Efficient consistency for erasure-coded data via versioning servers

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

March 2003
CMU-CS-03-127

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This paper describes the design, implementation and performance of a family of protocols for survivable, decentralized data storage. These protocols exploit storage-node versioning to efficiently achieve strong consistency semantics. These protocols allow erasure-codes to be used that achieve network and storage efficiency (and optionally data confidentiality in the face of server compromise). The protocol family is general in that its parameters accommodate a wide range of fault and timing assumptions, up to asynchrony and Byzantine faults of both storage-nodes and clients, with no changes to server implementation or client-server interface. Measurements of a prototype storage system using these protocols show that the protocol performs well under various system model assumptions, numbers of failures tolerated, and degrees of reader-writer concurrency.

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. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory or the U.S. Government.

Keywords: Decentralized storage, consistency protocol, versioning servers, distributed file systems

1 Introduction

Survivable storage systems spread data redundantly across a set of decentralized storage-nodes, in an effort to ensure the availability of that data despite the failure or compromise of storage-nodes. Perhaps one of the most difficult aspects of designing a survivable storage system is predicting the faults, threats, and environments to which it will be subjected. A system can be designed pessimistically, i.e., built upon weak assumptions, though often this comes with high performance cost. Alternatively, the system can optimistically “assume away” certain environments or threats to gain performance.

This paper describes a family of consistency protocols that exploit data versioning within storage-nodes to efficiently provide strong consistency for erasure-coded data. The protocol family covers a broad range of system model assumptions with no changes to the client-server interface, server implementations, or system structure. For each combination of key system model assumptions (crash vs. Byzantine servers, crash vs. Byzantine clients, synchronous vs. asynchronous communication, total number of failures), there is a suitable member of the protocol family. Protocol instances are distinguished by their read and write thresholds (minimum number of storage-nodes contacted to ensure correctness), read and write policies (actual number of storage-nodes contacted), and data encoding mechanisms. Weaker assumptions lead to larger thresholds, more demanding policies, and more expensive encoding mechanisms However, for any given set of system assumptions, the protocol is reasonably efficient. The protocol scales with its requirements—it only does work necessitated by the system model.

Each protocol in the family works roughly as follows. To perform a write, clients write time-stamped fragments to at least a write threshold of storage-nodes. Storage-nodes keep all versions of fragments they are sent. To perform a read, clients fetch the latest fragment versions from a read threshold of storage-nodes. The client determines whether the fragments comprise a consistent, complete write; usually, they do. If they do not, additional fragments or historical fragments are fetched, until a consistent, complete write is observed. Only in cases of failures (storage-node or client) or read-write concurrency is additional overhead incurred to maintain consistency.

In the common case, the consistency protocol proceeds with little overhead beyond actually reading and writing data fragments. More specifically, the protocol is efficient in three ways. First, by using m-of-n erasure codes (i.e., m-of-n fragments are needed to reconstruct the data), a decentralized storage system can tolerate multiple failures with much less network bandwidth (and storage space) than with replication [58, 60]. Second, by matching the value of m to the read threshold size, no extra communication is required for consistency in the common case. A client crash during a write operation, a misbehaving server, and read-write concurrency can introduce protocol overhead. The first two should be very rare. As well, most studies of distributed storage systems (e.g., [6, 14, 26, 44]) indicate that minimal writer-writer and writer-reader sharing occurs (usually well under 1% of operations). Third, the execution of the protocol falls on the clients, leaving storage-nodes to the servicing of simple read and write requests. This results in scalability gains by following the well-known scalability principal of shifting work from servers to clients [24]. Clients are responsible for encoding and decoding data, detecting potential consistency problems, and resolving them.

This paper is comprised of two parts: (i) a partial development of the consistency protocol family and (ii) an investigation of family members’ performance behavior in a 20 node cluster. Our partial development of the protocol family focuses on an instance suitable for use in an asynchronous system that tolerates the Byzantine failure of storage-nodes and clients. In this very weak system model, the protocol offers strong consistency semantics, namely a variant of linearizability in which read operations are permitted to abort (in presumably uncommon circumstances). We identify various other members of the protocol family that eliminate read aborts, are better suited for stronger system models (e.g., synchrony and non-Byzantine failures), and that offer other features.

The second part of the paper investigates the performance of representative members of the protocol

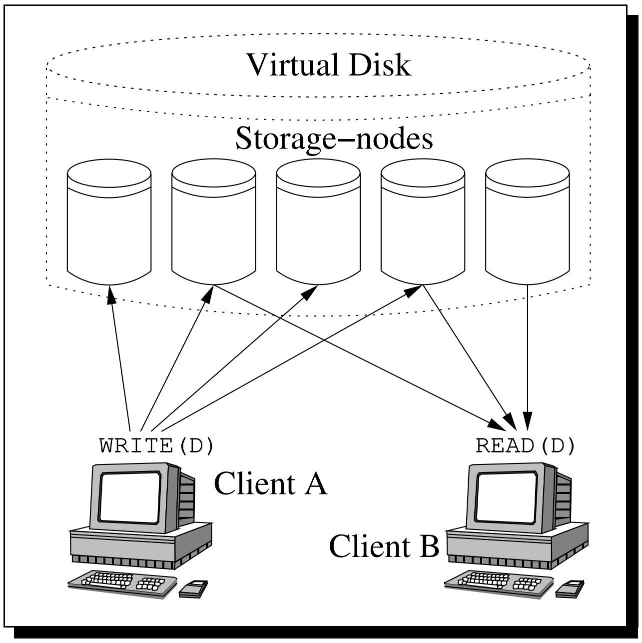

Figure 1: **High-level architecture for survivable storage.** Spreading redundant data across independent storage-nodes increases the likelihood of data surviving failures of storage nodes. Clients update multiple servers to complete a write and (usually) readers fetch information from multiple servers to complete a read.

family. The response time of a mixed workload is shown to scale nearly linearly up to as many as 20 storage-nodes for three of the four selected protocols. The throughput that a 7 storage-node configuration can service is shown to scale nearly linearly as clients are added to the system until the network begins to saturate. The impact of read-write concurrency on system performance is shown to be nominal for a highly concurrent workload.

