Antiquity: Exploiting a Secure Log for Wide-Area Distributed Storage

Hakim Weatherspoon*
Cornell University
hweather@cs.cornell.edu

Patrick Eaton*
EMC Corporation
eaton_patrick@emc.com

Byung-Gon Chun and John Kubiatowicz
University of California, Berkeley
{bgchun,kubitron}@cs.berkeley.edu

ABSTRACT

Antiquity is a wide-area distributed storage system designed to provide a simple storage service for applications like file systems and back-up. The design assumes that all servers eventually fail and attempts to maintain data despite those failures. Antiquity uses a secure log to maintain data integrity, replicates each log on multiple servers for durability, and uses dynamic Byzantine fault-tolerant quorum protocols to ensure consistency among replicas. We present Antiquity's design and an experimental evaluation with global and local testbeds. Antiquity has been running for over two months on 400+ PlanetLab servers storing nearly 20,000 logs totaling more than 84 GB of data. Despite constant server churn, all logs remain durable.

Categories and Subject Descriptors

C.2.4 [**COMPUTER-COMMUNICATION NETWORKS**]: Distributed Systems—*Peer-to-peer applications*; D.4.3 [**OPERATING SYSTEMS**]: File Systems Management—*Distributed File Systems*; D.4.5 [**OPERATING SYSTEMS**]: Reliability—*Fault-tolerance*; D.0 [**SOFTWARE**]: General—Distributed wide-area storage systems

General Terms

Reliability, Performance, Design, Experimentation, Security

Keywords

Distributed Storage System, wide-area, archival storage systems, data integrity, data durability

*Work done while authors were graduate students at University of California, Berkeley.

This research was supported by the National Science Foundation under Cooperative Agreement No. ANI-0225660, `http://project-iris.net/`. Hakim Weatherspoon was supported in part by an Intel Foundation PhD Fellowship.

EuroSys'07, March 21–23, 2007, Lisboa, Portugal.

1. INTRODUCTION

Many new distributed systems—like PlanetLab [5], the Global Information Grid (GIG) [3], and GRID—are composed of machines from multiple autonomous organizations that are geographically dispersed. In these systems, servers cooperate to provide services such as persistent storage. Systems designed in this manner exhibit good scalability and resilience to localized failures such as power failures or local disasters. Unfortunately, distributed systems involving multiple, independently-managed servers suffer from new challenges such as security (including malicious components), automatic management (reliable adaptation to failure in the presence of many individual components), and instability. In PlanetLab, for example, typically less than half of the active servers are stable (available for 30 days or more) [35].

Providing secure, consistent, and available storage in these systems that exhibit extremely high levels of churn, failure, and even deliberate disruption is a challenging problem. Existing wide-area distributed storage systems, however, are not well-suited for such environments. They often support only immutable (read-only) data, do not provide consistent access to mutable (modifiable) data, do not protect and secure access to data, or are not designed for the target environment (e.g. assume fail-stop failures).

Antiquity is a distributed storage system designed to maintain data securely, consistently, and with high availability in a dynamic wide-area environment. It uses a secure log structure to maintain the integrity of stored data. It replicates data on multiple servers so that data can be retrieved later even when some replicas fail. It integrates fault-tolerance protocols to handle faults ranging from server outages to Byzantine attacks.

To test our solutions, we deployed a prototype on PlanetLab, a surprisingly volatile environment [35]. Antiquity has been running in the wide-area for over two months on 400+ PlanetLab [5] servers maintaining nearly 20,000 logs containing more than 84 GB of data. Despite the volatility of the underlying system, all logs are durable; that is, no data is lost and all logs can be read. However, tests using periodic random reads reveal that, at any given time, 6% of the logs are not modifiable since they do not have a quorum (threshold) of replicas available temporarily due to server failure on PlanetLab. All eventually become modifiable again due to Antiquity's quorum repair protocol. Antiquity's quorum repair protocol replaces lost replicas while maintaining data consistency.

Antiquity was developed in the context of OceanStore [38]. In particular, a component of OceanStore was a primary replica implemented as a Byzantine agreement process. This primary replica serialized and cryptographically signed all updates. Given this total order of all updates, the question was how to durably store and maintain the order. Antiquity's implementation of the interface and structure of a secure log assisted in durably maintaining the order

over time. When data is later read from Antiquity, the secure log and repair protocols ensure that data will be returned and that returned data is the same as stored.

The contributions of this paper are as follows.

- The design and analysis of a secure log interface that can be easily implemented in a distributed, fault-tolerant fashion.
- Design and implementation of a dynamic Byzantine fault-tolerant quorum repair protocol that maintains consistency and durability in the face of recurring server failure.
- Evaluation of an operational system that combines these features and is currently running in the wide-area.

This paper presents Antiquity's design and evaluates how effectively it can maintain data. In Section 2, we present an overview of Antiquity's goals, design, and assumptions. We describe the design in detail in Sections 3 and 4. In Sections 5 and 6, we evaluate Antiquity's ability to maintain data and discuss our experiences. Section 7 describes related work; Section 8 concludes.

2. OVERVIEW

Antiquity is a generic wide-area storage system that provides secure, durable storage. It is designed to serve as the storage layer for a variety of applications such as file systems [11, 34] and backup [36, 38]. Antiquity stores application data in a secure log to protect data integrity. It simultaneously supports many applications where application state is stored as separate logs. It provides to applications a limited interface by which they can create new logs, append data to the head of an existing log, and read data at any position in the log. It guarantees fault-tolerance through replication, consistency via dynamic Byzantine fault-tolerant quorum protocols, and efficiency by aggregation.

We describe how Antiquity integrates the above design points into one cohesive system in Sections 3 and 4, but first we discuss the goals, system model, and assumptions used to design Antiquity.

2.1 Storage System Goals

The design of Antiquity was guided by the following goals.

- Integrity: Only the owner can modify the log. Any unauthorized modifications to the log, as in substitution attacks, should be prevented.
- Incremental Secure Write and Random Read Access: A client can add data to a log securely as it is created, without local buffering. Further, the client can read arbitrary blocks without scanning the entire log.
- Durability and Consistency: The log should remain accessible despite temporary and permanent server failure. The system should ensure that logs are updated in a consistent manner.
- Efficiency/Low overhead: Protocols should limit the number of cryptographic operations and the amount of communication needed across the wide area. The infrastructure should amortize the cost of maintaining data and verifying certificates when possible.

2.2 System Model

The storage system stores logs on behalf of clients. The types of clients storing data in the system can vary widely as shown in Figure 1. The client may be the end-user machine, the server in a client-server architecture, or a replicated service. In any case, the storage system identifies a client and its secure log by a cryptographic key pair; only principals that possess the private key can modify the log. Requests that modify the state of the log must include a certificate signed by the principal's private key. Although a log is non-repudiably bound to a single key pair, multiple instances of the principal may exist simultaneously. If multiple devices possess the same private key, then they can directly modify the same log.

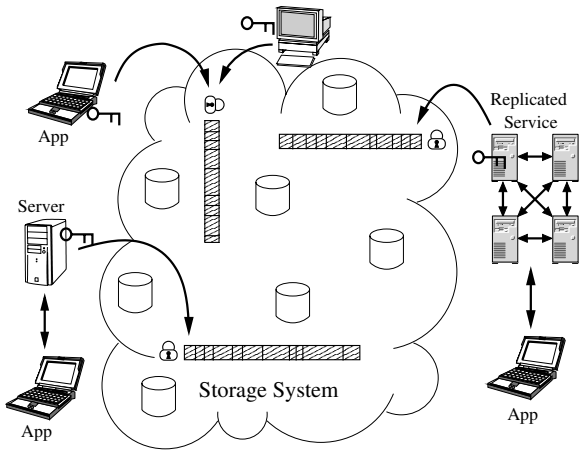


Figure 1: A log-structured storage infrastructure can provide storage for end-user clients, client-server systems, or replicated services.

Storage resources for maintaining the log are pre-allocated in chunks. When a new chunk, or *extent*, needs to be allocated, the system consults the *administrator*. The administrator authenticates the client needing to extend its log and selects a set of storage servers to host the extent. The newly-allocated portion of the log is replicated on the set of selected storage servers. To access or modify the extent, clients interact directly with the storage servers.

Applications interact with the log through a client library that exports a thin interface—`create()`, `append()`, and `read()`. To create a new log, a client obtains a new key pair and invokes the `create()` operation. The administrator authenticates the request and selects a set of servers to host the log.

After a log has been created, a client uses the `append()` operation to add data to the head of the log. The client library communicates directly with the log's storage servers to append data. The interface ensures that data is added to the log sequentially by predicating each write on the previous state of the log. If conflicting `append()` operations are submitted simultaneously, the predicate ensures at most one is applied to the log[1].

Data written to a log cannot be explicitly deleted. Instead, Antiquity supports implicit deletion based on an expiration time. The expiration time is set by the administrator when the extent is created. After the expiration time passes, the system can reclaim resources belonging to the extent. A client can prevent the expiration of an extent by extending the expiration time, though we do not discuss that feature further in this paper.

2.3 Assumptions

We assume that clients follow specified protocols, except for crashing and recovering. A malfeasant client, whether due to soft-

[1]We assume a storage server atomically handles each request. That is, a server processes requests one at a time, even though multiple requests may have been received at the same time.

