A protocol family for versatile survivable storage infrastructures

Garth R. Goodson, Jay J. Wylie, Gregory R. Ganger, Michael K. Reiter

CMU-PDL-03-103
December 2003

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

Survivable storage systems mask faults. A protocol family *shifts the decision of which types of faults from implementation time to data-item creation time. If desired, each data-item can be protected from different types and numbers of faults. This paper describes and evaluates a family of storage access protocols that exploit data versioning to efficiently provide consistency for erasure-coded data. This protocol family supports a wide range of fault models with no changes to the client-server interface or server implementations. Its members also shift overheads to clients. Readers only pay these overheads when they actually observe concurrency or failures. Measurements of a prototype block-store show the efficiency and scalability of protocol family members.*

Acknowledgements: We thank the members and companies of the PDL Consortium (including EMC, Hewlett-Packard, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, Panasas, Seagate, Sun, and Veritas) for their interest, insights, feedback, and support. We thank IBM and Intel for hardware grants supporting our research efforts. This material is based on research sponsored by the Air Force Research Laboratory, under agreement number F49620-01-1-0433, and by DARPA/ITO's OASIS program, under Air Force contract number F30602-99-2-0539-AFRL. Garth Goodson was supported by an IBM Fellowship.

Keywords: survivable storage, protocol family, Byzantine fault-tolerance, atomic registers, erasure codes

1 Introduction

Survivable, or fault-tolerant, storage systems protect data by spreading it redundantly across a set of storage-nodes. In the design of such systems, determining which kinds of faults to tolerate and which timing model to assume, are important and difficult decisions. Fault models range from crash faults to Byzantine faults [27] and timing models range from synchronous to asynchronous. These decisions affect the access protocol employed, which can have a major impact on performance. For example, a system's access protocol can be designed to provide consistency under the weakest assumptions (i.e., Byzantine failures in an asynchronous system), but this induces potentially-unnecessary performance costs. Alternatively, designers can "assume away" certain faults to gain performance. Traditionally, the fault model decision is hard-coded during the design of the access protocol.

This traditional approach has two significant shortcomings. First, it limits the utility of the resulting system—either the system incurs unnecessary costs in some environments or it cannot be deployed in harsher environments. The natural consequence is distinct system implementations for each distinct fault model. Second, all data stored in any given system implementation must use the same fault model, either paying unnecessary costs for less critical data or under-protecting more critical data. For example, temporary and easily-recreated data incur the same overheads as the most critical data.

This paper promotes an alternative approach, in which the decision of which faults to tolerate is shifted from design time to data-item creation time. This shift is achieved through the use of a *family* of access protocols that share a common server implementation and client-server interface. A *protocol family* supports different fault models in the same way that most access protocols support varied numbers of failures: by simply changing the number of storage-nodes utilized, and some read and write thresholds. A protocol family enables a given infrastructure of storage-nodes to be used for a mix of fault models and number of faults tolerated, chosen independently for each data-item.

This paper describes and evaluates a family of access protocols that exploit data versioning within storage-nodes to efficiently provide consistency for erasure-coded data. The protocol family covers a broad range of fault model assumptions (crash vs. Byzantine servers, crash vs. Byzantine clients, synchronous vs. asynchronous communication, client repairs of writes vs. not, total number of failures) with no changes to the client-server interface or server implementation. Protocol family members are distinguished by choices enacted in client-side software: the number of storage-nodes that are written and the logic employed during a read operation. Weaker assumptions require the use of more storage-nodes and additional computation for the client. For example, tolerating some number of Byzantine storage-nodes requires at least two times more storage-nodes than tolerating the same number of crash storage-nodes. Likewise making no timing assumptions requires approximately two times more storage-nodes than assuming synchrony. Regardless of the assumptions made, the server implementation does not change.

Each member of the protocol family works roughly as follows. To perform a write, a client sends time-stamped fragments to the set of storage-nodes. Storage-nodes keep all versions of fragments they are sent until garbage collection frees them. To perform a read, a client fetches the latest fragment versions from the storage-nodes and determines whether they comprise a completed write; usually, they do. If they do not, additional fragments or historical fragments are fetched, until a completed write is observed (or, some family members may abort). Only in certain cases of failures or concurrency are there additional overheads incurred to maintain consistency.

This protocol family is particularly interesting because it is efficient in three ways. First, all members support *m*-of-*n* erasure codes (i.e., any *m* of a set of *n* erasure-coded fragments can be used to reconstruct the data), which can tolerate multiple failures with less network bandwidth (and storage space) than replication [52, 53]. Second, most read operations complete in a single round trip: only reads that observe write concurrency or failures (of storage-nodes or a client write) may incur additional work. Most studies of distributed storage systems (e.g., [4, 39]) indicate that concurrency is uncommon (i.e., writer-writer and writer-reader sharing occurs in under 1% of operations). Failures, although tolerated, should also be rare. Moreover, a subsequent write effectively replaces the work-inducing state, thus preventing future reads from incurring additional costs. Third, most protocol processing is performed by clients, increasing scalability via the well-known principle of shifting work from servers to clients [22].

This paper details the implementation and performance of a storage infrastructure built using the protocol family. The infrastructure consists of a block-based interface and a single storage-node implementation. Such an infrastructure could be used directly by "storage-brick" based systems, such as Petal [28], FAB [11], and IceCube [35]. As expected, it scales well with the type and number of faults tolerated. In particular, the versioning avoids inter-server coordination, and the use of erasure codes avoids excessive write network bandwidth utilization. Concurrency and garbage collection overheads have minimal impact. Wide-area latencies increase response times, of course, but do not affect scalability regarding the type and number of faults tolerated.

The remainder of this paper is organized as follows. Section 2 discusses the protocol family concept and related work. Section 3 describes our protocol family and its membership. Section 4 explains what each protocol member provides and develops bounds, on, for example, the minimum number of storage-nodes required by each protocol member. Section 5 details how protocol members are realized within a common software implementation. Section 6 explores its performance characteristics.

2 Background and related work

Figure 1 illustrates the abstract architecture of a fault-tolerant, or survivable, distributed storage system. To write a data-item *D*, Client A issues write requests to multiple storage-nodes; each write request includes a replica or an erasure coded data-fragment, depending on the data distribution scheme used. To read *D*, Client B issues read requests to an overlapping subset of storage-nodes. This basic scheme provides access to data-items even when subsets of the storage-nodes have failed. To provide reasonable storage semantics, however, a system must guarantee that readers see consistent answers. For shared storage systems, this usually means linearizability [20] of operations.

Access protocols and consistency. Consistent access to survivable storage requires a protocol that addresses three sources of problems: access concurrency, storage-node failures, and client failures (resulting in partial or corrupt updates). Many protocols have been proposed, implemented, and used to address various mixes of these problems and assumptions about their characteristics.

Most storage systems assume benign crash failures by clients and servers, simplifying the problems significantly. Under such assumptions, access concurrency can be addressed via leases [16], optimistic concurrency control [26], or serialization through a primary (e.g., [29, 28]). Partial writes by clients that fail can be addressed by two-phase commit [17] or by post-hoc repair (in systems using replication).

Most systems implementing Byzantine fault-tolerant services adopt the state machine approach [46], wherein all operations are processed by all server replicas in the same order. Long believed to be too costly for extensive use in practice, this approach was recently employed by Castro and Liskov to implement a reasonably-performing replicated NFS service [6]. An alternative to replicated state machines is Byzantine quorum systems [30], which can also provide similar semantics [31]. These approaches linearize arbitrary operations, whereas our protocol family only linearizes read/write operations.

Previous protocol families. Our protocol family tolerates a hybrid failure model of storage-nodes (i.e., a mix of crash and Byzantine failures). The concept of hybrid failure models was introduced in [50]; other protocols have been developed for such failure models (e.g., [12] considers reliable broadcast, consensus and clock synchronization in the hybrid failure model).

Cristian et al. [7] systematically derive a logical family of atomic broadcast protocols for a range of fault models (from crash faults to a subset of Byzantine faults) in a synchronous environment. [7] uses

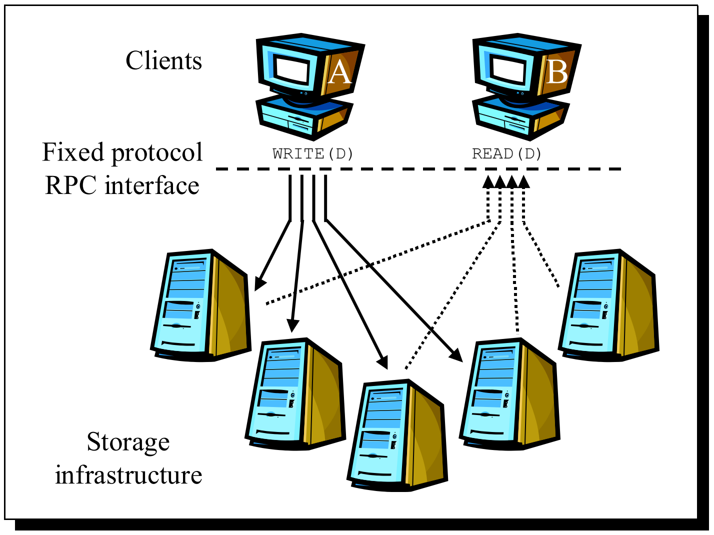

Figure 1: **High-level architecture for survivable storage.** In a survivable storage system, clients write and read data from multiple storage-nodes to achieve fault-tolerance. Our protocol family uses a fixed client–storage-node interface; consequently, different data-items stored within the same storage infrastructure can tolerate different types and numbers of failures.

the term "family" to refer to the logical construction of the protocols rather than their implementation. Members of our protocol family are realized in a common implementation as well as being logically related. Moreover, [7] only addresses synchronous protocols.