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 who host the data-item. To read *D*, Client B issues read requests to an overlapping subset of storage-nodes. This basic scheme allows readers and writers to successfully access data-items even when subsets of the storage-nodes have failed. To provide reasonable storage semantics, however, the system must guarantee that readers see consistent answers. For example, assuming no intervening writes, two successful reads of *D* should produce the same answer (the most recent write) independent of which subset of storage-nodes are contacted.

2.1 Decentralized storage

A common data distribution scheme used in such storage systems is data replication. That is, a writer stores a replica of the new data-item value at each storage-node to which it sends a write request. Since each storage-node has a complete instance of the data-item, the main difficulty is identifying and retaining the most recent instance. It is often necessary for a reader to contact multiple storage-nodes to ensure that it sees the most recent instance. Examples of distributed storage systems that use this design include Harp [34], Petal [33], BFS [8], and Farsite [1].

Some distributed storage systems spread data among storage-nodes more space-efficiently, using erasure coding or even simple striping. With striping, a data-item is divided into fragments, all of which are needed to reconstruct the entire data-item. With erasure coding, a data-item is encoded into a set of fragments such that any sufficient subset allows reconstruction. (A primitive version of this is the RAID mechanism [45] of striping plus parity computed across the stripe units.) With these data distribution schemes, reads require fragments from multiple servers. Moreover, the set of fragments must correspond to the same write operation or else the reconstituted "data" will be incoherent. Examples of distributed storage systems that use erasure coding include Zebra [21], SwiftRAID [35], Intermemory [10], Cheops [4], Myriad [9], and PASIS [59, 60].

A challenge that must be confronted in the design of decentralized storage systems is that of partially completed write operations. Write operations in progress and incomplete write operations by clients that crash are both instances of partially completed write operations. A common approach to dealing with partial writes in non-Byzantine-tolerant systems is two-phase commit [20]. This works for both replication and erasure coding, but adds a round-trip for every write. The partial write problem can also be addressed via "repair," which involves a client or storage-node distributing a partially written value to storage-nodes that have not seen it. This is only an option for systems using erasure codes when partial writes provide enough information to reconstruct the original data.

A common approach to dealing with concurrency is to suppress it, either via leases [19] or optimistic concurrency control [31]. Alternately, many non-Byzantine-tolerant systems (e.g., Harp [34] and Petal [33]) serialize their actions through a primary storage-node, which becomes responsible for completing the update.

An alternate approach to handling both problems is to have the data stored on storage-nodes be immutable [50, 51]. By definition, this eliminates the difficulties of updates for existing data. In doing so, it shifts the problem up one level; an update now consists of creating a new data-item and modifying the relevant name to refer to it. Decoupling the data-item creation from its visibility simplifies both, but making the metadata service fault-tolerant often brings back the same issues. Examples of systems that use this model include SWALLOW [50], Amoeba [42], and most of the recent peer-to-peer file systems (e.g., Past [52], CFS [12], Farsite [1], and the archival portion of Oceanstore [30]).

Ivy [43] provides decentralized read/write access to immutable stored data in a fashion similar to our approach. Per-client update logs (which are similar to version histories) are merged by clients at read time (which is similar to write classification in our protocol). Ivy differs from our protocol in that it does not provide strong consistency semantics in the face of data redundancy or concurrent updates. Whereas certain failures may result in unclassifiable writes in our protocol, correct operations may lead to unresolvable situations in Ivy (in this way, Ivy is similar to Coda [25]).

Amiri et al. [4, 5] use a "stripe map" to communicate to storage-nodes the set of other storage-nodes that host related stripe units. In the Palladio project [16], stripe maps enables the masking of partial writes by clients. Since client writes employ two-phase commit, storage-nodes can detect a failure of the client using a timeout. Upon client failure detection, the stripe map enables the storage-nodes to perform a reconciliation protocol that results in either the write being completed (repaired) or aborted (deleted at all nodes involved). In our protocol, when repair is allowed, it is initiated by clients and made possible by retaining versions on the storage-nodes. If repair is not allowed, each subsequent read operation requires the client to resolve the partial write operation.

2.2 Byzantine fault-tolerance

Most systems that tolerate Byzantine client and server failures use Byzantine fault-tolerant agreement to maintain replicated state machines [53]. Long believed to be too costly for extensive use in practice, this approach was recently employed by Castro and Liskov [8] to implement a reasonably-performing replicated NFS service. Their implementation is well suited for achieving consistent transition between immutable versions, as demonstrated by its use in Farsite [1]. Our versioning-based protocols are enabled by the fact that storage actions (read and write) are simpler than arbitrary state machines. An alternative to replicated state machines is Byzantine quorum systems [38], of which our protocols can be viewed as employing a particular type. Byzantine quorums have been used to implement shared objects with semantics similar to those offered by replicated state machines (e.g., [37]) and with correspondingly higher cost than we encounter here.

Herlihy and Tygar [22] were perhaps the first to apply quorums to the problem of protecting the confidentiality and integrity of replicated data against a threshold of Byzantine-faulty servers, as we do here. However, in contrast to our work, that work did not focus on achieving strong data semantics in the face of concurrent access, did not admit the full range of system models that we consider here, and did not include an implementation or performance analysis. As such, our contributions are substantially different.

2.3 Consistency semantics

Our primary target consistency semantics (linearizability [23] with read aborts) have been studied previously. Notably, Pierce [47] presents a protocol implementing these semantics in a decentralized storage system using replication. This protocol is achieved by conjoining a protocol that implements pseudo-regular semantics (regular semantics [32] with read aborts) with a "write-back" protocol (repair). The penultimate step of a read operation is to write-back the intended return value, which ensures that the return value of the read operation is written to a full quorum before it is returned. Our protocols go beyond this work by accommodating erasure-coded data and providing greater efficiency: e.g., our common case read operation is a single round. Initial work on our protocol is described in [18].