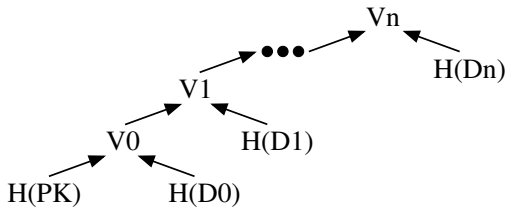


Figure 2: To compute the verifier for an extent, the system uses the recurrence relation $V_i = H(V_{i-1} + H(D_i))$. $V_{-1} = H(PK)$ where PK is a public key.

ware fault or compromised key, can prevent the system from appending data to a log. It cannot, however, affect data already stored in its log or logs belonging to other principals. If a principal's private key should be compromised, an attacker could append data to the log, but it cannot destroy data previously stored in the log. A principal can retrieve data from a log until the log's expiration time.

We assume that the administrator, tasked to select sets of storage servers to host logs [29], is trusted and non-faulty. The design, however, includes several mechanisms to mitigate the cost and consequences of this assumption. While each log uses a single administrator, different logs can use different administrators. By allowing multiple instances, the role of the administrator scales well. Second, the administrator's state can be stored as a secure log in the system. Thus, the durability of the state can be assured like any other log. Third, the state of the administrator can be cached to reduce the query load on an administrator. Finally, the administrator can be implemented as a replicated service to improve availability further.

Storage servers may exhibit Byzantine faults. We assume that, in the set of storage servers selected by the administrator to host a particular extent, a maximum threshold number of servers is faulty.

3. ANTIQUITY'S SECURE LOG

Antiquity supports a secure, append-only, log abstraction where a single log is owned by a single principal and identified by a cryptographic key pair. Only the owner of the log can `append()` to it. This narrow interface helps reduce the complexity of implementing Antiquity where consistency and durability need to be maintained efficiently. Though a single log is bound to a single cryptographic key pair, Antiquity maintains many logs associated with many separate key pairs.

Antiquity stores a log as an ordered sequence of container objects called extents. Similar to log segments in a log-structured file system [30], extents have a finite maximum size; however, each extent contains an ordered collection of variable-sized application-level blocks of data. Further, all data in an extent belongs to the same log owned by a single principal. To guard data integrity, individual log elements (blocks), whole extents, and the entire log itself are all self-verifying, which means the name of an object verifies its content.

In the following two subsections we describe the secure log structure and usage semantics. We conclude the section with a description an example file system application built on top of the secure log interface.

3.1 Secure Log Structure

A secure log is composed of two types of extents. The log head is a mutable, key-verified extent; all other extents are immutable hash-verified extents. The key-verified log head is named by a secure hash of the public key associated with the log. To verify the contents of the log head, a server compares the data to the *verifier* included in the certificate (after confirming the signature on the certificate). A verifier is a cryptographically-secure hash that asserts the integrity of both the content and append-order of an extent.

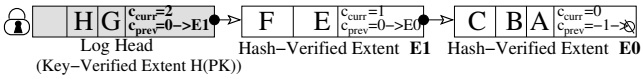


Figure 3: In two-level naming, each block is addressed by a tuple (*extent_name*, *block_name*). A block_name is simply the secure hash of the block. The extent_name for the log head is the secure hash of the public key H(PK). The extent_name for a hash-verified extent is the verifier.

We assume the mutable key-verified extent at the log head has a finite maximum size. When it becomes full, the system copies the content of the log head into an immutable hash-verified extent. A hash-verified extent is named by a function of the content of the extent. Specifically, the extent is named by the verifier in the extent's most recent certificate. A server can verify the integrity of a hash-verified extent by comparing an extent's contents to its name. These self-verifying techniques were made popular by the Self-certifying Read-only File System [16].

Making Extents Self-Verifying.

An extent verifier is computed from the names of the blocks in the extent using the chaining method [24] shown in Figure 2. Assume an extent contains a sequence of data blocks, D_i. Each data block is named with a secure, one-way hash function, $H(D_i)$. The verifier is computed using the recurrence relation $V_i = H(V_{i-1} + H(D_i))$, where $+$ is the concatenation operator. We bootstrap the process by defining V_{-1} to be a hash of the public key that signs the extent's certificate. This convention ensures that the names of extents owned by different principals do not conflict.

Creating verifiers in this manner has several advantages. When a block is appended to the log, the client can compute the verifier incrementally. This means it must hash only the new data, not all data in the log, to compute the running verifier. Additionally, a given verifier can be produced by only one particular sequence of `append()` operations. Thus, chaining creates a verifiable, time-ordered log recording data modifications. Furthermore, requiring the latest verifier as a predicate in subsequent `append()` operations assures servers maintain a consistent state of the log. Finally, when the log head is copied from a key-verified extent to a hash-verified extent, the verifier can be used as the new hash-verified name without modification.

Two-Level Naming.

To provide random access to any element in the log, Antiquity implements *two-level naming*. In two-level naming, each block is addressed not by a single name, but by a tuple. The first element of the tuple identifies the enclosing extent; the second element names the block within the extent. Retrieving data from the system is a two-step process. The system first locates the enclosing extent; then, it extracts individual application-level blocks from the extent. Both blocks and extents are self-verifying. Figure 3 illustrates two-level naming.

Two-level naming introduces added complexity in computing the address of a block of data. When an application writes a block to the log, the block is stored in the mutable extent at the head of the log. Because the log head is a mutable extent, the system cannot know the name of the hash-verified extent where the block will eventually and permanently reside.

Interface for Aggregation:

status	=	*create*(H(PK), cert);
status	=	*append*(H(PK), cert, predicate, data[ ]);
status	=	*snapshot*(H(PK), cert, predicate);
status	=	*truncate*(H(PK), cert, predicate);
status	=	*put*(cert, data[ ]);
status	=	*renew*(extent_name, cert);
cert	=	*get_cert*(extent_name);
data[]	=	*get_blocks*(extent_name, block_name[ ]);
extent	=	*get_extent*(extent_name);
data	=	*get_head*(extent_name);
mapping	=	*get_map*(extent_name);

Table 1: To support aggregation of log data, we use an extended API. A log is identified by the hash of a public key ($H(PK)$). Each mutating operation must include a certificate. The `snapshot()` and `truncate()` operations manage the extent chain; the `renew()` operation extends an extent's expiration time. The `get_blocks()` operation requires two arguments because the system implements two-level naming. The `get_*`(extent_name) operations return either the entire extent or items stored within an extent such as the certificate, mapping, or various blocks (e.g. head). The `extent_name` is either H(PK) for the log head or verifier for hash-verified extents.

To resolve this problem, each extent is assigned an integer corresponding to its position in the chain. When data is appended to the log, the address returned to the application identifies the enclosing extent by this counter. Each extent records the mapping between counter and permanent, hash-verified extent name for the previous extent. Both the mapping to the previous extent and position in the extent chain are stored in a metadata block (first block) within each extent; a call to `get_map()` returns the mapping (extent_counter, extent_name) of the previous extent (e.g. `get_map(E1)` in Figure 3 returns mapping (0, E0)).

Aggregating blocks into extents and extents into a log improves the system's efficiency in several ways. First, breaking a log into extents enables servers to intelligently allocate space for extents. Extents have a maximum size while the log itself can grow to be arbitrarily large. Second, extents decouple the infrastructure's unit of management from the client's unit of access. As a result, the storage infrastructure can amortize management costs over larger collections of data. Third, two-level naming reduces the query load on the system because clients need to query the infrastructure only once per extent, not once per block. Assuming data locality—that clients tend to access multiple blocks from an extent—systems can exploit the use of connections to manage congestion in the network better. Finally, clients writing multiple blocks to the log at the same time need only to create and sign a single certificate.

3.2 Using a Secure Log

To interact with the log, a client relies on a library that communicates with Antiquity using the interface shown in Table 1. The interface extends the `create()`/`append()` interface used by applications to support extents. All mutating operations require a certificate signed by the client for authorization. The certificate includes the verifier of the new version of the extent. The interface ensures that updates are applied sequentially by predicating each operation on the previous state of the extent. Upon completion of the operation, the certificate is stored with the extent. The `snapshot()` and `truncate()` operations help manage the chain of extents. The `renew()` operation extends the expiration time of an extent; we will not discuss `renew()` further.

Certificate contents:

verifier	token that verifies contents of log
num_blocks	the number of blocks in the container
size	the size of data stored in the container
seq_num	certificate sequence number
timestamp	creation time of certificate
ttl	time the certificate remains valid

Table 2: A certificate is contained within each operation and stored with each log. It includes fields to bind the log to its owner and other metadata fields.

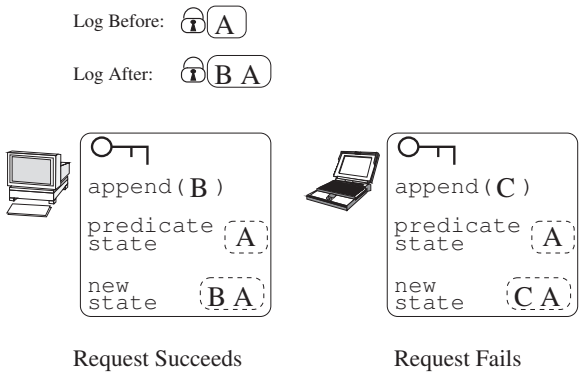


Figure 4: The interface ensures that data is added to the log in a sequential fashion by predicating each write on the previous state of the log. If conflicting `append()` operations are submitted simultaneously, the predicate ensures at most one is applied to the log, leaving the log in a consistent state.