In [34], [7] is extended to an implementation for the timed asynchronous model [8]. Specifically, the Timewheel group communication system provides nine distinct semantics (one of three atomicity guarantees with one of three order guarantees) on a per broadcast granularity. It does not address Byzantine failures.

A framework for developing distributed services that are easily configured to handle different failures is developed in [21]. The framework is distinguished from Timewheel in that high-level protocols are built out of micro-protocols [5] that implement individual semantic properties (e.g., an atomicity guarantee or an ordering guarantee) or mask particular failure types. The framework provides a methodology for building micro-protocols, which are composed into high-level protocols, as well as a run-time system for distributed services.

Both Timewheel and the configurable framework provide a modular means of employing distinct protocols. Our protocol family is distinguished from these in that servers implement a fixed protocol regardless of the specific member being employed. Additionally, the majority of the client implementation is shared across protocol family members: different threshold parameters are required depending on the timing model and the number and types of storage-node failures tolerated; as well, a small amount of additional logic is required to tolerate Byzantine clients and perform client repair.

Additional related work. Our protocol family shares characteristics with quorum systems and exploits versioning servers. Both techniques are used in many ways by many systems. Here, we discuss a few.

Byzantine fault-tolerant protocols for implementing read-write objects using quorums are described in [19, 30, 32, 40]. Of these, Martin et al. [32] is closest to members of our protocol family in semantics and in that it uses a type of versioning. In our protocols, a read operation may retrieve data-fragments for several versions of the data-item in the course of identifying the return value of a read. Similarly, readers in [32] "listen" for updates (versions) from storage-nodes until a complete write is observed. Conceptually, our approach differs in that clients read past versions, rather than listening for future versions broadcast by servers. Our protocol family has several advantages over [32]: it works for erasure-coded data, it is more message efficient, it requires less computation ("listeners" require digital signatures in some fault models), and work is performed mostly by clients rather than servers.

We contrast our use of versioning to maintain consistency with systems in which each write creates a new, immutable version of a data-item (e.g., [36, 43]). Such systems shift consistency problems to the metadata mechanism that resolves data-item names to a version. So, systems that employ such an approach (e.g., Past [45] and CFS [9]) require a version tracking service to find the latest version. Our protocol family does not—indeed, these systems could use our access protocol family for such a service.

A full file service requires more general functions than read and write, such as "insert name into directory." As discussed earlier, this can be achieved with state machine replication. Some systems (e.g., Farsite [2] and OceanStore [25]) use replicated state machines for such metadata functions (including the tracking of the name–version mapping). Another option is to layer higher-level services, as needed, atop a base read/write service—for example, Frangipani [51] provides file services atop Petal [28]. Our protocol family is a versatile base read/write service, upon which file services and other storage services can be built.

Ivy [37] provides decentralized read/write access to immutable stored data in a fashion similar to some members of our protocol family. Per-client update logs (which are similar to version histories) are merged by clients at read time. Ivy differs from our protocols in that it does not provide strong consistency semantics in the face of data redundancy or concurrent updates.

We present, in detail, a single member of the protocol family (the asynchronous repairable protocol member with Byzantine clients and Byzantine storage-nodes) in [14]. In [14], we include proof sketches of that member's safety and liveness properties, as well as a favorable performance comparison (for read-write storage) relative to Byzantine tolerant replicated state-machines [6]. This paper goes beyond that paper by generalizing to the protocol family and exploring the result.

3 Protocol family

This section describes our protocol family for accessing erasure-coded data with versioning servers.

3.1 Overview

We describe each protocol family member in terms of N storage-nodes and an arbitrary number of clients. There are two types of operations — reads and writes. Each read/write operation involves read/write requests from a client to some number of storage-nodes. We assume that communication between a client and a storage-node is point-to-point, reliable (e.g., TCP), and authenticated.

At a high level, the protocol proceeds as follows. A data-item is *encoded* into data-fragments; any threshold-based erasure code (e.g., information dispersal [42], short secret sharing [24], or replication) could be used. Logical timestamps are used to totally order all write operations and to identify data-fragments from the same write operation across storage-nodes. For each correct write, a client constructs a logical timestamp that is guaranteed to be unique and greater than that of the *latest complete write* (the complete write with the highest timestamp). A write operation is defined as *complete* if sufficient storage-nodes have executed write requests to guarantee that no subsequent read operation can return a previously written value. Storage-nodes provide fine-grained versioning; a correct storage-node stores a data-fragment version (indexed by logical timestamp) for each write request it executes.

To perform a read operation, a client issues read requests to a set of storage-nodes. From the responses, the client identifies the *candidate*, which is the data-fragment version returned with the greatest logical timestamp. The read operation *classifies* the candidate as complete, incomplete or unclassifiable based on the number of read responses that share the candidate's timestamp. If the candidate is classified as *complete*, then the read operation is complete; the value of the candidate is returned. If it is classified as *incomplete*, the candidate is discarded, another read phase is performed to collect previous versions of data-fragments, and classification begins anew; this sequence may be repeated. If the candidate is *unclassifiable*, members of the protocol do one of two things: repair the candidate or abort the read operation.

3.2 Family membership

Each member of the protocol family is characterized by four parameters: the timing model, the storage-node failure model, the client failure model, and whether client repair is allowed. Eight protocol members result from the combination of these characteristics, each of which supports a hybrid failure model (crash and Byzantine) of storage-nodes.

Timing model. Protocol family members are either asynchronous or synchronous. Asynchronous members rely on no timeliness assumptions (i.e., no assumptions about message transmission delays or execution rates). In contrast, synchronous members assume known bounds on message transmission delays between correct clients/storage-nodes and their execution rates. As well, synchronous members require loosely synchronized clocks among clients and storage-nodes—protocols to achieve approximate clock synchronization in today's networks are well known, inexpensive, and widely deployed [33]. In an asynchronous system, storage-node crashes are indistinguishable from slow communication. In a synchronous system, storage-nodes that crash are detectable via timeouts—which provides useful information to the client.

Storage-node failure model. Family members are developed with a hybrid storage-node failure model [50]. Up to t storage-nodes may fail, $b \leq t$ of which may be Byzantine faults; the remainder can only crash. Such a model can be converted to a wholly crash (i.e., $b = 0$) or wholly Byzantine (i.e., $b = t$) model for storage-node failures. We assume that Byzantine storage-nodes can collude with each other and with any Byzantine clients.

Byzantine storage-nodes can corrupt their data-fragments. As such, it must be possible to detect and mask up to b storage-node integrity faults. Cross checksums [13] are used to detect corrupt data-fragments; this is described in Section 5.1.

Client failure model. Each member of the protocol family tolerates crash client failures and may additionally tolerate Byzantine client failures. Crash failures during write operations can result in subsequent read operations observing an incomplete or unclassifiable write operation. Readers cannot distinguish read-write concurrency from a crash failure during a write operation.

As in any general storage system, an authorized Byzantine client can write arbitrary values to storage, which affects the value of the data but not its consistency. Byzantine failures during write operations can additionally result in a write operation that lacks integrity; the decoding of different sets of data-fragments could lead to clients observing different data-items. Mechanisms for detecting any such write operation performed by a Byzantine client are described in Section 5.1. These mechanisms successfully reduce Byzantine actions to either being detectable or crash-like, allowing Byzantine clients to be tolerated without any change to the thresholds. Fine-grained versioning can facilitate detection, recovery, and diagnosis from storage intrusions [49].

Client repair. Each member of the protocol family either allows, or does not allow, clients to perform repair. Repair enables a client that observes an unclassifiable (i.e., *repairable*) candidate during a read operation to perform a write operation, which ensures that the candidate is complete, before it is returned.

In systems that differentiate write privileges from read privileges, client repair may not be possible. Non-repair protocol members allow read operations to abort. Reads can be retried at either the protocol or application level. At the protocol level, concurrency is often visible in the timestamp histories—an aborted read could be retried until a stable set of timestamps is observed. Other possibilities include requiring action by some external agent or blocking until a new value is written to the data-item (as in the "Listeners" protocol of Martin et al. [32]).

Protocol	**Asynchronous repairable**	**Asynchronous non-repair**	**Synchronous repairable**	**Synchronous non-repair**
N	$2t+2b+1 \leq N$	$3t+3b+1 \leq N$	$t+b+1 \leq N$	$t+2b+1 \leq N$
Q_C	$t+b+1 \leq Q_C$ $Q_C \leq N-t-b$	$t+b+1 \leq Q_C$ $Q_C \leq N-2t-2b$	$t+1 \leq Q_C$ $Q_C \leq N-b$	$t+1 \leq Q_C$ $Q_C \leq N-2b$
m	$1 \leq m \leq Q_C - t$	$1 \leq m \leq Q_C + b$	$1 \leq m \leq Q_C - t$	$1 \leq m \leq Q_C + b - t$
Complete	$\lvert CandidateSet \rvert \geq Q_C + b$		$\lvert CandidateSet \rvert \geq Q_C - f + b$	
Incomplete	$\lvert CandidateSet \rvert < Q_C - t$		$\lvert CandidateSet \rvert < Q_C - f$	

Table 1: **Protocol family constraint summary**

3.3 Protocol guarantees

The target safety property for the protocol family is *linearizability* [20]. Operations are linearizable if their return results are consistent with an execution in which each operation is performed instantaneously at a distinct point in time between its start time and its completion time. It is necessary to adapt linearizability for some members of the protocol family.