Versioning storage-nodes in our protocol provide capabilities similar to "listeners" in the recent work of Martin, et al. [39]. The listeners protocol guarantees linearizability in a decentralized storage system. A read operation establishes a connection with a storage-node. The storage-node sends the current data-item value to the client. As well, the storage-node sends updates it receives back to the client, until the client terminates the connection. Thus, a reader may be sent multiple versions of a data item. In our protocol, readers look backward in time via the versioning storage-nodes, rather than listening into the future. Looking back in time is more message-efficient in the common case, yields a lighter-weight server implementation, and does not necessitate repair to deal with client failures.

3 System model

We describe the system infrastructure in terms of *clients* and *storage-nodes*. There are N storage-nodes and an arbitrary number of clients in the system.

We say that a client or storage-node is *correct* in an execution if it satisfies its specification throughout the execution. A client or storage-node that deviates from its specification is said to *fail*. We assume a hybrid failure model for storage-nodes. Up to t storage-nodes may fail, $b \leq t$ of which may be Byzantine faults; the remainder are assumed to crash benignly. A client or storage node that does not exhibit a Byzantine failure (it is either correct or crashes) is said to be *benign*. We assume that Byzantine clients and storage-nodes are computationally bounded so that we can employ cryptographic primitives (e.g., cryptographic hash functions and encryption).

We assume that communication between clients and storage-nodes is point-to-point, reliable, and authenticated: A correct storage-node (client) receives a message from a correct client (storage-node) if and only if that client (storage-node) sent it. When convenient, we will represent communication using `SEND` and `RECEIVE` primitives. A message `RECEIVE`'d bears an identifier of client/storage-node from which it was received.

We consider both synchronous and asynchronous models. In an asynchronous system, we make no assumptions about message transmission delays or the execution rates of clients or storage-nodes. In contrast, in a synchronous system, there are known bounds on message transmission delays between correct clients/storage-nodes and their execution rates. As well, in the synchronous model, we assume that clients and storage-nodes have loosely synchronized clocks (i.e., in the synchronous model, all clocks are synchronized to within some constant, τ, of the same value). Protocols to achieve approximate clock synchronization in today's networks are well known, inexpensive, and widely deployed [40, 41]. Alternately, a distinct synchronous communication path can be assumed for clock synchronization (e.g., GPS).

There are two types of *operations* in the protocol — *read operations* and *write operations* — both of which operate on *data-items*. Clients perform read/write operations that issue multiple read/write *requests* to storage-nodes. A read/write request operates on a *data-fragment*. A data-item is *encoded* into data-fragments. Requests are *executed* by storage-nodes; a correct storage-node that executes a write request is said to *host* that write operation.

Clients encode data-items in an erasure-tolerant manner; thus the distinction between data-item and data-fragment. We only consider threshold erasure codes in which any m of the n encoded data-fragments can decode the data-item. Examples of such codes are replication, Reed-Solomon codes [7], secret sharing [54], RAID 3/4/5/6 [45], information dispersal (IDA) [49], short secret sharing [29], and "tornado" codes [36].

Storage-nodes provide fine-grained versioning, meaning that a correct storage-node hosts a version of the data-fragment for each write request it executes. Storage-nodes offer interfaces to write a data-fragment at a specific logical time, to query the greatest logical time of a hosted data-fragment, to read the hosted data-fragment with the greatest logical time, and, to read the hosted data-fragment with the greatest logical time at or before some logical time.

4 Consistency protocol

We have developed a family of consistency protocols that efficiently support erasure-coded data-items by taking advantage of versioning storage-nodes. Members of the protocol family are differentiated based on the system model assumed (asynchronous or synchronous, type of storage-node and client failure), the specified protocol thresholds (write and read thresholds), the specified write and read policy, additional encode mechanisms employed, and whether repair is performed. We sketch the protocol at a high level, then give more details about the asynchronous protocol under the hybrid failure model. Other members of the protocol family are reductions or slight modifications of the asynchronous consistency protocol. In Section 4.4, we discuss other protocols from the same family as the asynchronous protocol.

At a high level, the protocol proceeds as follows. Logical timestamps are used to totally order all write operations and to identify write requests from the same write operation across storage-nodes. For each write, a logical timestamp is constructed that is guaranteed to be unique and greater than that of the *latest complete write* (the complete write with the highest timestamp).

To perform a read operation, clients issue read requests to a set of storage-nodes. Once at least a read threshold of storage-nodes reply, the client identifies the *candidate*, which is the data-item version returned with the greatest logical timestamp. The set of read responses that share the timestamp of the candidate are the *candidate set*. The read operation *classifies* the candidate as complete, partial or unclassifiable. 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 *partial* (i.e., not complete), the candidate is discarded, a new candidate is identified, previous data-item versions are requested, and classification begins anew; this sequence may be repeated. If information has been solicited from all possible storage-nodes and the candidate remains unclassifiable, the read operation aborts.

```
READ() :
 1: r := MAX[R_min, W_min]
 2: StorageNodeSet := {1, ..., N}
 3: ResponseSet := DO_READ(StorageNodeSet, r, *)
 4: loop
 5:   ⟨CandidateSet, LT_C⟩ := CHOOSE_CANDIDATE(ResponseSet)
 6:   if (|CandidateSet| ≥ W_min) then
 7:     /* Classify candidate as complete */
 8:     Data := DECODE(CandidateSet)
 9:     RETURN(Data)
10:   else if (|CandidateSet| + (N − |ResponseSet|) < W_min − b) then
11:     /* Classify candidate as partial, determine new candidate */
12:     ResponseSet := ResponseSet − CandidateSet
13:     ⟨CandidateSet, LT_C⟩ := CHOOSE_CANDIDATE(ResponseSet)
14:     ResponseSet := ResponseSet ∪
            DO_READ(StorageNodeSet − ResponseSet,
                    r − |ResponseSet|, LT_C)
15:   else if (r < R) then
16:     /* Candidate is unclassifiable, must read more */
17:     INCREMENT[r]
18:     ResponseSet := ReponseSet ∪
            DO_READ(StorageNodeSet − ResponseSet,
                    r − |ReponseSet|, LT_C)
19:   else
20:     /* Candidate is unclassifiable and cannot read more */
21:     RETURN(⊥)
22:   end if
23: end loop
```

```
WRITE(Data) :
 1: LT := GET_TIME()
 2: LT := MAKE_TIMESTAMP(LT)
 3: {D_1, ..., D_N} := ENCODE(Data)
 4: for all StorageNode ∈ {1, ..., N} do
 5:   SEND(StorageNode, WRITE_REQUEST, LT, D_i)
 6: end for
 7: repeat
 8:   ResponseSet := ResponseSet ∪
          RECEIVE(StorageNode, WRITE_RESPONSE)
 9: until (|ResponseSet| == W)
```

```
DO_READ(ReadSet, ReturnCount, LT) :
 1: for all StorageNode ∈ ReadSet do
 2:   SEND(StorageNode, READ_REQUEST, LT)
 3: end for
 4: repeat
 5:   ResponseSet := ResponseSet ∪
          RECEIVE(StorageNode, READ_RESPONSE)
 6: until (|ResponseSet| == ReturnCount)
 7: RETURN(ResponseSet)
```

