quickly, conserve network bandwidth, and reduce inner ring load.

We ran our streaming-data benchmark with various numbers of clients reading the single data object. We compared the performance of update propagation under two dissemination policies. Under the "no multicast tree" policy, all nodes sharing the object connected directly to an inner ring server to receive update results. Using the "multicast tree" policy, readers ask the system to create replicas in the middle of the network and then connect to those replicas, locating them through Tapestry. Ideally, many replicas will connect to common nodes in the network. The result would be a reduction in the network contention and inner ring load.

We wanted to show that the multicast tree conserves network resources. Figure 9 shows the average number of Tapestry hops crossed by dissemination messages with and without a multicast tree. As the number of replicas increases, the benefit of the multicast tree increases. Because the topology Tapestry overlay network is similar to the underlying network topology, a reduction in the number of Tapestry hops indicates a probable reduction in true network distance. We plan to measure and compare these Tapestry-level measurements to true network distance in our future work.

We also observe the effect of the reduced network usage on the performance of the system. Figure 10 shows the latency of the phases of an update, from the initial request to the arrival of the response at each replica. A modest reduction in network utilization can have a significant affect on the performance of the system. When employing the "no multicast tree" policy, the inner ring nodes must handle load proportional to the number of replicas sharing the object. When this number is large, the load in serving all of the replicas impacts not only the time to deliver results to the replicas, but also the time it takes to create new versions of data.

When using the more advanced dissemination tree, the inner ring nodes can create new versions more quickly, and replicas receive notification of new nodes more quickly on average. We believe that the jitter in the graph is simply due to variance in the random placement of client machines. We are currently checking this hypothesis.

6.2.5 Replica Management

Currently our replication is somewhat naïve, in that it usually only places replicas on client machines. How-

ever, clients do have the ability to request remote machines to host a replica on their behalf–we used this ability in the previous experiment. We are very interested in exploring the extent to which we can automate the process of deciding where and when to request these additional replicas.

As a preliminary investigation, we reran the previous experiment and disabled the remote replicas. Our goal was to see how well our current approach to remote replicas work.

It turns out that disabling the remote replicas does increase the network utilization. The general trend of Figure 9 remained identical, but the new line was 6-8% higher than when remote replicas were used. This difference suggests that the remote replicas do help.

Unfortunately, the remote replicas also add approximately 200 ms, on average, to the latency of update propagation. The slower response time could be attributed to somewhat longer dissemination trees, as one would expect.

Thus our simple client-driven approach to replica management seems to form more efficient, yet taller multicast trees. It remains to be seen which property will dominate when we scale the system beyond these relatively small networks. We believe that more sophisticated replica management could achieve the same benefits without increasing overall latency. For instance, placing replicas on local network gateways or very lightly loaded servers may perform better than the current system, which ignores such information.

7 Discussion

We believe that our replication system will greatly improve OceanStore's real-world performance. These preliminary results suggest that the second tier of replicas really does improve latency and reduce load on the inner ring. Of course, these results rely on synthetic benchmarks running on an artificial topology. Although both were designed for some degree of realism, the true behavior of our replication system should be examined in a true, global-scale network with real users providing the workload.

The dissemination tree provides a surprising benefit, considering its simplicity. Perhaps more sophisticated heuristics [11] would improve its efficiency further. The tree already understands how to periodically reconnect links for robustness and simple adaptability; this interface could be used by a plug-in optimization algorithm to form better, less greedy links.

Eagerly pushing updates to replicas and applying them locally is a great idea until write traffic begins to dominate read traffic. We should add a simple switch that disables the application of the update if the local machine is already overloaded. We could the reduce bandwidth consumption by requesting only invalidation messages during times that we are unable to process the full update anyway.

The overhead of downloading and verifying a data object's certificate and heartbeats seem to be amortized well across the many reads a client performs against the object. However, our system is poorly optimized for a client who wishes to perform a single, small read request. There are several clever solutions to this problem, some of which we discuss in the following section. A simple solution would be to piggy-back a read request onto the request to join a dissemination tree. That optimization could eliminate the initial round-trip for each data object.

One characteristic that we have not been able to measure accurately is the "freshness" of the second tier. We would like to measure how often a client reads data for which a newer version has already been created on the inner ring. Alternately, we would like to enable clients to specify their consistency requirements in terms of the number of versions they can safely fall behind. Currently we can only measure and restrict the staleness of a replica by its age, not the number of unseen versions.

Finally, the inconclusive results of our examination of remote replicas (in Section 6.2.5) is disappointing. We believe that replica management is one of the most interesting directions in distribute storage research, but our current experimental setup does not allow us to scale to very interesting network sizes.

8 Future Work

There are several interesting experiments and optimizations we hope to analyze in the near future.