The use of clock synchronization by synchronous members introduces clock skew, which affects the definition of operation duration, and thus linearizability. Due to clock skew, two operations may have the same timestamps, even though they are not globally concurrent. Clock skew extends the definition of operation duration in synchronous members such that these operations are considered concurrent.

Byzantine clients may not follow the protocol execution. Read operations by Byzantine clients are excluded from the set of linearizable operations. Write operations are only included if they pass validation, which ensures they are well formed (see Section 5.1). Write operations by Byzantine clients do not have a well-defined start time and are thus concurrent to all preceding operations.

Non-repair protocol members allow reads to abort due to insufficient classification information: aborted reads are excluded from the set of linearizable operations. Such members achieve "linearizability with read aborts", which is similar to Pierce's "pseudo-atomic consistency" [40]. That is, the set of all write operations and all complete read operations is linearizable.

Every member of the protocol family has strong liveness properties. Operations performed by correct clients of synchronous repairable protocol members terminate. Operations performed by correct clients of asynchronous repairable protocol members are wait-free [18, 23]. Members that do not allow repair achieve similar properties to those that do; however, read operations may terminate by returning the "abort" value.

4 Protocol constraints

To guarantee that linearizability and liveness are achieved, a number of constraints must hold. For each member protocol, N, m, and the read classification rules are constrained with regard to b and t (from the hybrid model of storage-node failure). N is the number of storage-nodes in the system and m is the "decode" parameter of an m-of-n erasure code (note, n always equals N in our system).

This section develops four sets of constraints: one for each pair of the cross-product of asynchronous/ synchronous and repair/non-repair. The constraints are not affected by whether the failure model for clients is crash or Byzantine. A summary of the constraints for the protocol family is presented in Table 1. Proofs that the constraints provide the safety and liveness guarantees indicated, as well as precise definitions of those guarantees, are presented in [15].

4.1 Protocol properties

Constraints for each protocol family member are derived to satisfy a number of desired properties.

These properties are described using the following terminology. A client or storage-node is *correct* in an execution if it satisfies its specification throughout the execution; otherwise it *fails*. If it is either correct or only crashes, it is *benign*.

Since many of the properties involve read classification, recall that the candidate is the data-item version, returned by a read request, with the greatest logical timestamp. The set of read responses that share the candidate's timestamp are the *candidate set*.

The desired protocol properties are:

WRITE TERMINATION: This property ensures that all write operations can complete.

READ CLASSIFICATION: There are two classification rules: one to classify a candidate as complete and one to classify a candidate as incomplete. A candidate that is not classified by either rule is repairable if the protocol admits repair, and unclassifiable otherwise.

REAL UNCLASSIFIABLE/REPAIRABLE CANDIDATES: This property ensures that colluding Byzantine storage-nodes are unable to fabricate a candidate that a correct client deems unclassifiable/repairable.

CLASSIFIABLE COMPLETE CANDIDATES: This property is only necessary for non-repair protocol members; it ensures that Byzantine storage-nodes cannot make all read operations abort. Consider a write operation and a subsequent read operation performed in isolation by correct clients (i.e., the client does not fail and there is no concurrency). This property ensures that the read operation will return the value written by the write operation regardless of storage-node failures.

DECODABLE CANDIDATES: m must be constrained so that complete or repairable candidates can be decoded. Moreover, if $m > b$, Byzantine storage-nodes cannot collude to decode data-fragments. In particular, if secret sharing [47] or short secret sharing [24] is employed, data-fragment confidentiality can be ensured.

4.2 Asynchronous constraints

In an asynchronous environment, a read operation can only wait for $N - t$ responses from storage-nodes, since slow communication is indistinguishable from crashed storage-nodes. Moreover, of the responses, up to b can be arbitrary.

COMPLETE WRITE DEFINITION: In an asynchronous system, a write is complete once a total of Q_C benign storage-nodes have executed the write.

WRITE TERMINATION: There must be sufficient benign storage-nodes in the system for a write operation to complete. Since only $N - t$ responses can be awaited,

$$\begin{aligned} Q_C + b &\le N - t, \\ Q_C &\le N - t - b. \end{aligned} \tag{1}$$

READ CLASSIFICATION: To classify a candidate as complete, a candidate set of at least Q_C benign storage-nodes must be observed. In the worst case, at most b members of the candidate set may be Byzantine, thus,

$$|CandidateSet| - b \ge Q_C \Rightarrow \text{complete}. \tag{2}$$

A candidate set can only be classified incomplete if a client knows that a complete write does not exist in the system (i.e., fewer than Q_C benign storage-nodes host the write). For this to be the case, the client must have queried all possible storage-nodes ($N - t$), and must assume that nodes not queried host the candidate in consideration. So,

$$|CandidateSet| + t < Q_C \Rightarrow \text{incomplete}. \tag{3}$$

REAL UNCLASSIFIABLE/REPAIRABLE CANDIDATES: To ensure that Byzantine storage-nodes cannot fabricate an unclassifiable/repairable candidate, a candidate set of size b must be classifiable as incomplete. Substituting b into (3),

$$b + t < Q_{\mathrm{C}}. \tag{4}$$

4.2.1 Asynchronous repairable

DECODABLE REPAIRABLE CANDIDATES: Any repairable candidate must be decodable. The lower bound on candidate sets that are repairable follows from (3) (since the upper bound on classifying a candidate as incomplete coincides with the lower bound on repairable):

$$1 \leq m \leq Q_{\mathrm{C}} - t. \tag{5}$$

CONSTRAINT SUMMARY:

$$\begin{aligned}
|CandidateSet| \geq Q_{\mathrm{C}} + b &\Rightarrow \text{complete};\\
|CandidateSet| < Q_{\mathrm{C}} - t &\Rightarrow \text{incomplete};\\
t + b + 1 \leq Q_{\mathrm{C}} &\leq N - t - b;\\
2t + 2b + 1 &\leq N;\\
1 \leq m &\leq Q_{\mathrm{C}} - t.
\end{aligned}$$

4.2.2 Asynchronous non-repair

Repairable candidates in repairable members are unclassifiable candidates in non-repair members (i.e., read operations can abort). An additional constraint is introduced to ensure that Byzantine storage-nodes cannot force isolated reads subsequent to write operations by correct clients to abort.

CLASSIFIABLE COMPLETE CANDIDATES: For this property to hold, a read operation must observe at least $Q_{\mathrm{C}} + b$ responses from benign storage-nodes—sufficient responses to classify the candidate as complete (2). A write operation by a correct client may only complete at $N - t$ storage-nodes, and a subsequent read operation may not observe responses from t benign storage-nodes. Further, b observed responses may be Byzantine. So,

$$\begin{aligned}
Q_{\mathrm{C}} + b &\leq (N - t) - t - b,\\
Q_{\mathrm{C}} &\leq N - 2t - 2b.
\end{aligned} \tag{6}$$

DECODABLE COMPLETE CANDIDATES: A candidate classified as complete, (2), must be decodable:

$$1 \leq m \leq Q_{\mathrm{C}} + b. \tag{7}$$

CONSTRAINT SUMMARY:

$$\begin{aligned}
|CandidateSet| \geq Q_{\mathrm{C}} + b &\Rightarrow \text{complete};\\
|CandidateSet| < Q_{\mathrm{C}} - t &\Rightarrow \text{incomplete};\\
t + b + 1 \leq Q_{\mathrm{C}} &\leq N - 2t - 2b;\\
3t + 3b + 1 &\leq N;\\
1 \leq m &\leq Q_{\mathrm{C}} + b.
\end{aligned}$$

4.3 Synchronous constraints

In a synchronous environment, crashed storage-nodes are detectable. Let f be the number of observed timeouts by a particular read operation. Clearly, $0 \leq f \leq t$; as well, f is at least as great as the number of crashed storage-nodes in the system when the read operation began—Byzantine storage-nodes can "return" timeouts and storage-nodes may fail at any time during the read operation.

COMPLETE WRITE DEFINITION: In a synchronous system, a write is complete once a total of Q_C benign storage-nodes have executed the write or have crashed.