Figure 2: **Asynchronous consistency protocol pseudo-code.**

4.1 Asynchronous protocol

We describe the asynchronous protocol in detail to give an intuition about the properties of the protocol, and the necessary conditions to achieve the properties. Pseudo-code for the asynchronous protocol is given in Figure 2. Terms used in the pseudo-code are explained in this section.

The write threshold, $W_{\min}$, defines a complete write operation: a write operation is *complete* the moment $W_{\min} - b$ benign storage-nodes have executed write requests. Intuitively, a consistency protocol must ensure that a write operation, once complete, replaces the previous value. The read threshold, $R_{\min}$, ensures that the protocol provides such consistency: a read operation must have at least $R_{\min}$ read responses before identifying the candidate. Essentially, the write and read thresholds establish threshold-quorums that intersect at some number of correct storage-nodes.

Beyond ensuring consistency, the protocol should enable complete writes to be classified as such. To this end, correct clients implement write and read policies. Correct clients send write requests to all N storage-nodes; they wait for $W \geq W_{\min}$ write responses before continuing. The value of W depends on the system model. For example, $W \leq N - t$ in an asynchronous system, since a client cannot wait for more responses: t storage-nodes could never respond. Correct clients consider read responses from up to $R \geq R_{\min}$ storage-nodes before aborting due to an unclassifiable candidate.

The major task of the read operation is to identify, and then classify, a candidate. To classify a candidate as complete, read requests from $W_{\min} - b$ correct storage-nodes that host the candidate must be collected. Two factors complicate classification: Byzantine storage-nodes and incomplete information. Up to b responses to R read requests can be arbitrary. As such, at least $W_{\min}$ responses supporting a candidate are required to classify the candidate as complete. To be safe, all storage-nodes for which there is no information must be assumed to not host the candidate. Line 6 of READ() tests the above condition to classify the candidate under consideration. Line 1 of READ() ensures that a read operation may complete with a single round of communication.

To classify a candidate as partial, the read operation must observe that it is impossible for $W_{\min} - b$ or more correct storage-nodes to host the candidate. To be safe, all storage-nodes for which there is no information must be assumed to host the write. Line 10 of READ() tests the partial classification condition.

In some cases, it is impossible for the read operation to classify the candidate as complete or partial. Interestingly, both partial and complete writes can be unclassifiable. Line 15 of READ() determines if the read policy (R) allows for more read requests, or if the operation must abort.

On line 1 of WRITE(), the function GET_TIME() is called. The current logical time of the data-item is determined by considering time query responses from at least $R_{\min}$ storage-nodes. The implementation of GET_TIME() is very similar to the function DO_READ(), except the *ResponseSet* consists solely of logical timestamps. The high bits of a logical timestamp are the *data-item time*. To make a timestamp, the client increments the data-item time and appends its client identifier and request identifier. The low bits of the timestamp distinguish write operations issued at the same logical data-item time.

The protocol supports erasure-coded data. The functions ENCODE() on line 3 of WRITE() erasure codes the data-item into N data-fragments. Conversely, function DECODE() on line 8 decodes m data-fragments, returning the data-item. Mechanisms that provide confidentiality and integrity guarantees can be employed in concert with erasure codes. Such mechanisms and the properties they provide are discussed more fully in Section 4.3.

4.2 Asynchronous protocol properties

We develop the protocol to have two major properties. First, it ensures the consistency of stored data-items. Second, it ensures that if all clients are correct, write and read operations that follow the write and read policies complete. In this section, we develop constraints on $W_{\min}$, $R_{\min}$, W, R, and N for the asynchronous protocol which are sufficient to achieve the desired properties. Formal statements of the desired properties and proofs that they are achieved will be provided in an extended version of this paper.

Linearizability with read aborts: Operations are *linearizable* if their return values are consistent with an execution in which each operation is performed instantaneously at a distinct point in time between its invocation and completion [23]. *Linearizability with read aborts* restricts the definition of linearizability to write operations and read operations that return a value. As such, read operations that abort are excluded from the schedule of linearized operations. Linearizability with read aborts is similar to Pierce's "pseudo-atomic consistency" [47].

To achieve linearizability with read aborts, the protocol must ensure that a read operation will notice the latest complete write operation as a candidate (assuming the read operation does not abort). Therefore, it is necessary that a read operation and a write operation "intersect" at at least one correct storage-node:

$$b + N < W_{\min} + R_{\min}. \tag{1}$$

Operations can terminate: A necessary condition for liveness in the asynchronous model is that,

$$W, R \leq N - t. \tag{2}$$

Without this constraint, write and read operations could await responses forever, thus never completing.

Constraints (1) and (2) yield lower bounds on $W_{\min}$, $R_{\min}$, and N (remember, $W_{\min} \leq W$, and $R_{\min} \leq R$):

$$t + b < W_{\min}; \tag{3}$$

$$t + b < R_{\min}; \tag{4}$$

$$2t + b < N. \tag{5}$$

Constraints (3) and (4) follow from substituting (2) into (1). Constraint (5) follows from the resulting lower bounds on $W_{\min}$ and $R_{\min}$.