8.1 Replica Control

The ability to switch between propagating full updates and smaller invalidation messages allows replicas to automatically tune the amount of bandwidth they consume. The possible control algorithms and parameter space should be explored more fully.

Perhaps more interestingly, we could modify the OceanStore update format to explicitly include a *selection* object, like read requests, which describes the portions of the object modified by the update. We could also add selections to dissemination tree leases, which describe the portions of the object of interest to the child.

These additions would allow sophisticated bandwidth optimizations in the dissemination tree. First, parents need only forward updates whose range overlaps a child's selection. Second, an invalidation message

would now invalidate only the specified portion of the object. Parents could use this information to prune down the selection leased by a child. Children could use this information to prove that unaffected portions of previous versions of the data object are still valid.

Selection objects are small relative to heartbeats and updates, requiring only a few bytes per range. If necessary, their size can be optimized by forming large ranges which cover several smaller ones. This optimization may cause an update to invalidation more data than necessary, but does not affect correctness.

8.2 Replica Management

There is so much left in the world of replica management. We have two main directions along which we intend to explore.

First, we would like to implement a proactive replica system which predicts user activity and creates local replicas before they are needed. We can extract a great deal of locality within and across user access streams, even when user data itself is opaque. We can use this information to cluster related files, and then monitor and migrate clusters together. Hopefully clients will often find their data ready and waiting for them when they need it, hiding the initial request latency inherent in demand-based replication.

Second, we will evaluate the potential of a game-theoretic approach to replica management. By explicitly including bids and payments in inter-replica communication, we can produce a more flexible and responsive system than the simple lease-based approach built so far. An economic model also provides a natural way to distribute resources among competing clients and cooperate across administrative boundaries.

8.3 Reading Tentative Data

We can extend the read API to allow clients to read *tentative data*—versions created by updates that the inner ring has not yet committed.

This functionality would require modest additions to the API and extra bookkeeping, but would significantly increase client-perceived latency. When consistency requirements are loose, or update conflicts are rare, tentative data should prove very useful.

Tentative data is required for useful disconnected operation, during which a mobile client cannot communicate with the inner ring. We could even allow multiple clients to share tentative data, allowing a group of disconnected clients to make useful progress.

8.4 Better Cache Management

Our current cache manager allows limited control over the contents of the replica's memory cache. We would like to extend the replica stage's knowledge of the cache's contents. It could then make better-informed decisions when choosing a version to satisfy a client read, or when deciding whether forwarding an entire read request or merely requests for individual blocks would best conserve network resources.

8.5 Reading Archival Data

Finally, we would like to extend second tier functionality to operate in the absence of the primary tier.

The inner ring is an on-line entity, processing updates for active data. When a client reads very old or inactive data objects, the data must be read out of the OceanStore archive instead.

Secondary replicas would provide many of the same benefits to readers of inactive data. Small modifications are required because certificates and data must initially be read from the archive, rather than the inner ring. Several consistency optimizations are also available, given the knowledge that no new versions will be written.

9 Conclusion

We have built a data replication system for a global-scale storage system. This system automatically places soft-state replicas of data objects near the clients who access them.

The replicas automatically organize themselves into a multicast dissemination tree to distribute both updates and requests. Clients access replicas through a flexible API that allows verification of both the integrity and the consistency of the results.

Our initial measurements suggest that our replication system effectively reduce client read latency and that the self-organizing dissemination trees propagate new data quickly and efficiently

Much of our system relies on certain characteristics of the underlying OceanStore. Specifically, our time travel interface requires the versioning support present in the OceanStore data format. Also, our security model requires cryptographic support from the primary replica servers. This currently limits the scope of our work to the OceanStore system; however, we believe that the dissemination tree and verification components of our replica system are generally applicable. It should be easily portable to other P2P storage systems if they were extended to include signed, mutable data objects.

The system will now serve as a framework for future research. We plan to explore more sophisticated replica

management and control mechanisms that will further improve performance and stability for global-scale storage.

10 Acknowledgments

I would like to thank my wonderful wife Cheryl for her support and encouragement.

I also owe a great deal of thanks to my fellow project members, who built OceanStore and contributed a great deal to my own work: Patrick Eaton, Sean Rhea, and Hakim Weatherspoon. I would like to thank Mike Franklin for helping with this report and Mike Howard for fixing our toys when I broke them. Finally, I thank my research advisor, John Kubiatowicz.

References

[1] http://www.napster.com/.

[2] Akamai technologies, inc. http://www.akamai.com/.

[3] Gnutella. http://www.gnutellanews.com/information/.

[4] The internet archive wayback machine. http://www.archive.org/.

[5] Mojonation. http://www.mojonation.net/.

[6] R. Anderson. The eternity service. In *Proceedings of Pragocrypt*, 1996.

[7] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless Network File Systems. In *Proc. of ACM SOSP*, December 1995.