WRITE TERMINATION: Sufficient benign storage-nodes must exist for a write operation to complete:

$$Q_\text{C} + b \leq N. \tag{8}$$

READ CLASSIFICATION: In the worst case, the candidate set plus the set of timeouts can contain b Byzantine storage-nodes, with the remainder being benign. Thus,

$$(|CandidateSet| + f) - b \geq Q_\text{C} \Rightarrow \text{complete}. \tag{9}$$

To classify a candidate as incomplete, a client must be sure that the candidate is not part of a complete write. For synchronous protocol members, the classification rule assumes that responses (up to t of which may be timeouts) from all N storage-nodes have been received. The client must assume that the candidate set and the set of timeouts are from benign storage-nodes, thus:

$$|CandidateSet| + f < Q_\text{C} \Rightarrow \text{incomplete}; \tag{10}$$

REAL UNCLASSIFIABLE/REPAIRABLE CANDIDATES: To prevent Byzantine storage-nodes from fabricating a write operation, a candidate set of size b must be classifiable as incomplete. If b responses within the candidate set come from Byzantine storage-nodes, then $f \leq t - b$. So, (10) becomes,

$$\begin{aligned} b + (t - b) &< Q_\text{C}, \\ t &< Q_\text{C}. \end{aligned} \tag{11}$$

4.3.1 Synchronous repairable

DECODABLE REPAIRABLE CANDIDATES: The lower bound on candidate sets that are repairable follows from (10). Since in the worst case, $f = t$,

$$1 \leq m \leq Q_\text{C} - t. \tag{12}$$

CONSTRAINT SUMMARY:

$$\begin{aligned} |CandidateSet| &\geq Q_\text{C} - f + b \Rightarrow \text{complete}; \\ |CandidateSet| &< Q_\text{C} - f \Rightarrow \text{incomplete}; \\ t + 1 &\leq Q_\text{C} \leq N - b; \\ Q_\text{C} + b &\leq N; \\ 1 &\leq m \leq Q_\text{C} - t. \end{aligned}$$

4.3.2 Synchronous non-repair

CLASSIFIABLE COMPLETE CANDIDATES: To ensure that a write from a correct client can be classified as complete, a read operation must observe at least $Q_C + b$ responses from benign storage-nodes. Since up to b storage-nodes may be Byzantine,

$$\begin{aligned} Q_C + b &\leq N - b, \\ Q_C &\leq N - 2b. \end{aligned} \tag{13}$$

DECODABLE COMPLETE CANDIDATES: A candidate classified as complete must be decodable. Thus, the upper bound on m follows from the minimum value of (9):

$$1 \leq m \leq Q_C + b - t. \tag{14}$$

CONSTRAINT SUMMARY:

$$\begin{aligned} |CandidateSet| &\geq Q_C - f + b \Rightarrow \text{complete}; \\ |CandidateSet| &< Q_C - f \Rightarrow \text{incomplete}; \\ t + 1 \leq Q_C &\leq N - 2b; \\ t + 2b + 1 &\leq N; \\ 1 \leq m &\leq Q_C + b - t. \end{aligned}$$

5 Design and implementation

This section describes the design and implementation of our protocol family. It presents the mechanisms employed to encode and decode data, to protect data integrity, and to authenticate requests. It then details the protocol family in pseudo-code form and provides implementation details about our prototype block-based storage system.

5.1 Mechanisms

This subsection describes mechanisms employed for encoding data, preventing Byzantine clients and storage-nodes from violating consistency, and authenticating client and storage-node requests. We assume that storage-nodes and clients are computationally bounded such that cryptographic primitives can be effective.

Erasure codes. We consider only threshold erasure codes in which any m of the N encoded data-fragments can decode the data-item; moreover, every m data-fragments can be used to deterministically generate the other $N - m$ data-fragments.

In our implementation, if $m = 1$, then replication is employed. Otherwise, our base erasure code implementation stripes the data-item across the first m data-fragments; each *stripe-fragment* is $\frac{1}{m}$ the length of the original data-item. Thus, concatenation of the first m data-fragments produce the original data-item. These stripe-fragments are used to generate the *code-fragments* via polynomial interpolation within a Galois Field. Our implementation of polynomial interpolation was originally based on [10] (which conceptually follows [42]). We modified the source to use stripe-fragments and added an implementation of Galois Fields of size 2^8 that use a lookup table for multiplication.

Beyond our base erasure code implementation, we implemented secret sharing [47] and short secret sharing [24]. Our implementation of short secret sharing closely follows [24], using AES for the cipher. Such erasure codes can also provide a degree of confidentiality with regard to storage-nodes.

Data-fragment integrity. Byzantine storage-nodes can corrupt their data-fragments. As such, it must be possible to detect and mask up to b storage-node integrity faults. Cross checksums [13] are used to detect corrupt data-fragments: a cryptographic hash of each data-fragment is computed, and the set of N hashes are concatenated to form the *cross checksum* of the data-item. The cross checksum is stored with each data-fragment, enabling corrupted data-fragments to be detected.

Our implementation uses a publicly available implementation of MD5 [1] for all hashes.

Write operation integrity. Mechanisms are required to prevent Byzantine clients from performing a write operation that lacks integrity. If a Byzantine client generates random data-fragments (rather than erasure coding a data-item correctly), then each of the $\binom{N}{m}$ subsets of data-fragments could "recover" a distinct data-item. This attack is similar to *poisonous writes* for replication, as described by Martin et al. [32]. To protect against such Byzantine client actions, read operations must only return values that are written correctly (i.e., that are *single-valued*). To achieve this, the cross checksum mechanism is extended in three ways:

VALIDATING TIMESTAMPS: To ensure that only a single set of data-fragments can be written at any logical time, the hash of the cross checksum is placed in the low order bits of the logical timestamp.

STORAGE-NODE VERIFICATION: On a write, each storage-node verifies its data-fragment against the corresponding hash in the cross checksum. The storage-node also verifies that the cross checksum matches the low-order bits of the validating timestamp. A correct storage-node only executes write requests for which both checks pass. Thus, a Byzantine client cannot make a correct storage-node appear Byzantine—only Byzantine storage-nodes can return unverifiable responses.

VALIDATED CROSS CHECKSUMS: Combined, storage-node verification and validating timestamps ensure that the data-fragments being considered by any read operation were not fabricated by Byzantine storage-nodes. To ensure that the client that performed the write operation acted correctly, the cross checksum must be validated by the reader. For the reader to validate the cross checksum, all N data-fragments are required. Given any m data-fragments, the reader can generate the full set of N data-fragments a correct client should have written. The reader can then compute the "correct" cross checksum from the generated data-fragments. If the generated cross checksum does not match the validated cross checksum, then a Byzantine client performed the write operation. Only a single-valued write operation can generate a cross checksum that can be validated.

Authentication. RPC requests and responses are authenticated using HMACs (i.e., clients and storage-nodes have pair-wise shared secrets). Thus, the channels between clients and storage-nodes are authenticated. We assume some infrastructure is in place to distribute shared secrets—our implementation supports an existing Kerberos [48] infrastructure.

5.2 Storage-node design

Storage-nodes expose the same interface, regardless of the protocol member being employed—write and read requests for all protocol members are serviced identically. Storage-nodes offer interfaces to write a data-fragment at a specific logical time; to query the greatest logical time of a data-fragment; to read the data-fragment version with the greatest logical time; and to read the data-fragment with the greatest logical time before some logical time. As mentioned above, storage-nodes validate write requests before executing them (to protect against Byzantine clients). Pseudo-code for request validation is presented as part of the client read operation (in Section 5.3.2).

Storage-node implementation. Implicitly, each write request creates a new version of the data-fragment (indexed by its logical timestamp) at the storage-node. In addition to data, each write request contains a data-item cross checksum and a *linkage record* [3]. The linkage record consists of descriptions of the encoding scheme, family member, and addresses of the N storage-nodes for a specific data-item; it is fixed upon data-item creation.

The storage-node implementation uses a log-structured organization [44] to reduce the cost of data versioning. Like previous researchers [41, 49], our experiences indicate that retaining every version and performing local garbage collection come with minimal performance cost (a few percent) and that it is feasible to retain complete version histories for several days.

By default, a read request returns the most current data-fragment version, ordered by logical timestamp. Reads may also request the data-fragment corresponding to a specific logical timestamp, or just part of the version history (sequence of logical timestamps) corresponding with associated with a data-fragment.

Garbage collection. Pruning old versions, or garbage collection, is necessary to prevent capacity exhaustion of the storage-nodes. A storage-node in isolation cannot determine which local data-fragment versions are safe to garbage-collect, because write completeness is a property of a set of storage-nodes. A data-fragment version can be garbage-collected only if there exists a later complete write for the corresponding data-item. Storage-nodes can classify writes by executing the read protocol in the same manner as a client. However, no data need be returned for protocol members that do not tolerate Byzantine clients (since the cross checksum need not be validated). Linkage records provide sufficient information for the storage-nodes to know which other nodes host relevant data-fragments.

Garbage collection is implemented in the current prototype and it requires no additional RPCs. It currently must be initiated externally (e.g., by a CRON job). Research into policy issues, such as the appropriate frequency and order of garbage collection, is warranted.

5.3 Client design

Clients do most of the work, including the execution of the consistency protocol and the encoding and decoding of data-items. The client module provides a block-level interface to higher-level software. This subsection describes client read and write operations through the use of pseudo-code and the next subsection provides additional details about the actual implementation.

The read and write client pseudo-code relies on some new terms. The symbol, LT, denotes logical time and, $LT_{\text{candidate}}$, denotes the logical time of the candidate. The set, $\{D_1, \ldots, D_N\}$, denotes the N data-fragments; likewise, $\{S_1, \ldots, S_N\}$ denotes the N storage-nodes. The operator '$|$' denotes concatenation.

The symbol, $Wait_{\text{MAX}}$, defines the maximum number of responses for which write and read operations can wait for before having to terminate. In an asynchronous system, storage-nodes that crash cannot be distinguished from slow storage-nodes, so a client cannot wait for the last t responses. On the other hand, in a synchronous system, unresponsive storage-nodes are detectable through timeout "responses". Therefore, in an asynchronous system $Wait_{\text{MAX}} = N - t$, and in a synchronous system $Wait_{\text{MAX}} = N$.