Read operations can return a value: Read operations must be able to return a value despite the presence of Byzantine storage-nodes in the system. If Byzantine storage-nodes can always fabricate write requests that a read operation deems unclassifiable, then all read operations can be forced to abort. To ensure that Byzantine storage-nodes cannot always fabricate an unclassifiable candidate, a candidate set of size b must be classifiable as partial. To classify such a candidate as partial, it is necessary that,

$$b + (N - R) < W_{\min} - b. \tag{6}$$

Constraint (6) creates a relation between the read policy, R, and the definition of a complete write ($W_{\min}$). The less "aggressive" the read policy, the larger $W_{\min}$ must be to ensure that Byzantine storage-nodes cannot force all read operations to abort.

Assuming a read policy of $R = N - t - \kappa$, where κ determines the exact read policy R, constraint (6) becomes,

$$\begin{aligned} b + (N - (N - t - \kappa)) &< W_{\min} - b, \\ t + 2b + \kappa &< W_{\min}. \end{aligned} \tag{7}$$

Constraint (7) is more restrictive (on $W_{\min}$) than our previous lower bound, $t + b < W_{\min}$ (cf. (3)). The most aggressive read policy, $\kappa = 0$, yields $t + 2b < W_{\min}$.

Correct write operations are classifiable: Write operations by correct clients should be classifiable as complete. As such, the minimum intersection of write requests and read requests at correct storage-nodes guaranteed by the write and read policies must ensure sufficient information to classify the candidate as complete:

$$W + R - N - b \geq W_{\min}. \tag{8}$$

The term $W + R - N$ is the minimum intersection of write and read requests; up to b of the storage-nodes in the intersection may be Byzantine. Assuming the most aggressive write and read policies ($W = R = N - t$), constraint (8) becomes:

$$\begin{aligned} (N - t) + (N - t) - N - b &\geq W_{\min}, \\ N - 2t - b &\geq W_{\min}. \end{aligned} \tag{9}$$

Which implies, $W_{\min} + 2t + b \leq N$, a more restrictive lower bound on N, than constraint (5).

Complete write operations are decodable.: A complete write must be sufficient to decode a data-item. To achieve this property,

$$m \leq W_{\min} - b. \tag{10}$$

It is safe to let $m = 1$ (i.e., replication). But, we are interested in the space-efficiency offered by erasure codes. As such, the upper bound on m is of interest, since the size of data-fragments are inversely proportional to m.

Given the above constraints, there is still a large space from which to select values for $W_{\min}$, $R_{\min}$, W, R, and N. To limit the scope of this investigation, we focus on the smallest values of parameters $W_{\min}$, $R_{\min}$, and N that provide the desired properties. Assuming the most aggressive read and write policies, $W = R = N - t$, constraints (1), (7), and (9), demand that,

$$\begin{aligned} t + 2b &< W_{\min} \\ W_{\min} + 2t + b &\leq N \\ W_{\min} + R_{\min} &> N + b \end{aligned}$$

Thus, we focus on the threshold values $W_{\min} = t + 2b + 1$, $R_{\min} = 2t + 2b + 1$, and $N = 3t + 3b + 1$, which meet all of the constraints.

4.3 Data encoding

The implementation of `ENCODE()` and `DECODE()` are very important to the properties achieved by the protocol in Figure 2. Here, we describe three mechanisms that can be incorporated into their implementation, and the properties that these mechanisms provide.

Cross checksums: Cross checksums are used to detect if Byzantine storage-nodes corrupt data-fragments that they host. After erasure coding, a cryptographic hash of each data-fragment is computed, and the set of N hashes is concatenated to form the *cross checksum* of the data-fragments. The cross checksum is then appended to each data-fragment. A read operation uses the cross checksum to validate the integrity of data-fragments: each data-fragment that does not match its portion of the cross checksum is discarded. More than b matching instances of the cross checksum must be observed before it is used to validate data-fragments.

Cross checksums of erasure-coded data were proposed by Gong [17]. Krawczyk [28] extended cross checksums to make use of error-correcting codes; the space-efficiency of Krawczyk's *distributed fingerprints* comes at a cost in computation and complexity.

Validated cross checksums: In system models that admit Byzantine clients, cross checksums can be extended to detect Byzantine clients who write data-fragments that are not consistent (i.e., where different sets of m data-fragments reconstruct a different data-item). Given m data-fragments that are consistent with a cross checksum received from more than b storage nodes, the reading client regenerates the remaining $N-m$ data-fragments. If any of these $N-m$ generated data-fragments are inconsistent with the cross checksum, then decoding fails.

Note that this procedure requires that any m data-fragments uniquely determine the other $N-m$. This holds for the erasure-coding schemes that we employ.

When employing this mechanism, there are two options for where to place the cross checksum. In the first, the cross checksum is embedded as the low order bits of the logical timestamp of the write operation; we call this construction a *self-validating timestamp*. The second option is to leave the timestamp construction unchanged, and to append the cross checksum to each data-fragment as above. In this case, it is necessary to impose the constraint $2W_{\min} > N + b$ to prevent a Byzantine client from completing writes of two distinct data-items with the same logical time.

Short secret sharing: Short secret sharing [29] provides data confidentiality so long as fewer than m data-fragments are observed. Thus, if $m > b$, Byzantine storage-nodes are prevented (computationally) from leaking information about the contents of data-items. Information about access patterns and data-item size can still be leaked, of course.

4.4 Other members of protocol family

There are several branches of the protocol family which yield interesting protocols: achieved properties can be weakened, stronger assumptions can be made, and repair can be leveraged.

The asynchronous protocol described, achieves strong liveness guarantees. Granting Byzantine storage-nodes the power to prevent read operations from completing creates a larger set of valid protocol thresholds — these are attractive if the lower bounds on $W_{\min}$, $R_{\min}$, and N are of concern. We do not experiment with protocols that do not achieve the above properties in this work.