Two of the operations enumerated in Table 1—`create()` and `snapshot()`—create new replicas. Each of these operations requires that the system contact the administrator for a configuration, set of servers, to host the new replicas. The most common operation, `append()`, does not require any interaction with the administrator.

Adding Data via `append()`.

To `append()` data to the log, a client creates a request and submits it to the storage servers. A request has three arguments: the predicate is a verifier that securely summarizes the current state of the log, the new data to append to the log, and a new certificate that includes a new verifier and new sequence number. The verifier in the certificate summarizes the next state of the log after appending data. The sequence number is a monotonically increasing number. Table 2 shows the contents of a certificate.

When a server receives an `append()` request, it determines if a request succeeds or not. It performs several checks using local knowledge. The certificate contained in the request must include a valid signature. Also, the predicate verifier contained in the request must match the current state of the log recorded by the storage server. Additionally, the verifier in the certificate must match the new verifier after appending new data to the log. Further, the sequence number in the certificate must be greater than the one currently stored. If these conditions are met, the server writes the new data to the log on its local store and returns success to the client. Otherwise, the request is rejected and failure is returned. A client receiving a failure response would need to update its local state (via `get_cert()`, `get_head()`, and `get_blocks()`) and submit a new `append()` request based on updated state.

If conflicting `append()` operations are submitted simultaneously, the predicate ensures at most one is applied to the log leaving the log in a consistent state. For example, in Figure 4, a storage server applies a workstation's request even though a laptop simultaneously submitted a (conflicting) request. As a result of the order the server handles the requests, the workstation's predicate matches the state of the log and the request succeeds. Success is returned to the workstation. However, the laptop's predicate does not match, the request fails, and failure is returned. Since the laptop's request failed, it would need to update its local state and submit a new `append()` request based on updated state.

Reading Data via `get_blocks()`.

To read data, the client library first accesses the mappings stored in the log to determine the name of the extent holding the data. It then uses the `get_blocks()` operation to retrieve the requested blocks from that extent. To accelerate the translation between counter and extent name, the client library caches the immutable mappings. Also, as an optimization, each extent contains not just the mapping for the previous extent, but a set of mappings that allow resolution in a logarithmic number of lookups.

Managing the Extent Chain.

The chain of extents is managed via the `snapshot()` and `truncate()` operations. To prevent the extent at the log head from growing too large, the client library converts the log head to hash-verified form using the `snapshot()` operation. If the system must copy the extent to a new storage server, the transfer occurs directly between storage servers without client interaction. After data has been copied to a hash-verified extent, the library uses the `truncate()` to reset the log head and point to the previous extent created via `snapshot()`. While `snapshot()` and `truncate()` are typically used together, we have elected to make them separate operations for ease of implementation. Individually, each operation is idempotent, allowing the library to retry the operation until successful execution is assured. Further, each operation requires a predicate that prevents conflicting concurrent changes. The library uses the `append()`, `snapshot()`, and `truncate()` sequence to add more data to the log.

3.3 Example Application: A Versioning File System

To demonstrate the use of the secure log interface, consider the implementation of a versioning file system application. Figure 5(a) shows a sample file system to be stored. We ignore inodes for this example and assume that files and directories are stored as a single block. The application translates the file system into a Merkle tree [31] where the secure pointer to a child file or directory is the (extent_counter, block_name) tuple. This file system structure is similar to others [11, 38, 36] and was implemented on top of a secure log interface in less than a week by one graduate student [15]. We illustrate the state of the log after initially archiving the file system (Figure 5) and after modifying two files (Figure 6).

To archive the file system, the application first creates a secure log using the `create()` operation. `create()` initializes the map block (first block of the log head) with the values $c_{\text{curr}} = 0$ and $c_{\text{prev}} = -1 \rightarrow \phi$ (current extent_counter and mapping to previous extent, respectively). Next, the application traverses the file system in a depth first manner calling `append()`, `snapshot()`, and `truncate()`. In particular, the application calls `append(budget, sched)` and records the verifiable pointers—($c = 0$, H(`budget`)) and ($c = 0$, H(`sched`)), respectively—in the `proj1` directory. Then, it calls `append(proj1)` and records the verifiable pointer in the `docs` directory ($c = 0$, H(`proj1`)). Similarly, the application calls `append(report,reqs)` and records the verifiable pointers in the `proj2` directory. However, before calling `append(proj2)`, the client library calls `snapshot()` which creates a hash-verified extent with the extent_name $E0$. The hash-verified extent mirrors the log head. The client library then calls `truncate()` which removes all the blocks from the log head and updates the mapping block by incrementing the current extent_counter to $c_{\text{curr}} = 1$ and setting the mapping of the previous extent to $c_{\text{prev}} = 0 \rightarrow E0$. The application, then, continues by calling `append(proj2)` and records the verifiable pointer ($c = 1$, H(`proj2`)) in the `docs` directory. Notice that the extent_counter is $c = 1$ instead of $c = 0$. Finally, the application calls `append(docs)`, records the verifiable pointer in the `/` directory, and calls `append(/)`. Figures 5(b) and 5(c) show the resulting Merkle tree and secure log, respectively. In a similar fashion, Figures 6(a) and 6(b) show the modified Merkle tree and secure log, respectively, after writing new versions of the `report` and `reqs` documents (`report'` and `reqs'`).

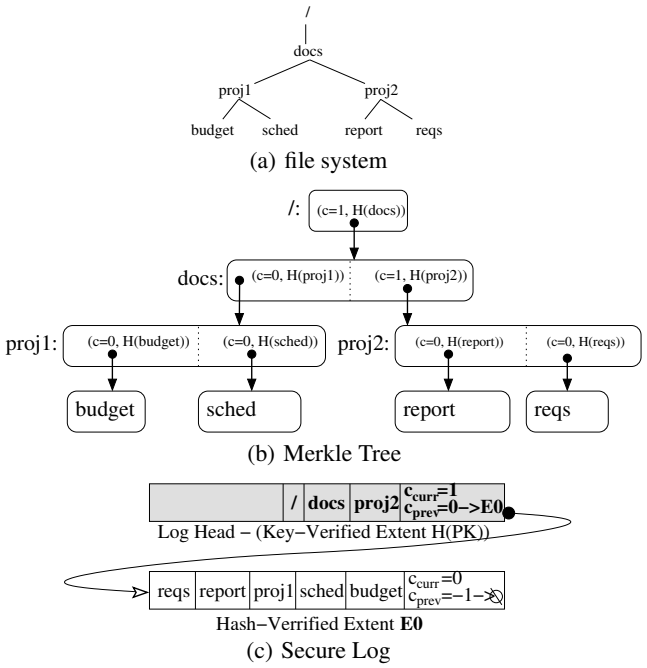


Figure 5: (a) An example file system. (b) The application translates the file system into a Merkle tree. The verifiable pointers are of the form (*extent_counter*, *block_name*). (c) The Merkle tree is stored in two extents. The first extent, $E0$, is filled and has been converted to a hash-verified extent. The second extent, the log head identified by $H(PK)$, is a partially-filled key-verified extent.

To read a particular file, the application reads the root of the file system stored at the head of the log and follows the pointers to the desired file. For example, assume the client wants to read the `sched` file. The application first calls `get_head(H(PK))` which returns the root of the file system `/'`. The root contains a verifiable pointer to the `docs'` directory ($c = 2$, H(`docs'`)). The application resolves the extent_counter $c = 2$ to the log head by calling `get_map(H(PK))`. The call returns the log head's current extent_counter value and map to the previous extent $c = 1 \rightarrow E1$ which can be cached for later use. Next, the application calls `get_blocks(H(PK), H(docs'))` which returns the `docs'`

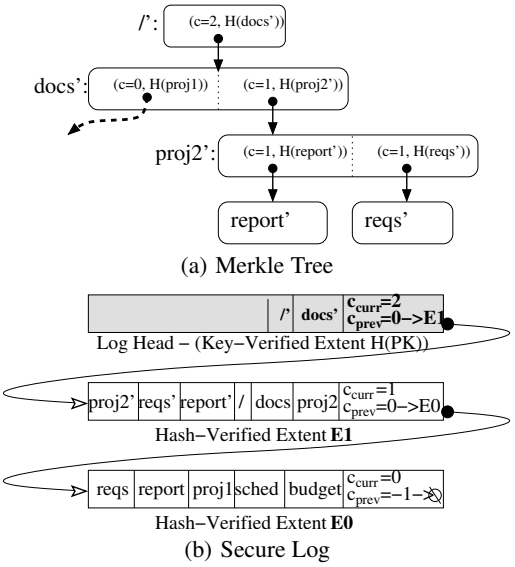


Figure 6: (a) The Merkle tree resulting from translating the updated file system. The dashed pointer indicates a reference to a block from the previous version. (b) The contents of the secure log after storing blocks of the updated file system.

directory. The `docs'` directory contains a verifiable pointer to the `proj1` directory ($c = 0$, H(`proj1`)). The application resolves the extent_counter $c = 0$ to extent_name $E0$ by calling `get_map(E1)`. The call returns the map to the previous extent $c = 0 \rightarrow E0$. Notice that the mapping from $c = 1 \rightarrow E1$ was cached from the first `get_map()` call on the log head. Finally, the application calls `get_blocks(E0,proj1)` to retrieve the `proj1` directory and `get_blocks(E0,sched)` to retrieve `sched`.