5.3.1 Write operation

The write operation pseudo-code is shown in Figure 2. The `WRITE` operation consists of determining the greatest logical timestamp, constructing write requests, and issuing the requests to the storage-nodes.

First, a timestamp greater than, or equal to, that of the latest complete write is determined by the `READ_TIMESTAMP` function on line 2 of `WRITE`. In a synchronous system, the client's local clock is read (line 2 of `READ_TIMESTAMP`). In an asynchronous system, `GET_TIME` requests are issued to the storage-nodes. Responses are collected, and the highest timestamp is identified, incremented, and returned.

Next, the `ENCODE` function, on line 3, encodes the data-item into N data-fragments. Hashes of the data-fragments are used to construct a cross checksum (line 4). The function `MAKE_TIMESTAMP`, called on line 5, generates a logical timestamp for the write operation by combining the hash of the cross checksum and the time determined by `READ_TIMESTAMP`.

Finally, write requests are issued to all storage-nodes. Each storage-node is sent a specific data-fragment, the logical timestamp, and the cross checksum. A storage-node validates the cross checksum

```
WRITE(Data) :
 1: /* Encode data, construct timestamp, and write data-fragments */
 2: Time := READ_TIMESTAMP()
 3: {D_1, ..., D_N} := ENCODE(Data)
 4: CC := MAKE_CROSS_CHECKSUM({D_1, ..., D_N})
 5: LT := MAKE_TIMESTAMP(Time, CC)
 6: DO_WRITE({D_1, ..., D_N}, LT, CC)

READ_TIMESTAMP() :
 1: if (Synchronous = TRUE) then
 2:     Time := READ_LOCAL_CLOCK()
 3: else
 4:     for all S_i ∈ {S_1, ..., S_N} do
 5:         SEND(S_i, GET_TIME)
 6:     end for
 7:     ResponseSet := ∅
 8:     repeat
 9:         ResponseSet := ResponseSet ∪ RECEIVE(S, GET_TIME)
10:     until (|ResponseSet| = Wait_MAX)
11:     Time := MAX[ResponseSet.LT.Time] + 1
12: end if
13: RETURN(Time)

MAKE_CROSS_CHECKSUM({D_1, ..., D_N}) :
 1: for all D_i ∈ {D_1, ..., D_N} do
 2:     H_i := HASH(D_i)
 3: end for
 4: CC := H_1 | ... | H_N
 5: RETURN(CC)

MAKE_TIMESTAMP(Time, CC) :
 1: LT.Time := Time
 2: LT.Verifier := HASH(CC)
 3: RETURN(LT)

DO_WRITE({D_1, ..., D_N}, LT, CC) :
 1: for all S_i ∈ {S_1, ..., S_N} do
 2:     SEND(S_i, WRITE_REQUEST, LT, D_i, CC)
 3: end for
 4: ResponseSet := ∅
 5: repeat
 6:     ResponseSet := ResponseSet ∪ RECEIVE(S, WRITE_RESPONSE)
 7: until (|ResponseSet| = Wait_MAX)
```

Figure 2: **Client-side write operation pseudo-code.**

with the corresponding portion of the timestamp and validates the data-fragment with the cross checksum before executing a write request (i.e., storage-nodes call VALIDATE listed in the read operation pseudo-code). The write operation returns to the issuing client once enough WRITE_RESPONSE replies are received (line 7 of DO_WRITE).

5.3.2 Read operation

The pseudo-code for the read operation is shown in Figure 3. The read operation iteratively identifies and classifies candidates until either a complete or repairable candidate is found or the operation aborts due to insufficient information (only non-repair members can abort). Before a repairable or complete candidate is returned, the read operation validates its correctness.

The read operation begins by issuing READ_LATEST requests to all storage-nodes (via the DO_READ

```
READ($Repair$) :
ResponseSet := DO_READ(READ_LATEST, $\perp$)
loop
    $\langle CandidateSet, LT_{\text{candidate}} \rangle$ := CHOOSE_CANDIDATE($ResponseSet$)
    if ($|CandidateSet| \geq$ COMPLETE) then
        /* Complete candidate: return value */
        if (VALIDATE_CANDIDATE_SET($CandidateSet$)) then
            $Data$ := DECODE($CandidateSet$)
            RETURN($Data$)
        end if
    else if ($|CandidateSet| \geq$ INCOMPLETE) then
        /* Unclassifiable candidate found: repair or abort */
        if ($Repair$ = TRUE) then
            if (VALIDATE_CANDIDATE_SET($CandidateSet$) then
                $\{D_1, \ldots, D_N\}$ := GENERATE_FRAGMENTS($CandidateSet$)
                DO_WRITE($\{D_1, \ldots, D_N\}$, $LT_{\text{candidate}}$, $CC_{\text{valid}}$)
                $Data$ := DECODE($\{D_1, \ldots, D_N\}$)
                RETURN($Data$)
            end if
        else
            RETURN(ABORT)
        end if
    end if
    /* Incomplete candidate or validation failed: loop again */
    $ResponseSet$ := DO_READ(READ_PREVIOUS, $LT_{\text{candidate}}$)
end loop

DO_READ(READ_COMMAND, $LT$) :
for all $S_i \in \{S_1, \ldots, S_N\}$ do
    SEND($S_i$, READ_COMMAND, $LT$)
end for
$ResponseSet := \emptyset$
repeat
    $Resp$ := RECEIVE($S$, READ_RESPONSE)
    if (VALIDATE($Resp.D$, $Resp.CC$, $Resp.LT$, $S$) = TRUE) then
        $ResponseSet := ResponseSet \cup Resp$
    end if
until ($|ResponseSet| = Wait_{\text{MAX}}$)
RETURN($ResponseSet$)

VALIDATE($D$, $CC$, $LT$, $S$) :
if ((HASH($CC$) $\neq LT.Verifier$) OR (HASH($D$) $\neq CC[S]$)) then
    RETURN(FALSE)
end if
RETURN(TRUE)

VALIDATE_CANDIDATE_SET($CandidateSet$)
if ($ByzantineClients$ = TRUE) then
    /* Byzantine clients: regenerate fragments for validation */
    $\{D_1, \ldots, D_N\}$ := GENERATE_FRAGMENTS($CandidateSet$)
    $CC_{\text{valid}}$ := MAKE_CROSS_CHECKSUM($\{D_1, \ldots, D_N\}$)
    if ($CC_{\text{valid}} = CandidateSet.CC$) then
        RETURN(TRUE)
    else
        RETURN(FALSE)
    end if
end if
/* Crash clients: return TRUE */
RETURN(TRUE)
```

Figure 3: **Client-side read operation pseudo-code.**

function). Each storage-node responds with the data-fragment, logical timestamp, and cross checksum corresponding to the highest timestamp it has executed.

The integrity of each response is individually validated by the `VALIDATE` function, line 7 of `DO_READ`. This function checks the cross checksum against the validating timestamp and the data-fragment against the appropriate hash in the cross checksum. Since correct storage-nodes perform the same validation before executing write requests, only responses from Byzantine storage-nodes can fail the reader's validation.

After sufficient responses have been received, a candidate for classification is chosen. The function `CHOOSE_CANDIDATE`, on line 3 of `READ`, determines the candidate timestamp, denoted $LT_{\text{candidate}}$, which is the greatest timestamp in the response set. All data-fragments that share $LT_{\text{candidate}}$ are identified and returned as the candidate set. At this point, the candidate set contains a set of data-fragments that share a common cross checksum and logical timestamp.

Once a candidate has been chosen, it is classified as either complete, unclassifiable (repairable), or incomplete. If the candidate is classified as incomplete, a `READ_PREVIOUS` message is sent to each storage-node with the candidate timestamp. Candidate classification begins again with the new response set.

If the candidate is classified as complete or repairable, the candidate set is constrained to contain sufficient data-fragments (see Section 4) to decode the original data-item. At this point the candidate is validated. This is done through the `VALIDATE_CANDIDATE_SET` call on line 6 (for complete candidates) or line 13 (for repairable candidates) of `READ`.

For family members that do not tolerate Byzantine clients, this call is a no-op returning `TRUE`. Otherwise, the candidate set is used to generate the full set of data-fragments, as shown in line 3 of `VALIDATE_CANDIDATE_SET`. A validated cross checksum, CC_{valid}, is then computed from the newly generated data-fragments. The validated cross checksum is compared to the cross checksum of the candidate set (line 5 of `VALIDATE_CANDIDATE_SET`). If the check fails, the candidate was written by a Byzantine client; the candidate is reclassified as incomplete, and the read operation continues. If the check succeeds, the candidate was written correctly and the read enters its final phase. Note that this check either succeeds or fails for all correct clients, regardless of which storage-nodes are represented within the candidate set.

If necessary and allowed, repair is performed: write requests are issued with the generated data-fragments, the validated cross checksum, and the logical timestamp (line 15 of `READ`). Storage-nodes not currently hosting the write execute the write at the given logical time; those already hosting the write are safe to ignore it.

Finally, the function `DECODE` recovers the data-item from any m data-fragments.