The assumption of synchrony defines another branch of the protocol family tree. This assumption introduces the ability to have loosely synchronized clocks and to detect crashed storage-nodes via timeouts. The former is used to make `GET_TIME()` a local operation (cf. line 1 of the write operation). In this case, global time is used as the high bits of the logical timestamp. This reduces a write operation to a single round

Protocol	$W_{\min}$	$R_{\min}$	N	m
Asynch.	$t+2b+1$	$2t+2b+1$	$3t+3b+1$	$t+b+1$
Synch.	$t+b+1$	$t+b+1$	$2t+b+1$	$t+1$
Asynch. + Repair	$t+2b+1$	$t+b+1$	$2t+2b+1$	$b+1$
Synch. + Repair	$t+b+1$	$t+b+1$	$2t+b+1$	$b+1$

Table 1: **Example protocol instances.** The protocol thresholds are listed for the set of protocol instances evaluated in Section 6. All protocol instances listed employ the most aggressive write and read policy possible (e.g., $W = R = N - t$ in the asynchronous model).

of communication with storage-nodes. The definition of operation duration is extended by the clock skew of the clock synchronization protocol. Many have exploited the fact that messaging overhead and round-trip counts can be reduced with loosely synchronized clocks (e.g, Adya et. al [2, 3]).

Since crashed storage-nodes can be detected in the synchronous model, $W = N$ and $R = N$ are allowable write and read policies. Consequently, the lower bounds on protocol thresholds in the synchronous model are lower than in the asynchronous. As well, responses from all correct storage-nodes can be solicited; the lower bound on the range of candidate sets that are unclassifiable can be reduced to b (from $t+b$ in the asynchronous model).

Introducing repair into the protocol enables it to provide linearizability rather than linearizability with read aborts. To accomplish this, all unclassifiable candidates must be repairable. This constraint reduces the upper bound on m, making the protocol less space-efficient. However, the lower bound on N is also reduced; with repair, Byzantine storage-nodes need not be prevented from making complete write operations by correct clients unclassifiable. We include repairable protocols for the asynchronous and synchronous models in our experiments.

The members of the protocol family that we evaluate in Section 6 are listed in Table 1. The values of N and $W_{\min}$ in the asynchronous and asynchronous with repair systems are lower bounds. For synchronous systems, we experiment with thresholds we know to be correct (i.e., that achieve the desired properties), but we believe are not lower bounds (i.e., we have not completed the proof sketch for lower bounds yet, but our intuition is that N can be much lower).

4.5 Byzantine clients

If an authorized client is Byzantine, there is little a storage system can do to prevent the client from corrupting data. Such a client can delete data or modify it arbitrarily. The best a storage system can do is provide mechanisms that facilitate the detection of incorrect clients and the recovery from Byzantine client actions.

Storage-based intrusion detection could assist in the detection of malicious clients [46, 56]. As well, since the consistency protocol makes use of fine-grained versioning storage-nodes (i.e., self-securing storage [57]), recovery and diagnosis from detected storage intrusions is possible. Maliciously deleted data can be recovered, and arbitrarily modified data can be rolled back to its pre-intrusion state.

Encoding mechanisms can limit the power of Byzantine clients; however, they still have some power. Worse, collusion with Byzantine storage-nodes enables a Byzantine client to perform a write operation that is guaranteed to be unclassifiable (without repair, such operations result in read aborts). A Byzantine client can perform a large number of intentional partial write operations such that a subsequent read operation must issue many read requests to complete. Finally, a Byzantine client can perform write operations that result in integrity faults. In some situations, such faults cannot be masked, though they are always detected.

In a synchronous system, in which clients use synchronized clocks, Byzantine clients can perform write operations "in the future". Synchronizing storage-nodes clocks with client clocks would enable storage-nodes to bound how far into the future write operations can be performed (e.g., to 2τ).

5 Prototype implementation

This section describes a survivable block-store based on erasure-coding schemes, versioning storage nodes, and the consistency protocol described in Section 4. The client software can be configured to support different system model assumptions and failure tolerances. A given configuration affects the protocol thresholds and particular erasure-coding scheme utilized.

The system consists of clients and storage-nodes. The client module provides a block-level interface to higher level software, and uses a simple read-write RPC interface to communicate with storage-nodes. The client module is responsible for encoding and decoding data-item blocks to and from data-fragments, and for the execution of the consistency protocol. The storage-nodes are responsible for storing data-item fragments and their versions.

5.1 Storage-nodes

Storage-nodes provide storage and retrieval of data-fragment versions for clients. Table 2 shows the interface exported by a storage-node.

We use the Comprehensive Versioning File System [55] as the versioning storage-nodes.

Time: Storage-nodes provide an interface for retrieving the logical time of a data-item, but do not enforce its use. In the synchronous timing model, recall, clients are correct in using their local clocks as logical timestamps. To break ties, the client ID and request ID are concatenated to the logical timestamp.

Writes: In addition to data, each write request contains a *linkage record* and, optionally, data-item cross checksums. The cross checksums are stored with the data-fragment. The linkage record consists of the addresses of all the storage-nodes in the set of N for a specific data-item. Since there is no create call, the linkage information must be transmitted on each write request. Linkage records enable storage-nodes to perform decentralized garbage collection of old versions. Linkage records are similar to the "stripe maps" used in Cheops [4] and Palladio [16]. Indeed, the linkage record structure is introduced by Amiri and Golding in [5].

Reads: By default, a read request returns the data of the most current data-fragment version, as determined by the logical timestamps that accompany the write requests. To improve performance, read requests may also return a limited version history of the data-fragment being requested (with no corresponding data). Each history entry consists of a ⟨`client logical timestamp, cookie`⟩ tuple. This version history information allows clients to classify earlier writes without extra requests to storage-nodes. The cookie is an opaque data handle that can be returned to the storage-node on a subsequent read to efficiently request the data version that matches a given history entry.

Data versioning: The storage-node implementation uses a log-structured data organization to reduce the cost of data versioning. Like previous researchers [57], our experiences indicate that retaining every version and performing local garbage collection come with minimal performance cost (a few percent). Also, previous research ([48], [57]) indicates that the space required to retain multi-day version histories is feasible.

Garbage Collection: Pruning old versions, or garbage collection, is necessary to prevent capacity exhaustion of the backend storage-nodes. A storage-node in isolation, by the very nature of the protocol, cannot determine what local data-fragment versions are safe to garbage-collect. This is because write completeness is a property of a set of storage-nodes, and not of a single storage-node. An individual storage-node can garbage-collect a data-fragment version if there exists a later data-fragment version that is part of a complete write.