4. ANTIQUITY'S REPLICATION, CONSISTENCY, AND DURABILITY STRATEGIES

Antiquity replicates a secure log on multiple servers to provide durability. A log is durable if it persists over time. To maintain durability and ensure that progress can be made (that is, new data can be written to the log), the system must maintain consistency across replicas. The system must maintain consistency despite a variety of server and network failures and conflicting update requests. Server failures include transient failure such as reboot, permanent failure such as disk failure, and erroneous failure such as database corruption or machine compromise. Network failures include dropped connections, temporary partitions, and transmission failure such as message drop, reorder, delay, or corruption.

Antiquity employs a dynamic Byzantine fault-tolerant quorum protocol to satisfy the durability and consistency requirements. A quorum is a threshold, taking into account that some members may be faulty or malicious. For instance, Malkhi and Reiter [26] demonstrated that with self-verifying data, a configuration with $n > 3f$ servers and a quorum $q = n - f$ servers can make progress when up to f servers are faulty. In that work, configurations were static for the lifetime of the system. Martin and Alvisi [29] extended the protocol to maintain consistency in a dynamic environment. They utilize quorums to maintain two properties, *soundness* and *timeliness*, that guarantee consistency in a dynamic environment. Informally, soundness ensures data read by a client was previously written to a quorum of servers; timeliness ensures the data read is the most recent value written.

We have adapted the dynamic Byzantine quorum protocols to tolerate the failures and arbitrary behavior experienced on an environment such as PlanetLab. In particular, Antiquity creates a new configuration when a quorum in the old configuration is no longer available. We discuss the consistency via sound writes and durability via repair.

4.1 Consistency Semantics

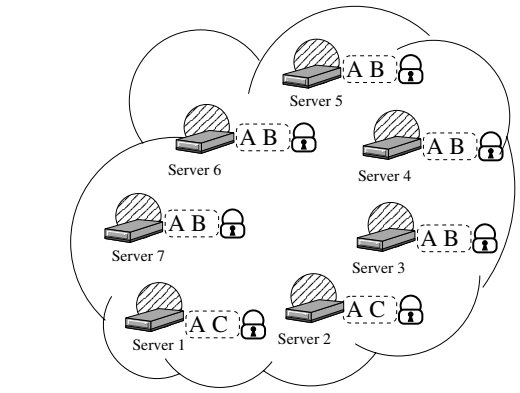
The client interacts with many replicas to complete a single operation. Operations that modify replicated state result in one of three states: sound, unsound, or undefined.

- The result of an operation is *sound* if the client receives a positive acknowledgment from a threshold of servers. A sound response means that the request succeeded and the data is durable.
- On the other hand, the result of an operation is *unsound* if the client receives a negative acknowledgment from enough servers such that positive acknowledgment from a threshold is no longer possible (e.g. sizeof(negative acks) $\geq$ sizeof(server set) $-$ threshold$+1$). A request fails if the result is unsound. The storage system does not maintain unsound results; thus, unsound writes are not durable.
- Finally, the result is *undefined* if it is neither sound or unsound. An undefined result means the client did not receive sufficient acknowledgment from servers perhaps due to network or server failure. In the case of an undefined result, a timeout occurs and the client does not know whether the request is sound or unsound.

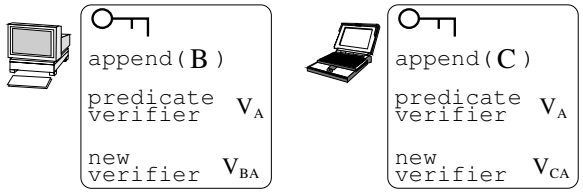
After a timeout, the client performs a `get_cert()` on all the servers and waits to receive acknowledgment from a threshold. If the state stored in the system has changed (another client updated the log), then the request is unsound. If the `get_cert()` fails to receive acknowledgment from a threshold of servers, then the client may trigger a repair audit that will determine the latest consistent state of the log (described in the next section). The client continually sends the request, reads the state of the system, then triggers a repair audit until the request is either sound or unsound.

4.2 Consistency Example

Figure 7 illustrates the notions of sound, unsound, and undefined writes. Assume a log is replicated on seven servers. A threshold required for consistency and a sound response is five positive acknowledgments. The number required for an unsound response is three negative acknowledgments (total minus a threshold plus one, $7 - 5 + 1 = 3$). The initial value stored on all the log replicas is A. Further, assume two clients, a workstation and laptop, simultaneously submit conflicting operations. The workstation attempts to append the value B and receives five positive acknowledgments and two negative, thus the response is sound since a threshold acknowledged positively. The laptop, on the other hand, attempts to append the value C and receives five negative acknowledgments and two positive, thus the response is unsound. With this scenario, the storage system should maintain the workstation's appended value B over time. Furthermore, in the above example, if the workstation receives one less positive acknowledgment (four instead of five), possibly due to network transmission error, then the result would be undefined and timeout. The workstation could read the latest replicated state of the secure log, trigger a repair audit that will repair the distributed secure log if necessary, and resubmit the request until it receives sufficient server acknowledgment.



(a) Storage System



(b) Request Succeeds (c) Request Fails

Figure 7: Semantics of a Distributed Secure Log. (a) A secure log with the value *A* is initially replicated on to seven servers. In (b), a workstation attempts to `append()` the value *B*, predicated on *A* already being stored. The result of the request is sound since it reaches a threshold of servers (servers 3-7). In (c), a laptop, which possess the same private key as the workstation, simultaneously attempts to `append()` value *C*, predicated on *A* already being stored. The result of the request is unsound since the predicate fails on a threshold servers. Note that the two servers (server 1-2) apply *C* since the predicate matches local state. However, the system should return value *B* in any subsequent reads.

Alternatively, if both requests received unsound responses (e.g. both received three negative acknowledgments), then the log replicas would be in an inconsistent state since a threshold of the log replicas state do not agree. When the log replicas are in an inconsistent state, new data cannot be added to a threshold of the log replicas. When no progress can be made, the replicas need to be *repaired* to a consistent state. The quorum repair protocol is discussed next.

4.3 Quorum Repair

The repair protocol restores log replicas to a consistent state such that the latest sound write is the last write stored by a threshold of log replicas. It may be used when a client cannot make progress because replicas of the log are in an inconsistent state or a quorum is not available due to server failures. Figure 8 shows the repair process.

When a storage server receives a *repair audit*, it attempts to read the latest replicated state (latest sound write) from the other servers in the configuration. If a quorum responds and the data is in a consistent state, the storage server takes no action. If, however, a quorum does not respond or the replicas are in an inconsistent state (wedged), then the storage server will create a repair request, record it in local stable storage, and submit it to the administrator. If the server observes that it already stores a signed repair request

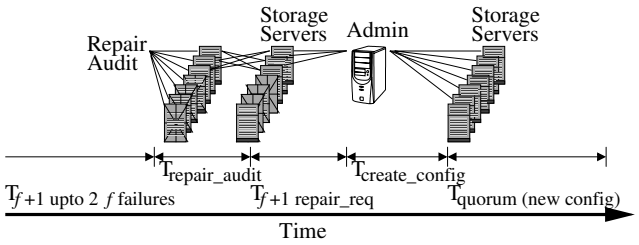


Figure 8: When a storage server believes that repair is needed, it sends a request to the administrator. After the administrator receives at least $f+1$ requests from servers in the current configuration, it creates a new configuration and sends message to servers in the set. The message describes the current state of the log; storage servers fetch the log from members of the previous configuration.

(a) **Soundness proof contents**

cert	certificate
config	configuration
ss_sigs[]	$2f+1$ or more signatures $<$H(cert+config)$>_{ss_priv}$

(b) **Configuration contents**

object_id	cryptographically secure name of object
client_id	hash of client's public key: H(PK)
ss_set[]	set of storage servers: set of H(ss_PK)
f	fault servers tolerated
seq_num	configuration sequence number
timestamp	creation time of configuration
ttl	time the configuration remains valid

Table 3: (a) A soundness proof can be presented by any machine to any other machine in the network to prove that a write was sound. To provide this guarantee, the proof contains a set of q storage server signatures over an append's certificate (Table 2) and the storage configuration (Table 3(b)). (b) A configuration defines a set of storage servers that maintain a replicated log.

on its local disk, it will forward the same request to the administrator. Once a storage server is in the repair state, it does not accept updates until a new configuration is created.