5.4 Client implementation

Our client implementation follows the pseudo-code described above. The client module is accessed through a set of library interface calls. These calls allow an application to control the encoding scheme, the threshold values, and the failure and timing models. The client protocol routines are implemented such that different protocol family members and thresholds may be specified for different data-items. Likewise, the storage-nodes for any given data-item are also specified via these interfaces, thus externalizing control (and responsibility) for such bootstrapping information; for our experiments we use a static set of N storage-nodes. Clients communicate with storage-nodes through a TCP-based RPC interface.

In an asynchronous environment, the client implementation issues `GET_TIME` requests to only $N+b-Q_{\text{C}}$ storage-nodes, since this ensures overlap with the latest complete write. In a synchronous environment, client clocks are synchronized using NTP [33]. To improve the responsiveness of write operations, clients return after the first $Q_{\text{C}}+b$ storage-nodes respond; the remainder of the requests complete in the background.

To improve the read operation's performance, only m read requests fetch the latest data pertaining to the data-fragment, while all receive version histories; this makes the read operation more network-efficient. Read requests also return a limited version history of the data-fragment requested (with no corresponding data). This version history allows clients to classify earlier writes without issuing additional storage-node requests. If necessary, after classification, extra data-fragments are fetched according to the candidate's timestamp.

6 Performance evaluation

This section evaluates protocol family performance in the context of the prototype block storage system.

6.1 Experimental setup

We use a cluster of 20 machines to perform experiments. Each storage-node is a dual 1GHz Pentium III machine with 384 MB of memory and a 9GB Quantum Atlas 10K disk. Each client is a single processor 2GHz Pentium IV machine. The machines are connected through a 100Mb switch. All machines run the Linux 2.4.20 SMP kernel.

In all experiments, clients keep a fixed number of read and write operations outstanding—when an operation completes, a new operation is issued immediately. Unless otherwise specified, requests are for random 16 KB blocks. Unless otherwise specified, Q_C and N are the minimum allowable values for the protocol member, as given in Table 2, and m is the maximum allowable value. Authentication costs (i.e., HMAC computations) are included in all experiments.

No caching is done on the clients. Storage-nodes use write-back caching, mimicking availability of 16 MB of non-volatile RAM. All experiments focus on the protocol costs: the working sets fit into memory and all caches are warmed up beforehand. Results from such experiments highlight the overheads introduced by the protocol and not those introduced by the disk system. It is, however, a full system implementation: each storage-node is backed by a real persistent data store, and compulsory cache flushes are serviced by the disk system.

6.2 Space-efficiency of protocol members

All protocol members can employ m-of-n erasure codes. Increasing m improves space-efficiency, since each data-fragment is $\frac{1}{m}$ the size of the data-item. Space-efficiency reduces the network bandwidth needed, which reduces the response time of operations.

To perform a write operation, N data-fragments are sent over the network. With each data-fragment, a cross checksum and linkage record are sent. Respectively, these are N times the size of a MD5 digest (16 bytes) and N times the size of a storage-node ID (4 bytes). RPC headers and arguments consume negligible bandwidth. Thus, the total amount of data sent over the network by a write operation is: $16\text{ KB} \times \frac{N}{m} + 20\text{ B} \times N^2$.

6.3 Computation costs

Computation costs are incurred to erasure code data. Additional computation costs are incurred to authenticate messages and protect against non-crash failures.

Erasure coding costs. Figure 4 shows the trends in the cost of encoding data with our erasure code implementation. For comparison, the performance of N-fold replication (i.e., N `memcpys`) is shown. Lines are shown for fixed m values of two and three. These lines illustrate that, as expected, the cost of an erasure code for a given m grows linearly with N, since the number of code-fragments grows with N.

Two other lines are shown in Figure 4 to illustrate the interesting impact of m on performance: the space-efficiency of an erasure code is inversely proportional to m whereas the cost of generating some aggregate amount of code-fragment is proportional to m. Consider the $m = \frac{N}{2}$ line. For each point on

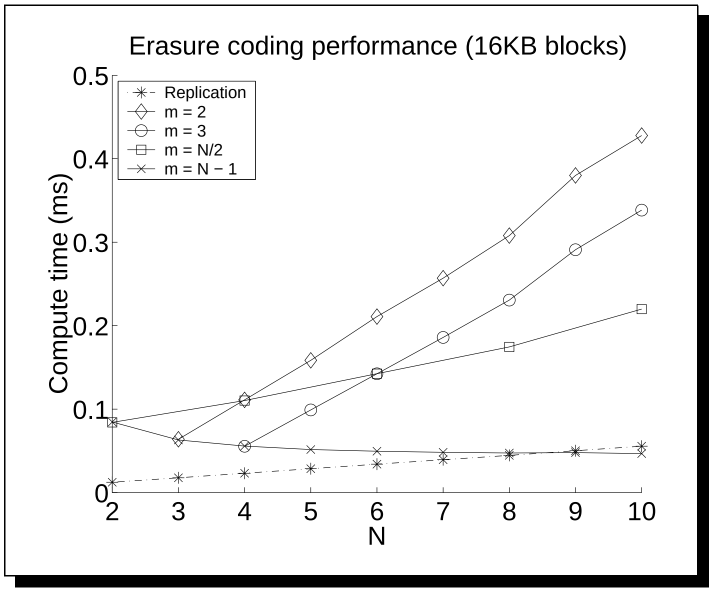

Figure 4: **Computational cost of erasure coding.** Block size, N, and m dictate the computational cost of erasure coding.

the line, erasure coding generates, in total, 16 KB of code-fragments, although the number and individual sizes of the code-fragments differ. When generating some aggregate amount of code-fragments, the cost of erasure coding grows linearly with m. For $m = N-1$, a single code-fragment is needed for each write; as expected, the cost of generating one fragment decreases with N, since the size of the fragment also decreases (to $\frac{1}{N-1}$).

Computation cost breakdown. Table 2 enumerates the client and storage-node computation costs for asynchronous repairable members tolerating one and four Byzantine storage-node faults.

CLIENTS: All protocol family members place the majority of the computational work on clients in the system. Erasure-coding is done by the client and requires nothing of the storage-node. The difference in computation costs for the two instances of the protocol member listed is due to their respective values of N and m. The cost of erasure coding with regard to N and m is dicussed above. The cost of generating cross checkums grows with $\frac{N}{m}$.

Read operations in protocol members with only crash clients are computationally less demanding than write operations. A read operation requires fewer hashes of data-fragments and generation of fewer code-fragments. In the best case, the m stripe-fragments can be concatenated and no code-fragments need be generated. In protocol members that tolerate Byzantine clients, read operations performs almost the same computation as write operations to validate the cross checksum (i.e., $N-m$ code-fragments are generated and N data-fragments hashes are taken).

Short secret sharing can be used in place of our default erasure code. Doing so adds $\approx$550 μs to the base erasure code costs for encrypting the data-item under the AES cipher and less than 20 μs for generating and secret sharing the encryption key (this cost depends on m and N). Both write and read operations incur these costs.

STORAGE-NODES: For each write request, a storage-node must verify both the timestamp and the data-fragment. Validating the data-fragment is roughly $\frac{1}{N}$ the work the client does in creating the cross checksum. A hash of the cross checksum is taken to verify the hash within the timestamp. Read requests require no significant computation by the storage-node (for the protocol).

AUTHENTICATION: Clients and storage-nodes must authenticate each RPC request and response. Authen-

	$b = t = 1$	$b = t = 4$
Storage-node: write operation costs		
Verify timestamp	1.56 μs	3.78 μs
Verify data-fragment	72.2	29.4
Client: write operation costs		
Encode: generate $N - m$ code-fragments	163	546
Generate one code-fragment	*54.2*	*45.5*
Generate cross checksum: hash N data-fragments	359	512
Hash one data-fragment	*71.2*	*30.1*
Generate validating timestamp	1.60	3.72
Client: read operation costs		
Verify data-fragments: hash m data-fragments	143	150
Best case decode: `memcpy` m stripe-fragments	6.84	7.86
Worst case decode: generate m code-fragments	108	228
Validate cross checksum (to tolerate Byz. clients)	522	1060

Table 2: **Client and storage-node computation costs.** Costs are broken down for the asynchronous repairable protocol member with Byzantine storage-nodes for: $b = t = 1$ and $b = t = 4$ ($N = 5,\ m = 2$ and $N = 17,\ m = 5$, respectively).

tication is performed over the RPC header and some RPC arguments. Cross checksums and data-fragments are not directly included in the authentication; however, the validating timestamp is included, and it indirectly authenticates the remainder. In all cases, authentication of an RPC message requires less than 2.5 μs.

6.4 Scalability

Figure 5 shows mean response times as a function of the number of tolerated storage-node failures. Read and write response times are shown for each of the four $\langle$asynchronous/synchronous, repairable/non-repair$\rangle$ protocol members that tolerate Byzantine storage-nodes and crash clients. t is constrained by our experimental setup; recall, we have 20 machines in our testbed.

All of the response time lines are straight, indicating that each protocol member scales linearly with t (faults tolerated). The flatness of most of the response time lines indicates that network bandwidth consumed and computation required does not grow substantially with t. As well, even though N grows with t, storage-node communication can usually be overlapped.

The read response time lines are so flat because of the space-efficiency of read operations. Read operations only fetch full data-fragments from m storage-nodes. Thus, read operations by all protocol members require only a total of $\approx$16 KB of data to be returned over the network. The slight slope is due to the increases in N and Q_C, which increases communication costs. The only difference in response time between protocol members with crash clients (shown in the plot) and members with Byzantine clients (not shown) is in computation costs incurred by read operations.