[8] A. Birrell, A. Hisgen, C. Jerian, T. Mann, and G. Swart. The echo distributed file system. Technical Report 111, DEC SRC, 1993.

[9] J. Bloemer, M. Kalfane, M. Karpinski, R. Karp, M. Luby, and D. Zuckerman. An XOR-based erasure-resilient coding scheme. Technical Report TR-95-048, The International Computer Science Institute, Berkeley, CA, 1995.

[10] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, and K. Worrell. A hierarchical internet object cache. In *Proc. of USENIX Summer Technical Conf.*, 1996.

[11] Y. Chen, R. Katz, and J. Kubiatowicz. Dynamic replica placement for scalable content delivery. In *Proc. of IPTPS*, March 2002.

[12] I. Clark, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proc. of the Workshop on Design Issues in Anonymity and Unobservability*, pages 311–320, Berkeley, CA, July 2000.

[13] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. of ACM SOSP*, October 2001.

[14] M. Dahlin, T. Anderson, D. Patterson, and R. Wang. Cooperative caching: Using remote client memory to improve file system performance. In *Proc. of OSDI*, November 1994.

[15] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. The Bayou architecture: Support for data sharing among mobile users. In *Proc. of IEEE Workshop on Mobile Computing Systems & Applications*, pages 2–7, 1994.

[16] R. Dingledine, M. Freedman, and D. Molnar. The freehaven project: Distributed anonymous storage service. In *Proc. of the Workshop on Design Issues in Anonymity and Unobservability*, July 2000.

[17] P. Druschel and A. Rowstron. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of ACM SOSP*, 2001.

[18] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary cache: A scalable wide-area Web cache sharing protocol. In *Proc. of ACM SIGCOMM Conf.*, pages 254–265, September 1998.

[19] M. Franklin, M. Carey, and M. Livny. Transactional client-server cache consistency: Alternatives and performance. *ACM TODS*, 22(3):315–363, September 1997.

[20] S. Hand and T. Roscoe. Mnemosyne: Peer-to-peer steganographic storage. In *Proc. of IPTPS*, March 2002.

[21] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the harp file system. In *Proc. of ACM SIGOPS*, 1991.

[22] A. Rubin M. Waldman and L. Cranor. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proc. 9th USENIX Security Symposium*, pages 59–72, August 2000.

[23] S. Mullender, G. van Rossum, A. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba–a distributed operating system for the 1990s. *IEEE Computer*, 23(5), 1990.

[24] M. Nelson, B. Welch, and J. Ousterhout. Caching in the sprite network file system. *IEEE/ACM Transactions on Networking*, 6(1):134–154, February 1988.

[25] T. W. Page Jr., R. G. Guy, J. S. Heidemann, D. H. Ratner, P. L. Reiher, A. Goel, G. H. Kuenning, and G. J. Popek. Perspectives on optimistically replicated, peer-to-peer filing. *Software Practice and Experience*, 28(2):155–180, February 1998.

[26] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The oceanstore prototype. In *Proc. of USENIX File and Storage Technologies FAST*, 2003.

[27] S. Rhea, C. Wells, P. Eaton, D. Geels, B. Zhao, H. Weatherspoon, and J. Kubiatowicz. Maintenance free global storage in oceanstore. In *Proc. of IEEE Internet Computing*. IEEE, September 2001.

[28] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the pangaea wide-area file system. In *Proc. of OSDI*, 2002.

[29] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the sun network filesystem. 1985.

[30] D. Santry, M. Feeley, N. Hutchinson, A. Veitch, R. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *Proc. of ACM SOSP*, December 1999.

[31] M. Satyanarayanan. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4), 1990.

[32] M. Satyanarayanan. Scalable, secure, and highly available distributed file access. *IEEE Computer*, 23(5), May 1990.

[33] J. Sidell, P. Aoki, S. Barr, A. Sah, C. Staelin, M. Stonebraker, and A. Yu. Data replication in Mariposa. In *Proc. of IEEE ICDE*, pages 485–495, February 1996.

[34] M. Stonebraker. The design of the Postgres storage system. In *Proc. of Intl. Conf. on VLDB*, September 1987.

[35] C. Thekkath, T. Mann, and E. Lee. Frangipani: A scalable distributed file system. In *Proc. of ACM SOSP*, 1997.

[36] H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Proc. of IPTPS*, March 2002.

[37] Matt Welsh. The staged event-driven architecture for highly concurrent server applications. Ph.D. Qualifying Examination Proposal, 2000.

[38] H. Yu and A. Vahdat. The costs and limits of availability for replicated services. In *Proc. of ACM SOSP*, October 2001.

[39] E. Zegura, K. Calvert, and S. Bhattacharjee. How to model an internetwork. In *Proc. of INFOCOM*, 1996.

[40] B. Zhao, A. Joseph, and J. Kubiatowicz. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, 2001.