Martin and Alvisi demonstrated that when the administrator receives $2f+1$ repair requests, it can create a new configuration to host the log while maintaining consistency (the latest sound write) between the old and new configurations [29]. Servers in the new configuration fetch the state from servers in the previous configuration. The administrator can reduce the amount of data that must be transferred during repair by retaining servers across configurations. After acquiring a copy of the log, a storage server in the new configuration responds to the administrator with a signature over the certificate (from the latest sound write) and the new configuration. The repair protocol is done after the administrator receives a quorum of responses from servers in the new configuration.

The Martin and Alvisi protocol can invoke a repair protocol when a quorum of servers is available. This, however, means that repair cannot be initiated when it is needed most—when less than a quorum of servers are available. Essentially, a quorum in the old configuration is required to agree to trigger a repair protocol that will create a new configuration. We adapted the protocol to allow repair to be triggered when less than a quorum is available while still maintaining consistency.

To ensure that no successful (sound) writes are lost during repair when less than a quorum is available, we base the repair protocol on a data structure called a *soundness proof*. Table 3(a) shows the contents of a soundness proof. A soundness proof includes a certificate (Table 2), a configuration (Table 3(b)), and a quorum q of server signatures over a hash of the certificate and configuration. It can be stored by and presented to any server as proof that a write was sound.

The new protocol is similar to the old, except the new protocol requires each server to store a soundness proof for successful operations and include the latest soundness proof in a repair request. We describe the base write protocol that creates the soundness proof below. The administrator then uses the latest soundness proof from the received repair requests to create a new configuration and initialize the configuration to the latest sound write.

To create soundness proofs required for repair, the base write protocol works as follows. There are two rounds; however, the second round is often sent with a subsequent operation. The client library does not report success to the application until the second round completes successfully. First, a client submits a request to the storage servers. When a storage server receives the message, it checks the request against its local state. If the request satisfies all conditions, the server stores the data to non-volatile storage and responds to the client with a signed positive acknowledgment. The client combines signed positive acknowledgments from a quorum of servers to create a soundness proof. Next, in the second round, the client sends the soundness proof to the servers, often as part of a subsequent operation. Each server stores the soundness proof to a stable storage and responds to the client. The client can be certain the log has been written successfully after sending the soundness proof to all servers and receiving responses from a quorum of servers [45].

4.4 Utilizing Distributed Hash Table (DHT) Technology for Data Maintenance

Antiquity uses distributed hashtable (DHT) technology to underly and connect the storage servers. It uses the DHT as a distributed directory; that is, the DHT does not store data, but rather it stores pointers that identify servers that store the data. A distributed directory provides a level of indirection that allows flexible data placement which can increase the durability and decrease the cost of repairing a given replica [8, 44]. The storage servers use the distributed directory to publish and locate extents and other storage servers. The storage servers also use the DHT in the traditional manner to cache soundness proofs to ensure they are available for all interested parties.

Antiquity also relies on the DHT to help monitor server liveness to determine when repair is necessary. Using a DHT to monitor the liveness of each extent separately is not efficient. Instead, Antiquity uses the DHT to monitor server availability and uses that metric as a proxy for extent availability. To monitor server availability, Antiquity periodically broadcasts a heartbeat message through a spanning tree defined by the DHT's routing tables [8]. A monitoring node receives liveness information from each node with a frequency depending on its distance in the spanning tree. If it fails to receive an expected heartbeat. it sends a repair audit.

Each server in Antiquity also serves as a *gateway*. A gateway accepts requests from a client and works on behalf of that client, determining the configuration to handle the request and multicasting the request to the appropriate storage servers. The use of a gateway lowers the bandwidth requirements of the client. Because all requests are signed and all data is self-verifying, inserting the gateway in the path between the client and the storage servers does not affect security. If the client believes a failure is due to a faulty gateway, it can resend the request through a different gateway. To make the soundness proof available to storage servers earlier, the gateway combines responses from the storage servers to create a proof and publishes that proof in the DHT.

Gateways perform other tasks to reduce load on the administrator. Gateways propose configurations; the administrator needs only to verify the configuration before signing it. Currently, Antiquity uses neighbor lists from the underlying DHT to determine configurations. To limit the number of configuration queries that an administrator must handle, other machines in the system can cache valid configurations. The administrator forwards its message to the gateway that handles the new configuration request, and the gateway multicasts the message to servers in the new configuration.

4.5 Discussion

Maintaining the consistency of data replicated across the wide-area is a challenging endeavor. Using a secure log structure and interface, however, significantly reduces the complexity of the design.

1. Most of the log is immutable and stored in hash-verified extents. The order and data integrity of those extent replicas are immediate—the extent name verifies both the order and content of a hash-verified extent. It is not possible for any server to corrupt a hash-verified extent in an undetectable manner.

2. The log head is the only extent in each log that is mutable and key-verified. The order and data integrity of the log head can be verified using the verifier contained in the certificate. This verifier ensures the order and data integrity of the entire log. There is only one sequence of appends that results in a particular verifier. The verifier provides a "natural" predicate that can be used to ensure the consistency of a log. Each storage server checks that the predicate verifier matches local state before applying any operation.

3. The secure log structure reduces the complexity of maintaining sound writes over time. Storage servers need to maintain only the latest soundness proof because all previous writes contribute to the verifier of the current state of the log. During a transient failure, a server needs only to retrieve soundness proofs from other servers. Once the server determines the latest sound write, it can then fetch any blocks that it is missing from an up-to-date server.

In summary, the secure log structure and its interface simplify the Antiquity design by reducing complexity of managing integrity and consistency.

5. EVALUATION

In this section, we evaluate the Antiquity design using a prototype running on PlanetLab and a local cluster. We focus our evaluation on the primitive operations provided by the storage system, but we also describe our experiences with a versioning archival backup application.

5.1 Experimental Environment

The Antiquity prototype is written in Java using an event-driven programming style. It uses the Bamboo distributed hashtable [39] to locate storage servers and extents.

We are currently running two separate Antiquity deployments. Both deployments are configured to replicate each extent on a configuration of seven storage servers. Thus, each deployment can

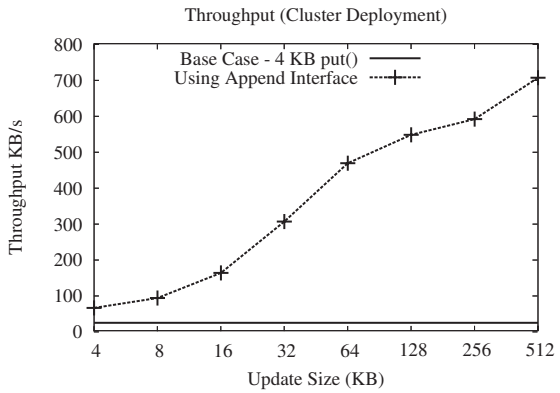


Figure 9: Aggregation increases system throughput by reducing computation at the client and in the infrastructure. The base case shows the throughput of a client that stores 4 KB blocks (and a certificate) using `put()` operation, as in a traditional DHT.

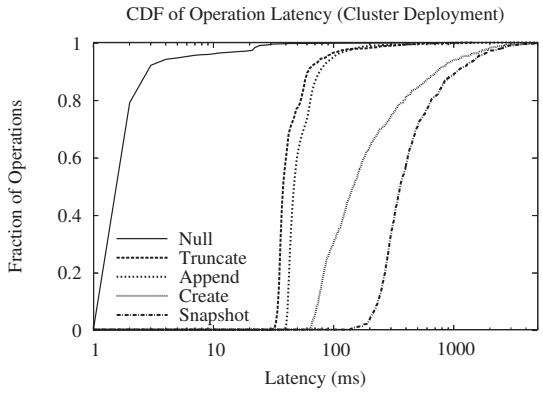


Figure 10: Different operations have widely varying latency. The latency is dependent on the amount of data that must be transferred across the network and the amount of communication with the administrator required. The latency CDF of all operations (even the `null()` RPC operation) exhibit a long tail due to load from other, unrelated jobs running on the shared cluster.

tolerate *two* faulty servers in each configuration. Both deployments are hosted on machines shared with other researchers, and, consequently, performance can vary widely over time.

The first deployment runs on 60 nodes of a local cluster. Each machine in the *storage cluster* has two 3.0 GHz Pentium 4 Xeon CPUs with 3.0 GB of memory and two 147 GB disks. Nodes are connected via a gigabit Ethernet switch. Signature creation and verification routines take an average of 3.2 and 0.6 ms, respectively. This cluster is a shared site resource; a load average of 10 on each machine is common.

The other deployment runs on the *PlanetLab* distributed research test-bed [5]. We use 400+ heterogeneous machines spread across most continents in the network. While the hardware configuration of the PlanetLab nodes varies, the minimum hardware requirements are 1.5 GHz Pentium III class CPUs with 1 GB of memory and a total disk size of 160 GB; bandwidth is limited to 10 Mbps bursts and 16 GB per day. Signature creation and verification take an average of 8.7 and 1.0 ms, respectively. PlanetLab is a heavily-contended resource; the average elapsed time of the cryptographic computations can be more than 210 and 10 ms.

We apply load to these deployments using 32 nodes of a different local cluster. Each machine in the *test cluster* has two 1.0 GHz Pentium III CPUs with 1.0 GB of memory and two 36 GB disks. Signature creation and verification takes an average of 6.0 and 0.6 ms, respectively. The cluster shares a 100 Mbps link to the external network. This cluster is also a shared site resource, but its utilization is lower than the storage cluster.