RPC Call	Description
`Read`	Read a block at/or before a logical timestamp (or latest version)
`Write`	Write a block with logical timestamp
`GetLTime`	Get logical data–item time of a block

Table 2: **Remote Procedure Call List.**

Storage-nodes are able to classify writes by executing the consistency protocol in the same manner as the client. Classification requires that storage-nodes know how to contact the other $N-1$ nodes hosting the write operation. Linkage records provide this information. As well, they also provide a means of access control; a storage-node can deny requests from all nodes not contained within the linkage record for a given data-fragment. In addition, implementation does not require the use of additional RPCs. Furthermore, garbage collection does not require the transfer of data, only that of history version information.

Policy issues remain regarding the frequency with which storage-nodes should perform garbage collection and the granularity at which it should be done.

5.2 Clients

The bulk of the consistency protocol is handled by the clients. The clients are responsible for encoding data blocks into data-fragments and storing them across a set of distributed storage-nodes. The clients are also responsible for enforcing the consistency properties through the correct classification of read requests.

5.2.1 Client interface and organization

In the current implementation, the client-side module is accessed through a set of library interface calls. These procedures allow applications to control the encoding scheme, the protocol threshold values (N, R, $R_{\min}$, W, $W_{\min}$), failure model (b, t, Byzantine clients), and timing model (synchronous or asynchronous), as well as more routine parameters such as block size. The client protocol routines are implemented in such a way that different parameters may be specified for different sets of data-items.

Writes: The client write path is implemented as follows. First, depending on the timing model, a logical timestamp is created. In the asynchronous case, this involves sending requests to at least $R_{\min}$ storage-nodes in order to retrieve the data-item time. In the synchronous model, the client's clock is read (all clients are synchronized using NTP [41]). A client specific request ID and client ID are then appended to create the logical timestamp.

Next, the data block is encoded into data-fragments using the specified scheme (see Section 5.2.2). A set of storage-nodes are then selected. If the write is a block overwrite, a list of storage-nodes must be passed into the write procedure, if not, a set of nodes is selected randomly from a list of well-known storage-nodes. Better procedures for selecting "good" storage-nodes are outside the scope of this paper.

Finally, write requests for each encoded data-fragment are issued to the set of N storage-nodes. Each write includes the linkage record, cross checksum, and logical timestamp. The write completes once $W_{\min}$ completions have been successfully returned. Upon completion, the linkage record is returned to the caller. The remaining $N-W$ responses are dropped by the client, although the writes may execute at the storage-nodes.

Reads: The client read path is somewhat more complicated, since it executes the resolution portion of the consistency algorithm. The application is expected to provide the linkage record naming the set of N storage-nodes from a previous write. Read requests are issued to the first $R_{\min}$ storage-nodes, so that systematic decoding of the data can be performed if all requests are successful. Once the $R_{\min}$ requests have completed, read classification begins.

Read classification proceeds as described in the pseudo code, in Figure 2, with a few optimizations. Since storage-nodes return a set of data-fragment version histories, the client need issue fewer read requests for timestamps of previous versions. Classification continues until either a complete read is found or the client exhausts the set of storage-nodes it can query.

Once classification completes and a complete read has been found, decoding of the data block can be attempted. To do so, it is necessary to have data from at least m data-fragments belonging to the same logical timestamp. If the data from fewer than m matching data-fragments are available (due to the use of version history information), it is necessary to request additional data from storage-nodes hosting the candidate write. Once m matching data-fragments have been collected, decoding is performed.

If cross checksums are in use (to tolerate Byzantine failures), checksum validation is first performed. If checksum validation fails, the integrity of the data-item cannot be guaranteed and the implementation currently returns an integrity fault. Similarly, if validated cross checksums or self-validating timestamps fail, an integrity fault occurs. If the data block is successfully decoded, it is returned to the client.

Metadata: For simplicity and modularity of the implementation, it is assumed that the client application maintains metadata as to where data blocks are stored (i.e., the linkage records). For example, to build a file system, linkage records could be stored within directories and inodes that are themselves stored by the block store. In this case, there must be some way of determining the root of the file system. For our experiments, a static set of N storage-nodes are used, obviating the need to save individual linkage records.

5.2.2 Encode and decode implementation

The encode and decode functions are tuned for efficiency. Erasure coding is used when availability is the focus and short secret sharing when confidentiality (from servers) is desired. This subsection describes both, as well as cryptographic algorithms.

Erasure codes: Although we have implementations of replication and RAID (i.e., single parity block), we focus on erasure codes that can protect against more than a single or double erasure.

Our erasure code implementation stripes the data-item across the first m data-fragments. Each *stripe-fragment* consists of $1/m$ the length of the original data-item.

Stripe-fragments are used to generate the *code-fragments*. Code-fragments are created by treating the m stripe-fragments as a vector of y values for some unique x value in a Galois Field — polynomial interpolation within the Galois Field based on the stripe-fragments yields the code-fragments. Each code-fragment has a unique x value as well. We use a Galois Field of size 2^8 so that each byte in the data-item corresponds to a single y coordinate. This decision limits us to 256 unique x coordinates (i.e., $N \leq 256$).

Since the polynomial is interpolated at the x coordinate points many times, interpolation coefficients are precalculated using "Newton's Formula" [27]. Given these $m \times (N-m)$ constants, each of the $N-m$ code-fragment interpolations require m multiplications and $2m$ additions. Multiplication in the Galois Field is implemented as a lookup table (i.e., a $2^8 \times 2^8$ B$=$ 64 KB lookup table is used). Addition in the Galois Field is just bitwise XOR. Although this implementation more closely resembles the algorithm described by Shamir for sharing secrets [54], we refer to it as information dispersal [49] because of its lack of secrecy.

Our implementation of polynomial interpolation was originally based on publicly available code for information dispersal [13]. We note that Hal Finney's 'secsplit.c' was acknowledged in [13]. We modified the source to make use of stripe-fragments and the lookup table. As well, some loops were unrolled so that addition (XOR) could occur on word boundaries, rather than byte boundaries.