The differences in write response time among protocol members is directly attributable to their differences in space-efficiency. Consider the synchronous repairable protocol member, which, in this experiment,

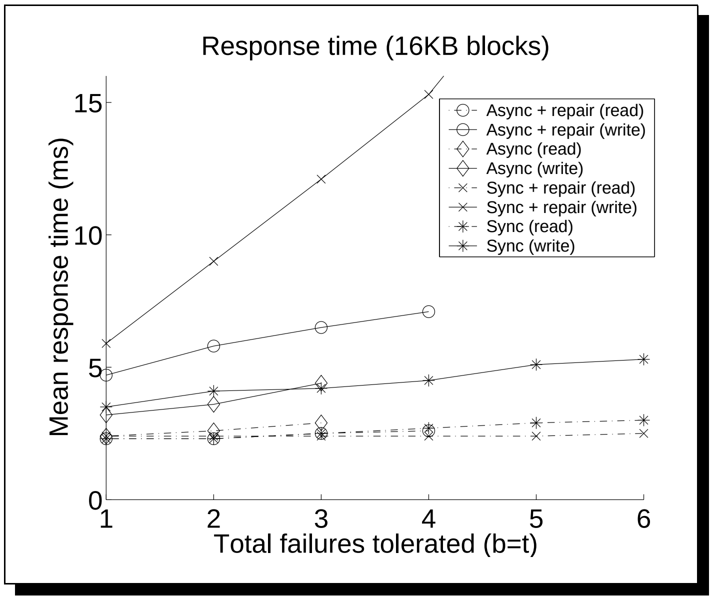

Figure 5: **Response time of protocol family members.** Mean response times of read and write operations of random 16 KB blocks are shown for the ⟨asynchronous/synchronous, repair/non-repair⟩ protocol family members with crash clients and Byzantine storage-nodes.

uses $m = 1$ (replication). Since $N = t + b + 1$ and $b = t$, then $N = 2t + 1$; at $t = 4$, 9-fold replication is employed, which consumes 146 KB of network bandwidth for each write, taking at least 11.9 ms on a 100 Mbps network link.

Increasing m improves the space-efficiency of protocol members. To increase m, one must also increase Q_C and N (constraints on m, Q_C and N are listed in Table 1). Figure 6 shows the effect of increasing m (along with Q_C and N) on mean write response time for the synchronous repairable protocol member. The increase from $m = 1$ to $m = 2$ yields a dramatic improvement: space-efficiency is doubled, network utilization is effectively halved. Additional increases in m further improve space-efficiency, although diminishing returns are encountered.

Throughput experiments were also performed. Due to space limitations, graphs of throughput measurements are omitted. The throughput scales until either the client or storage-node network links are saturated. The point of saturation depends on the space-efficiency of the protocol member. With our experimental setup, the clients and storage-nodes are never compute bound—with a faster network, of course, this may change.

6.5 WAN

To emulate WAN-like latencies, we used the NIST Net network emulator [38]. NIST Net is implemented as a Linux kernel module that allows for the emulation of various network conditions. We used NIST Net to set a 12.5 ms delay on each client/storage-node link, thus producing a perceived 25 ms round-trip delay. Figure 7 shows the results for both the synchronous and asynchronous non-repair protocol members with Byzantine storage-nodes and crash clients. In addition to the fixed 25 ms delay experiments, experiments with a 25 ms round-trip delay and 15 ms standard deviation were performed (the *stdv* lines).

First, consider the experiments with a fixed delay. Synchronous reads, synchronous writes, and asynchronous reads have approximately the same response time, as expected, given the single round-trip involved. The response time for asynchronous writes is twice the round-trip time; an extra round-trip is

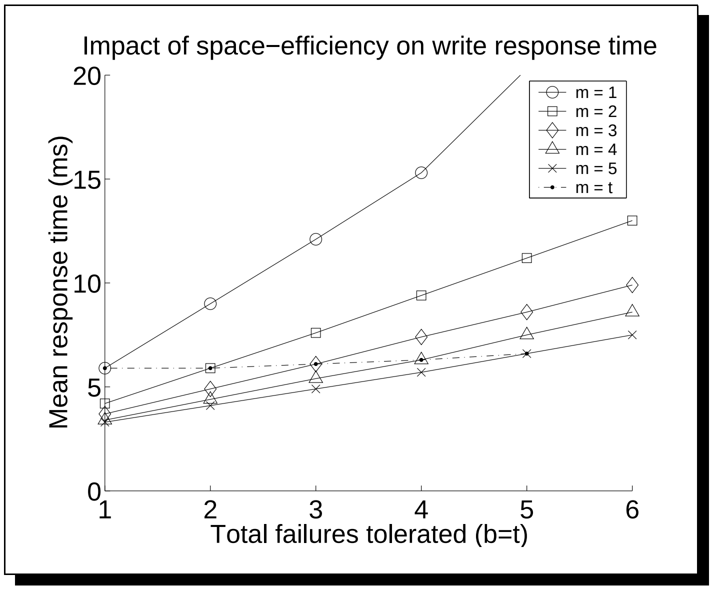

Figure 6: **Effect of *m* on write response time.** Increasing *m* (i.e., increasing space-efficiency) reduces the mean response time of write operations for the synchronous repairable protocol member with Byzantine storage-nodes and crash clients. Here, $Q_C = t + m$ and $N = 2t + m$.

required to obtain the logical timestamp.

Now, consider the impact of variance in round trip time. Synchronous write operations have lower response times than synchronous read operations. This is due to the prototype's implementation. Write operations return once $Q_C + b$ write requests have been acknowledged by storage-nodes, although N requests are issued. Thus, the write operation returns once the fastest acknowledgements have been received. On the other hand, read operations initially issue the minimum number of requests needed to classify a candidate as complete. As such, clients must wait for all read requests to complete. Asynchronous protocol members also benefit from the early-completion of writes, however, due to the additional round-trip to obtain the logical timestamp, writes do not complete faster than reads. Read response time could be reduced similarly by "over-requesting" [52, 53] (issuing additional requests) and terminating once a sufficient number have returned. There is a tradeoff between the responsiveness gained by issuing additional requests and the throughput lost by performing unnecessary work.

6.6 Concurrency

Read-write concurrency can lead to client read operations observing repairable writes or aborting. To explore the effect of concurrency on the system, we measure multi-client throughput when accessing overlapping block sets. The experiment consists of four clients, each with four operations outstanding. Each client accesses a range of eight data blocks, some overlapping with other clients and some not, and no outstanding requests from the same client going to the same block.

At the highest concurrency level—all eight blocks in contention by all clients—we observed neither significant drops in throughput nor significant increases in mean response time. For example, the asynchronous repairable protocol member classified the initial candidate as complete 88.8% of the time, and found repair was necessary only 3.3% of the time. Since repair occurs rarely, the effect on average response time and throughput is minimal.

Read aborts can occur when reads are concurrent to in-progress writes. By retrying read operations, almost all such "false" aborts can be eliminated.

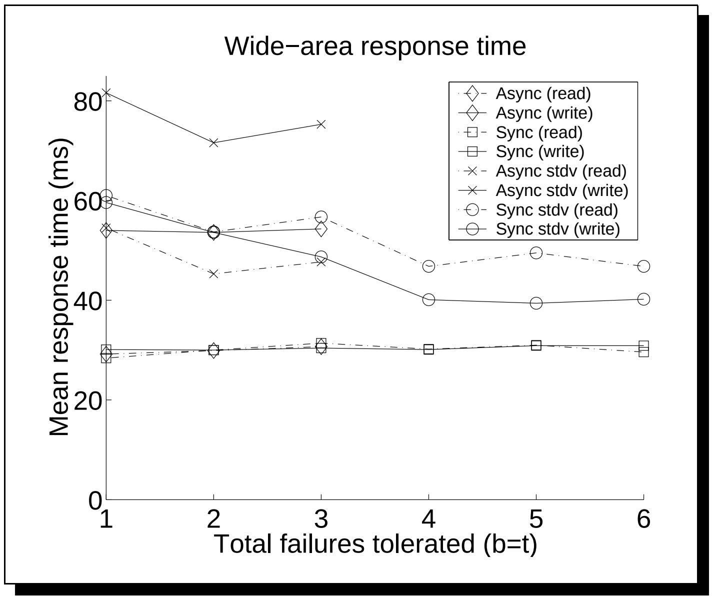

Figure 7: **Response time with emulated network delays.** The mean response times for the asynchronous & synchronous non-repair protocol members with crash clients and Byzantine storage-nodes are plotted. The first four lines show response times for a fixed round trip time of 25 ms. The *stdv* lines are for a round trip time of 25 ms with a standard deviation of 15 ms.

6.7 Other results

Storage-node failures. For clients of both synchronous and asynchronous protocol members, storage-node failures have minimal impact on performance.

Client crash failures. Client crash failures appear as partially written data. Subsequent reads may observe these writes as incomplete or unclassifiable. If they are unclassifiable, the read must either abort or attempt repair. Repair adds much of the cost of performing a write, though, the round-trip to obtain a logical timestamp in an asynchronous system is not needed.