Parts of the evaluation have been presented in earlier work. In particular, the cluster deployment improves upon the preliminary performance presented by Eaton et al. [14]. Together, the performance and deployment evaluations demonstrate the efficacy of a system such as Antiquity.

5.2 Cluster Deployment

In addition to serving as a tool for testing and debugging, the Antiquity deployment on the cluster also allows us to observe the behavior of the system when bandwidth is plentiful and contention for the processor is relatively low.

Figure 9 shows how aggregation improves write performance. In this test, a single client submits synchronous updates of various sizes to Antiquity. The client library translates the requests into `append()`, `snapshot()`, and `truncate()` commands. We record the write throughput observed by the client. The x-axis shows how much data the client writes with each request. The extent capacity is set to 1 MB. For comparison, we show the throughput of a client that stores data using synchronous `put()` operations with payload of 4 KB of application data, as in typical in a DHT.

Aggregation increases system throughput. At the client, aggregation reduces the cost of interacting with the system by amortizing the cost of creating and signing certificates and transmitting network messages over more data. In the storage system, aggregation reduces the number of quorum operations that must be performed to write a given amount of data to the system. The `put()` throughput is lower than `append()` operations of equivalent size because `put()` operations require the administrator to create a new configuration for each request.

Next, we measure the latency of individual operations. In this test, a single data source issues a variety of operations, including incremental writes of 32 KB using the `append()` interface. Extents are configured to have a maximum capacity of 1 MB. Figure 10 presents a Cumulative Distribution Function (CDF) of the latency of various operations. The latency of different operations varies significantly. The `append()` and `truncate()` operations are the fastest because they transfer little or no data and do not require any interaction with the administrator. The `create()` operation is slightly slower because, though it contains no application payload, it must contact the administrator to obtain a new configuration. Finally, the `snapshot()` operation is the slowest; it transfers large amounts of data and must contact the administrator to find a suitable configuration of storage servers. The latency distribution of all operations exhibit a long tail due to load from other unrelated processes running on the same machines; note, even the `null()` RPC call can take longer than one second, due to delay caused by load from unrelated jobs running on the cluster.

Table 4 decomposes the median latency into its component phases for different types of operations. Notice that interacting with the DHT consumes a significant fraction of time (`publish()` and `lookup()` are DHT operations). In particular, `append()` and `truncate()` interact with the DHT one time to `publish()` the soundness proof to support repair. However, operations that create extents (`create()` and `snapshot()`) interact with the DHT multiple times (locate/publish coordinating gateway, lookup

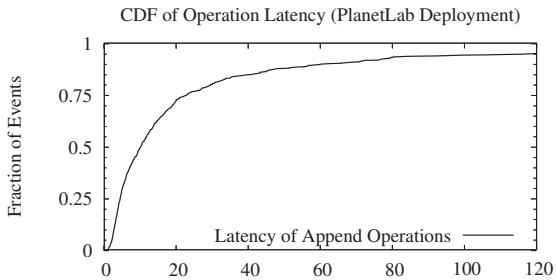
Phase	Time (ms) No Admin trunc	No Admin append	Admin create	Admin snapshot
T_{req}				
Signs Req	6.0	6.0	6.0	6.0
Send Req	1.8	4.2	1.8	1.8
Verify Req	0.6	1.0	0.6	0.6
`lookup()` Locations	(cached)	(cached)	13.2	13.2
`publish()` Gateway	(cached)	(cached)	7.2	7.2
subtotal	8.4	11.2	28.8	28.8
T_{create_config}				
`lookup()` Neighbors	N/A	N/A	6.6	6.6
Send Config Req	N/A	N/A	1.6	1.6
Verify Config Req	N/A	N/A	0.6	0.6
Create New Config	N/A	N/A	8.2	8.2
Sign New Config	N/A	N/A	3.2	3.2
Reply w/ New Config	N/A	N/A	1.6	1.6
subtotal	0.0	0.0	21.8	21.8
T_{quorum}				
Send Req	1.8	6.6	1.8	1.8
Verify Req	0.6	1.0	1.2	1.2
Fetch Extent				98.4
Disk	4.1	5.9	4.1	61.9
`publish()` Location			7.2	62.3
Sign Result	3.2	3.2	3.2	3.2
Send Reply	1.6	1.6	1.6	1.6
Verify Replies	4.2	4.2	4.2	4.2
`publish()` Proof	7.2	7.2	63.3	63.3
subtotal	22.7	29.7	86.6	297.9
T_{resp}				
Reply w/ Proof	1.7	1.7	1.7	1.7
Verify Proof	4.2	4.2	4.2	4.2
subtotal	5.9	5.9	5.9	5.9
Total – Median	37.0	46.8	143.1	354.4
(Min)	(31.0)	(38.0)	(62.0)	(137.0)

Table 4: Measured breakdown of the median latency times for all operations. The average network latency and bandwidth between applications on the test cluster and storage cluster is 1.7 ms and 12.5 MB/s (100 Mbs), respectively. The average latency and bandwidth between applications within the storage cluster is 1.6 ms and 45.0 MB/s (360 Mbs). All data is stored to disk using BerkeleyDB which has an average latency and bandwidth of 4.1 ms and 17.3 MB/s, respectively. Signature creation/verification takes an average of 6.0/0.6 ms on the test cluster and 3.2/0.6 ms on the storage cluster. Bandwidth of the SHA-1 routine on the storage cluster is 80.0 MB/s. Finally, DHT `lookup()` /`publish()` take an average of 4.2/7.2 ms.

replica locations, publish replica location, and publish soundness proof). Furthermore, multiple `publish()` operations to the same identifier often take longer than expected since `publish()` sometimes competes with other BerkeleyDB operations for use of the disk (e.g. BerkeleyDB log cleaning).

5.3 PlanetLab Deployment

In this section, we present results from the Antiquity deployment on PlanetLab. For reasons illustrated in Figure 11, the focus of our evaluation of the PlanetLab deployment is not on its performance. That graph plots the CDF of the latency of more than 800 operations that append 32 KB of data to logs in the systems. The accompanying table reports several key points on the curve. Given the best of circumstances, the latency of an `append()` operation



Min	25%	50%	90%	95%	99%	Max
1.08	4.41	10.2	63.1	124.5	302.1	615.9

Figure 11: The latency of operations on PlanetLab varies widely depending on the membership and load of a configuration. As an example, this graphs illustrates the CDF of the latency for appending 32 KB to logs stored in the system. The table highlights key points in the curves.

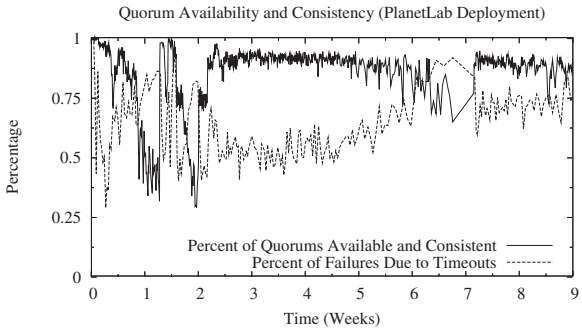
is one second. However, when configurations include distant or overloaded servers or bandwidth is restricted on some path, the latency increases considerably. Because of the characteristics of the PlanetLab testbed, many operations are very slow [37].

Instead, with the PlanetLab deployment, we focus on how the design maintains data over time, especially as machines fail. We built a simple test application that writes logs to the system and periodically reads them to check that they are still available. Each log consists of one key-verified extent (the log head) and an average of four hash-verified extents (the number of hash-verified extents vary uniformly with an average of four). Key-verified extents vary in size uniformly up to 1 MB; all hash-verified extents are 1 MB. The average size of a log is 4.5 MB (0.5 MB log head and 4 x 1MB hash-verified extents). The test application stores 18,779 logs (18,779 log heads and 75,085 hash-verified extents) totaling 84 GB. We stopped writing new data to Antiquity because we reached the PlanetLab-enforced storage quota. After writing an extent to the system, this test application records a summary of the extent in a local database.

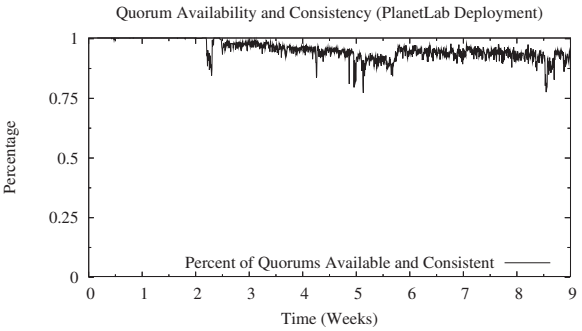
We perform various tests to measure the efficacy of the Antiquity deployment. First, we measure the percent of extents with at least a quorum of replicas available and in a consistent state in Section 5.3.1. This test measures the number of logs that can accept new writes from their owner. Next, in Section 5.3.2, we measure the cost of maintaining secure logs in terms of replicas created. In particular, we measure the average number of replicas created per unit time and the total number of replicas created. This test measures the systems ability to maintain sufficient replication levels in response to server failure.

5.3.1 Quorum Consistency and Availability

We compute quorum availability and consistency in two different ways. The first approach uses a test application that periodically reads a random extent. Every 10 seconds, the tester selects a random entry from the database and attempts to contact a quorum of the servers hosting that extent. It reports whether it was able to reach a quorum of servers. It also verifies that the replicas are in a consistent state and that state matches what was written. The second approach uses a server application availability trace, server database log, and extent configuration to compute the metrics. The first approach is an experimental method that includes intermittent effects such as server load and network performance. The sec-



(a) Periodic Application Read



(b) Server Application Availability Trace

Figure 12: Quorum Consistency and Availability. (a) Periodic reads show that 94% of quorums were reachable and in a consistent state. Up to 90% of failed checks are due to network errors and timeouts. (b) Server application availability trace shows that 97% of quorums were reachable and in a consistent state. This illustrates the increase in performance over (a) where timeouts reduced the percent of measured available quorums.

ond approach ignores such experimental effects and, instead, uses server availability to compute the metrics.

Figure 12(a) shows the percentage of quorums that were available and consistent over time, as measured by the first approach over a two-month period. The top curve shows the percentage of successful quorum checks. A software bug between week 1 through the middle of week 2 caused over half the servers not to respond to RPC requests. Periodic server application reboot temporarily masked the bug. But the performance continued to degrade until the problem was solved during the middle of week 2. A stale network file system handle prevented the test application from probing the system properly between weeks 6 and 7. Over the life of the test (including the period between week 1 through the middle of week 2 interruption), an average of 94% of checks reported that a quorum of servers was reachable and stored a consistent state. Even though, at any given time, 6% of the of the checks may not have a quorum of replicas available, later checks reveal that a quorum eventually becomes available and is consistent due to the repair protocol.

The observed availability matches computed estimates. Using a monitor on the remote hosts, we measured the average availability of machines in PlanetLab to be 90%. Note, this figure indicates that the node is up, not necessarily that the node can be reached over the network. Given that measurement, we would expect a quorum of servers to be available 94% of the time.

The lower curve on the plot shows the percentage of checks that

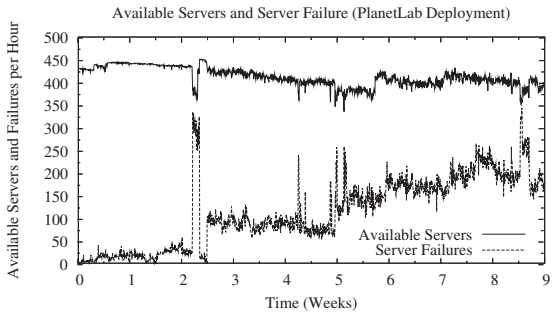


Figure 13: Number of servers with their Antiquity application available per hour. Additionally, number of servers with Antiquity application failures per hour. Most failures are due to restarting the unresponsive Antiquity instances. As a result, a single server may restart its Antiquity application multiple times per hour if the instance is unresponsive.

failed due to RPC failures, network disruptions, and other timeouts. We attempt to reach a quorum through five different gateways before marking a check failed. Our measurements show that up to 90% of the failed checks may be caused by components outside of Antiquity. This percentage increases as the load on PlanetLab increases. Furthermore, the high load causes a number of Antiquity processes to be terminated due to resource exhaustion. Thus, the actual percentage of consistent quorums (shown next) is higher than the 94% measured from the application.

Figure 12(b) plots quorum availability and consistency computed by the second approach. The server application availability trace used in this approach ignores the software bug; as a result, until the middle of week 2, 100% of the extents had at least a quorum available and consistent. After the middle of week 2, however, server churn increased, tripling from 24 server failures per hour to 76. The cause for the increase in server churn is a watchdog timer that restarts a server's Antiquity application when it is unresponsive for over six minutes. Figure 13 shows the number of servers available and server failures during each hour of the test.

5.3.2 Quorum Repair

Antiquity's repair process maintains the availability of a quorum of servers for each extent. Figure 14 plots the cumulative number of replicas created in the PlanetLab deployment. During the period of observation, Antiquity initially created a total of 657,048 extent replicas (each of the 93,864 extents were initially created with 7 replicas). The replicas initially accounted for 577 GB of replicated storage (84 GB of unique storage).

In order to maintain the availability of a quorum of servers, Antiquity triggers the `repair()` protocol when less than a quorum of replicas are available. Each `repair()` replaces at least three replicas since that is the least number of unavailable servers required to trigger `repair()` with $f = 2$. The deployment experienced an average of 114 (Antiquity application) failures per hour. In response to failures, Antiquity triggered `repair()` 92 times per hour. As the number of unavailable servers accumulated, nearly every failure triggered a `repair()`. Each `repair()` replaced an average of four replicas. As a result, Antiquity created a total of 653,028 replicas due to `repair()` during the two month period of observation. Repair required less than 0.31 KB/s (320 Bps) per server. Coupled with maintaining the availability and consistency of up to 97% of the extents, this demonstrates that Antiquity is capable of maintaining sufficient replication levels in response to

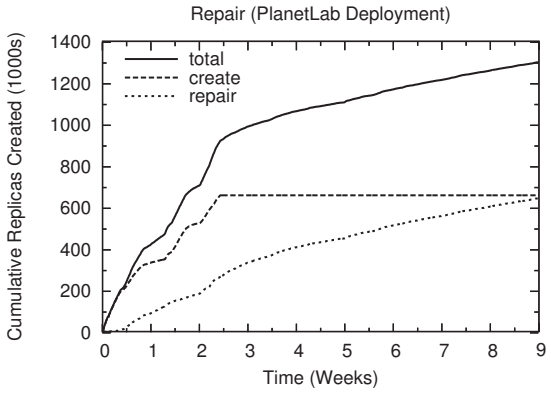


Figure 14: Number of replicas created over time due to storing new data and in response to failure.

server failure.

Another metric of concern is the time to repair an extent with less than a quorum of replicas available. On average, when a server failed, it took the system 30 minutes to detect and classify the server as failed (value of timeout) and three hours to replace replicas stored on the failed server with less than a quorum of remaining replicas available. Once repair completed for a particular extent, at least a quorum of servers were again available.

5.4 A Versioning Back-up Application

Finally, we have built a versioning back-up application that stores data in Antiquity. The application translates a local file system into a Merkle tree as shown in Figure 5 and used in similar previous systems [32, 11]. The application records in a local database when data was written to the infrastructure. It checks the local database before archiving any new data. This acts as a form of copy-on-write, reducing the amount of data transmitted. The file system we implemented is described fully in [15].

We stored the file system containing the Antiquity prototype (source code, object code, utility scripts, etc.) in PlanetLab. The file system is recorded in 15 1-MB extents. The system has repaired two of the 15 extents while ensuring both consistency and durability of the file system.

6. EXPERIENCE AND DISCUSSION

Putting it all together, Antiquity maintained 100% durability and 97% quorum availability of 18,779 logs broken into 93,864 extents. Reflecting on our experience, the structure of the secure log made this an easier task for three reasons.

First, maintaining the integrity of a secure log is easier than other structures since the verifier for the log (and each extent) defines the order of appends and cryptographically ensures the content. In particular, there is only one sequence of appends that results in a particular verifier. This verifier is used as a predicate to ensure that new writes are appended to the log in a consistent fashion. Furthermore, this verifier is used by the storage system to ensure that each replica stores the same state. In the deployment, this verifier was a critical component used to ensure the consistency and integrity of the log and all of its extents. Furthermore, it is cheap to compute, update, and compare.

Second, a storage system that implements a secure log is a layer or middleware in a larger system. The secure log abstraction bridges the storage system and higher level applications together. In fact, the secure log interface implemented by Antiquity is a result of breaking OceanStore into layers. In particular, a component of OceanStore was a primary replica implemented as a Byzantine Agreement process. This primary replica serialized and cryptographically signed all updates. Given this total order of all updates, the question was how to durably store and maintain the order? Furthermore, what should be the interface to this storage system? An append-only secure log answered both questions. The secure log structure assists the storage system in durably maintaining the order over time. The append-only interface allows a client to consistently add more data to the storage system over time. Finally, when data is read from the storage system at a later time, the interface and protocols ensure that data will be returned and that returned data is the same as stored.

Finally, self-verifying structures such as a secure log lend themselves well to distributed repair techniques. The integrity of a replica can be checked locally or in a distributed fashion. In particular, we implemented a quorum repair protocol where the storage server replicas used the self-verifying structure. The structure and protocol provided proof of the contents of the latest replicated state and ensured that the state was copied to a new configuration.

7. RELATED WORK

Antiquity builds on the experience of many prior systems.

7.1 Logs

The log-structured file system [30] used a log abstraction to improve the performance of local file systems. Zebra [20] uses a similar abstraction to improve the performance of a network file system. Schneier and Kelsey [41] and SUNDR [24] demonstrated how to use a secure log to store data on an untrusted remote machines. They do not address how to replicate the log.