Garbage collection. We assume a large window of storage version capacity, so garbage collection usually occurs during idle periods. But, even when it competes with real requests, garbage collection is inexpensive. Garbage collection requests are just batched read requests, except that no data need be returned for members that do not tolerate Byzantine clients. When Byzantine clients are tolerated, garbage collection must validate the cross checksum, which does require data-fragments.

7 Conclusion

A protocol family shifts fault model decisions from system implementation time to data creation time. Our family of access protocols exploits versioning and erasure coding to efficiently and scalably provide strong consistency and liveness properties for survivable storage.

References

[1] *RSA Data Security, Inc. MD5 Message-Digest Algorithm.* http://www.ietf.org/rfc/rfc1321.txt.

[2] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: federated, available, and reliable storage for an incompletely trusted environment. *Symposium on Operating Systems Design and Implementation* (Boston, MA, 09–11 December 2002), pages 1–15. USENIX Association, 2002.

[3] Khalil Amiri, Garth A. Gibson, and Richard Golding. *Scalable concurrency control and recovery for shared storage arrays*. CMU–CS–99–111. Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, February 1999.

[4] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. *ACM Symposium on Operating System Principles* (Asilomar, Pacific Grove, CA). Published as *Operating Systems Review*, **25**(5):198–212, 13–16 October 1991.

[5] Nina T. Bhatti and Richard D. Schlichting. A system for constructing configurable high-level protocols. *ACM SIGCOMM Conference* (Cambridge, MA, 28 August–1 September). Published as *Computer Communications Review*, **25**(4):138–150. ACM, 1995.

[6] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. *Symposium on Operating Systems Design and Implementation* (New Orleans, LA, 22–25 February 1999), pages 173–186. ACM, 1998.

[7] Flaviu Cristian, Houtan Aghili, Raymond Strong, and Danny Dolev. Atomic broadcast: from simple message diffusion to Byzantine agreement. *Information and Computation*, **118**(1):158–179, April 1995.

[8] Flaviu Cristian and Christof Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, **10**(6):642–657. IEEE, June 1999.

[9] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. *ACM Symposium on Operating System Principles* (Chateau Lake Louise, AB, Canada, 21–24 October 2001). Published as *Operating System Review*, **35**(5):202–215, 2001.

[10] Wei Dai. *Crypto++ reference manual*. http://cryptopp.sourceforge.net/docs/ref/.

[11] Svend Frolund, Arif Merchant, Yasushi Saito, Susan Spence, and Alistair Veitch. FAB: enterprise storage systems on a shoestring. *Hot Topics in Operating Systems* (Lihue, HI, 18–21 May 2003), pages 133–138. USENIX Association, 2003.

[12] Juan A. Garay and Kenneth J. Perry. A continuum of failure models for distributed computing. *International Workshop on Distributed Algorithms* (Halifax, Isreal, 02–04 November 1992), volume 647, pages 153–165. Springer Verlag, 1992.

[13] Li Gong. Securely replicating authentication services. *International Conference on Distributed Computing Systems* (Newport Beach, CA), pages 85–91. IEEE Computer Society Press, 1989.

[14] Garth R. Goodson, Jay J. Wylie, Gregory R. Ganger, and Michael K. Reiter. *Efficient Byzantine-tolerant erasure-coded storage*. CMU–PDL–03–104. Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA, December 2003.

[15] Garth R. Goodson, Jay J. Wylie, Gregory R. Ganger, and Michael K. Reiter. *The safety and liveness properties of a protocol family for versatile survivable storage infrastructures*. CMU–PDL–03–105. Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA, March 2004.

[16] Cary G. Gray and David R. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. *ACM Symposium on Operating System Principles* (Litchfield Park, AZ, 3–6 December 1989). Published as *Operating Systems Review*, **23**(5):202–210, December 1989.

[17] Jim N. Gray. Notes on data base operating systems. In , volume 60, pages 393–481. Springer-Verlag, Berlin, 1978.

[18] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages*, **13**(1):124–149. ACM Press, 1991.

[19] Maurice P. Herlihy and J. D. Tygar. How to make replicated data secure. *Advances in Cryptology - CRYPTO* (Santa Barbara, CA, 16–20 August 1987), pages 379–391. Springer-Verlag, 1987.

[20] Maurice P. Herlihy and Jeanette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, **12**(3):463–492. ACM, July 1990.

[21] Matti A. Hiltunen, Vijaykumar Immanuel, and Richard D. Schlichting. Supporting customized failure models for distributed software. *Distributed Systems Engineering*, **6**(3):103–111, September 1999.

[22] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, **6**(1):51–81, February 1988.

[23] Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM*, **45**(3):451–500. ACM Press, May 1998.

[24] Hugo Krawczyk. Secret sharing made short. *Advances in Cryptology - CRYPTO* (Santa Barbara, CA, 22–26 August 1993), pages 136–146. Springer-Verlag, 1994.

[25] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaten, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: an architecture for global-scale persistent storage. *Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, 12–15 November 2000). Published as *Operating Systems Review*, **34**(5):190–201, 2000.

[26] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, **6**(2):213–226, June 1981.

[27] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, **4**(3):382–401. ACM, July 1982.

[28] Edward K. Lee and Chandramohan A. Thekkath. Petal: distributed virtual disks. *Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, 1–5 October 1996). Published as *SIGPLAN Notices*, **31**(9):84–92, 1996.

[29] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp file system. *ACM Symposium on Operating System Principles* (Pacific Grove, CA, 13–16 October 1991). Published as *Operating Systems Review*, **25**(5):226–238, 1991.

[30] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed Computing*, **11**(4):203–213. IEEE, 1998.

[31] Dahlia Malkhi, Michael K. Reiter, Daniela Tulone, and Elisha Ziskind. Persistent objects in the Fleet system. *DARPA Information Survivability Conference and Exposition* (Anaheim, CA, 12–14 January 2001), pages 126–136. IEEE, 2001.

[32] Jean-Philippe Martin, Lorenzo Alvisi, and Michael Dahlin. Minimal byzantine storage. *International Symposium on Distributed Computing* (Toulouse, France, 28–30 October 2002), 2002.

[33] David L. Mills. *Network time protocol (version 3)*, RFC–1305. IETF, March 1992.

[34] Shivakant Mishra, Christof Fetzer, and Flaviu Cristian. The Timewheel Group Communication System. *IEEE Transactions on Computers*, **51**(8):883–899. IEEE, August 2002.

[35] Robert Morris. Storage: from atoms to people. *Keynote address at Conference on File and Storage Technologies*, January 2002.

[36] Sape J. Mullender. A distributed file service based on optimistic concurrency control. *ACM Symposium on Operating System Principles* (Orcas Island, Washington). Published as *Operating Systems Review*, **19**(5):51–62, December 1985.

[37] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: a read/write peer-to-peer file system. *Symposium on Operating Systems Design and Implementation* (Boston, MA, 09–11 December 2002). USENIX Association, 2002.

[38] NIST Internetworking Technology Group. *NIST Net*. http://snad.ncsl.nist.gov/itg/nistnet/.

[39] Brian D. Noble and M. Satyanarayanan. *An emperical study of a highly available file system*. Technical Report CMU–CS–94–120. Carnegie Mellon University, February 1994.

[40] Evelyn Tumlin Pierce. *Self-adjusting quorum systems for byzantine fault tolerance*. PhD thesis, published as Technical report CS–TR–01–07. Department of Computer Sciences, University of Texas at Austin, March 2001.

[41] Sean Quinlan and Sean Dorward. Venti: a new approach to archival storage. *Conference on File and Storage Technologies* (Monterey, CA, 28–30 January 2002), pages 89–101. USENIX Association, 2002.

[42] Michael O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, **36**(2):335–348. ACM, April 1989.

[43] David P. Reed. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems*, **1**(1):3–23. ACM Press, February 1983.

[44] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, **10**(1):26–52. ACM Press, February 1992.

[45] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. *ACM Symposium on Operating System Principles* (Chateau Lake Louise, AB, Canada, 21–24 October 2001). Published as *Operating System Review*, **35**(5):188–201. ACM, 2001.

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

[47] Adi Shamir. How to share a secret. *Communications of the ACM*, **22**(11):612–613. ACM, November 1979.

[48] Jennifer G. Steiner, Jeffrey I. Schiller, and Clifford Neuman. Kerberos: an authentication service for open network systems. *Winter USENIX Technical Conference* (Dallas, TX), pages 191–202, 9–12 February 1988.

[49] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, and Gregory R. Ganger. Self-securing storage: protecting data in compromised systems. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 23–25 October 2000), pages 165–180. USENIX Association, 2000.

[50] Philip Thambidurai and You keun Park. Interactive consistency with multiple failure modes. *Symposium on Reliable Distributed Systems* (10–12 October 1988), pages 93–100. IEEE, 1988.

[51] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: a scalable distributed file system. *ACM Symposium on Operating System Principles* (Saint-Malo, France, 5–8 October 1997). Published as *Operating Systems Review*, **31**(5):224–237. ACM, 1997.

[52] Hakim Weatherspoon and John D. Kubiatowicz. Erasure coding vs. replication: a quantitative approach. *First International Workshop on Peer-to-Peer Systems (IPTPS 2002)* (Cambridge, MA, 07–08 March 2002), 2002.

[53] Jay J. Wylie, Michael W. Bigrigg, John D. Strunk, Gregory R. Ganger, Han Kiliccote, and Pradeep K. Khosla. Survivable information storage systems. *IEEE Computer*, **33**(8):61–68. IEEE, August 2000.