7.2 Byzantine Fault-Tolerant Services

Byzantine fault-tolerant services have been proposed to help meet the challenges of unsecured, distributed environments. FarSite [2], OceanStore [38], and Rosebud [40] built distributed storage systems using Byzantine fault-tolerant agreement protocols [21, 7]. Abd-El-Malek et al. [1], Goodson et al. [18], the COCA project [46], Fleet [27], and Martin and Alvisi [29] built reliable services using Byzantine fault-tolerant quorum protocols [26]. Martin and Alvisi define a protocol that allows the configuration to be changed with the help of an administrator. HQ [10] has a hybrid structure that is similar to Antiquity's use of an administrator. During normal operation, clients interact using an efficient Byzantine quorum voting protocol. Under write contention or failures, a separate Byzantine agreement (or administrator for Antiquity) is invoked to resolve conflicts and possibly reconfiguring the set of servers. None of these systems trigger reconfiguration reactively.

7.3 Wide-area Distributed Storage Systems

Many researchers have used distributed hash table (DHT) technology to build wide-area distributed storage systems. Notable examples are Carbonite [8], CFS [11], Glacier [19], Ivy [34], PAST [13], Total Recall [6], and Venti [36]. Carbonite and Total Recall optimize for the wide-area by reducing the number of replicas created due to transient failures. Glacier uses aggregation to reduce storage overheads. Ivy uses a log structure similar to Antiquity; however, the log is block-based instead of extent-based. In particular, to grow the log, Ivy creates new blocks that incur high overheads since each block is individually maintained; whereas,extents reduce these overheads since Antiquity supports aggregation via a secure-append operation. None of these systems

implement a Byzantine fault-tolerant consistency algorithm. Chain Replication [44] and Etna [33] both implement consistency protocols, but assume fail-stop failures.

7.4 Replicated Systems

Systems like GFS [17], Harp [25], Petal [22], Frangipani [43], and XFS [4] replicate data to reduce the risk of data loss. GFS and XFS also use aggregation. These systems target well-connected environments.

Distributed databases [12], the Amoeba distributed operating system [42], the Myriad online disaster recovery system [23], and EMC storage systems [9] use the wide-area replication to increase durability. Myriad and EMC replicate data between a primary and backup site. Wide-area recovery is initiated after site failure; single disk failure is repaired locally with RAID.

7.5 Digital Libraries

Digital libraries such as LOCKSS [28] preserve journals and other electronic documents. The documents are read-only and cannot be updated. They are replicated at many sites for durability. Many documents in a digital library do not have an "owner"; thus, the system uses voting to maintain the integrity. Antiquity, in contrast, assumes that all logs have an owner and only the owner can make changes to the log.

8. CONCLUSION

We described the design of the Antiquity wide-area distributed storage system. The design is tailored for dynamic environments where server failure is common. Antiquity combines a secure append-only log interface with dynamic Byzantine fault-tolerant quorums and quorum repair to maintain data integrity, consistency, and durability. Evaluation of a prototype running on PlanetLab demonstrates that the design is effective. The prototype stores 84 GB of data on 400+ servers under constant churn and all logs remain durable. At any given moment, however, 6% of the logs do not have a quorum (threshold) of replicas available temporarily due to server failure on PlanetLab. All eventually become available—Antiquity successfully repaired all quorums to an available and consistent state with its quorum repair protocol.

9. AVAILABILITY

The Antiquity source code is published under the BSD license and is freely available http://antiquity.sourceforge.net.

10. ACKNOWLEDGMENTS

We would like to thank Ken Birman whose comments and advice have greatly improved the presentation of this work. We are grateful to Anthony Joseph who has provided valuable input on the design and implementation of Antiquity. Also, Robbert van Renesse and Einar Vollset have provided valuable feedback on the presentation of Antiquity. Finally, we would like thank Mike Howard for maintaining the cluster at Berkeley and all of the groups that have contributed to making PlanetLab available. Without these two testbeds, this work would not have been possible.

11. REFERENCES

[1] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable byzantine fault-tolerant services. In *Proc. of ACM SOSP*, Oct. 2005.

[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 OSDI*, Dec. 2002.

[3] N. S. Agency. Global information grid (gig). `http://www.nsa.gov/ia/industry/gig.cfm`. Last accessed September 2006.

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

[5] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, , and M. Wawrzoniak. Operating system support for planetary-scale network services. In *Proc. of NSDI*, Mar. 2004.

[6] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. Voelker. Totalrecall: Systems support for automated availability management. In *Proc. of NSDI*, Mar. 2004.

[7] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. of OSDI*, 1999.

[8] B. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient replica maintenance for distributed storage systems. In *Proc. of NSDI*, San Jose, CA, May 2006.

[9] E. Corp. Symmetrix remote data facility. `http://www.emc.com/products/networking/srdf.jsp`. Last accessed April 2006.

[10] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proc. of OSDI*, Nov. 2006.

[11] 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.

[12] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swindhart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proc. of ACM PODC*, pages 1 – 12, 1987.

[13] 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.

[14] P. Eaton, H. Weatherspoon, and J. Kubiatowicz. Efficiently binding data to owners in distributed content-addressable storage systems. In *3rd International Security in Storage Workshop*, Dec. 2005.

[15] P. R. Eaton. *Improving Access to Remote Storage for Weakly Connected Users*. PhD thesis, EECS Department, University of California, Berkeley, January 11 2007.

[16] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. In *Proc. of OSDI*, Oct. 2000.

[17] S. Ghemawat, H. Gobioff, and S. Leung. The google file system. In *Proc. of ACM SOSP*, pages 29–43, Oct. 2003.

[18] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Byzantine-tolerant erasure-coded storage. Technical Report CMU-CS-03-187, Carnegie Mellon University School for Computer Science, Sept. 2003.

[19] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proc. of NSDI*, May 2005.

[20] J. H. Hartman and J. K. Ousterhout. The zebra striped network file system. In *Proc. of ACM SOSP*, 1993.

[21] L. Lamport, R. Shostak, and M. Pease. The Byzantine

Generals Problem. *ACM TOPLAS*, 4(3):382–401, 1982.

[22] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proc. of ASPLOS*, pages 84–92, 1996.

[23] S. A. Leung, J. MacCormick, S. E. Perl, and L. Zhang. Myriad: Cost-effective disaster tolerance. In *Proc. of USENIX FAST*, Jan. 2002.

[24] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (sundr). In *Proc. of OSDI*, pages 121–136, Dec. 2004.

[25] 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.

[26] D. Malkhi and M. Reiter. Byzantine quorum systems. In *Proc. of ACM STOC*, pages 569 – 578, May 1997.

[27] D. Malkhi, M. K. Reiter, D. Tulone, and E. Ziskind. Persistent objects in the fleet system. In *DISCEX II*, 2001.

[28] P. Maniatis, M. Roussopoulos, T. Giuli, D. S. H. Rosenthal, and M. Baker. The lockss peer-to-peer digital preservation system. *ACM Trans. Comput. Syst.*, 23(1):2–50, 2005.

[29] J.-P. Martin and L. Alvisi. A framework for dynamic byzantine storage. In *Proc. of the Intl. Conf. on Dependable Systems and Networks*, June 2004.

[30] J. Matthews, D. Roselli, A. Costello, R. Wang, and T. Anderson. Improving the performance of log-structured file systems with adaptive methods. In *Proc. of ACM SOSP*, Oct. 1997.

[31] R. Merkle. A digital signature based on a conventional encryption function. In *Proc. of CRYPTO*, pages 369–378. Springer-Verlag, 1988.

[32] S. J. Mullender and A. S. Tanenbaum. A distributed file service based on optimistic concurrency control. In *Proc. of ACM SOSP*, pages 51–62, Dec. 1985.

[33] A. Muthitacharoen, S. Gilbert, and R. Morris. Etna: A fault-tolerant algorithm for atomic mutable dht data. Technical Report MIT-LCS-TR-993, MIT Laboratory for Computer Science, June 2004.

[34] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proc. of OSDI*, 2002.

[35] L. Peterson, A. B. E. Fiuczynski, , and S. Muir. Experiences building planetlab. In *Proc. of OSDI*, Nov. 2006.

[36] S. Quinlan and S. Dorward. Venti: A new approach to archival data storage. In *Proc. of USENIX FAST*, Jan. 2002.

[37] S. Rhea, B. Chun, J. Kubiatowicz, and S. Shenker. Fixing the embarrassing slowness of opendht on planetlab. In *Proc. of USENIX Workshop on Real, Large Distributed Systems (WORLDS)*, Dec. 2005.

[38] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the OceanStore prototype. In *Proc. of USENIX FAST*, 2003.

[39] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a dht. In *Proc. of USENIX*, June 2004.

[40] R. Rodrigues and B. Liskov. Rosebud: A scalable byzantine-fault-tolerant storage architecture. Technical Report MIT-LCS-TR-932, MIT Laboratory for Computer Science, Dec. 2003.

[41] B. Schneier and J. Kelsey. Cryptographic support for secure logs on untrusted machines. In *Proc. of USENIX Annual Technical Conf.*, Jan. 1998.

[42] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, 1990.

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

[44] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proc. of OSDI*, May 2004.

[45] H. Weatherspoon. *Design and Evaluation of Distributed Wide-Area On-line Archival Storage Systems*. PhD thesis, EECS Department, University of California, Berkeley, October 13 2006.

[46] L. Zhou, F. Schneider, and R. van Renesse. Coca: A secure distributed on-line certification authority. *ACM Transactions on Computer Systems*, pages 329–368, Nov. 2002.