Stripe-fragments make the erasure code a *systematic encoding*. If the first m data-fragments are passed into the decode function, no polynomial interpolation is required — the stripe-fragments need only be copied into the data-item buffer. Each stripe-unit passed in to decode reduces the number of fragments that must be interpolated during decode by one.

Short secret sharing: Our implementation of short secret sharing closely follows [29]. We use a cryptographically secure pseudo-random number generator to generate an encryption key. The key is used to encrypt the data-item buffer under AES in CBC mode. Data-fragments written to storage-nodes consist of a fragment based on the key and a fragment based on the encrypted data-item.

The key is erasure-coded using Secret Sharing [54]. In secret sharing, the data-item is treated as a data-fragment, $m-1$ random data-fragments are generated, and $N-m+1$ code-fragments are interpolated. The data-fragment based on the data-item is not stored (i.e., the key is not stored).

The portions of data-fragments that correspond to the encrypted data-item are generated using the erasure code technique outlined above.

To decode m data-fragments, the first portions are decoded via secret sharing to recover the key. The encrypted data-item is recovered via the normal decode technique for erasure-coded data described above. Finally, the key is used to decrypt the data-item.

Cryptographic algorithms: We use publicly available implementations of SHA1 [13], a cryptographically secure pseudo-random number generator (based on ANSI X9.17 Appendix C) [13], and AES [15]. We note that Steve Reid is acknowledged as the originator of the SHA1 code [13].

6 Evaluation

This section uses the prototype system to evaluate performance characteristics of the protocol family.

6.1 Experimental setup

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

Each storage-node is configured with 128 MB of data cache, and no caching is done on the clients. All experiments show results using write-back caching at the storage nodes, mimicking availability of 16 MB of non-volatile RAM. This allows us to focus on the overheads introduced by the protocol and not those introduced by the disk subsystem.

All experiments employ a similar base policy:

- Clients issue writes to all N storage-nodes, but wait for only $W_{\min}$ responses.
- Clients read the first $R_{\min}$ storage-nodes, where the first m data-fragments are stored, taking advantage of systematic encodings.
- Clients write in a manner such that the systematically encoded data-fragments (i.e., the stripe-fragments) are randomly distributed across storage-nodes, avoiding hot spots due to the above read policy.
- Clients keep a fixed number of read and write operations outstanding. Once an operation completes, a new operation is issued. Unless otherwise specified, requests are for 16 KB blocks with a 2:1 ratio of reads to writes.
- Information dispersal is employed in all experiments for which $b = 0$. Short-secret sharing and cross checksums are employed in all experiments for which $b > 0$.
- For the asynchronous model, clients fetch logical times from storage-nodes before issuing a write.

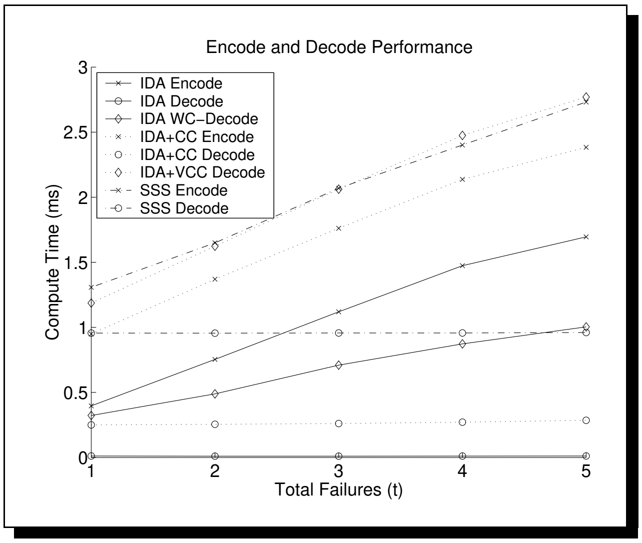

Figure 3: **Encode and decode of 16 KB blocks.** The erasure code parameters (N & m) are taken from the asynchronous model with $b = t$ (i.e., $N = 6t + 1, m = 2t + 1$). In the legend, IDA means information dispersal algorithm, SSS means short secret sharing, WC means worst case, CC means cross checksum, and VCC means validated cross checksum. Measurements of decode cost assume a systematic decoding using stripe-fragments, except for the worst case decode measurement.

6.2 Encode and decode performance

The *blowup* of an erasure code indicates the proportion of network and storage capacity consumed by the encoded data. The blowup has a significant impact on the performance of systems that use erasure codes. Note, in coding theory, the inverse of the blowup is called the *rate* of a code (i.e., blowup is not a standard term from coding theory).

For our basic erasure code, the blowup is $\frac{N}{m}$. In Figure 3, the blowup of the erasure codes are $\frac{6t+1}{2t+1}$. So, for $t = 2$, $m = 5$, $N = 13$, and the blowup of the encoding is $\frac{13}{5}$. Thus, a 16 KB block is encoded into 13 data-fragments, each of which is 3.2 KB in size. In total, these data-fragments consume 41.6 KB of network and storage capacity.

A more complicated formula for blowup is required for short secret sharing and cross checksums. Both mechanisms add additional overhead: the space for secret sharing the key and the space for the cross checksum respectively.

The three solid lines in Figure 3 show performance for IDA encode and decode. The cost of IDA encode grows with t because m grows with t; the cost of interpolating each code byte grows with m.

The common-case IDA decode exploits the systematic encoding to achieve near-zero computation cost (i.e., just the cost of `memcpy`). The IDA WC-Decode line represents the “worst case” in which only code-fragments are available. The decode, therefore, requires interpolation of m data-fragments. IDA encode is more expensive than worst case decode because encode must interpolate $N - m > m$ data-fragments.

The three dotted lines show the cost of encode and decode when IDA is combined with cross-checksums. The IDA+CC Encode line, when compared to the IDA Encode line, shows the cost of computing SHA1 digests of the N data-fragments. The IDA+CC Decode line, when compared to the IDA Decode line, shows the cost of computing m SHA1 digests of data-fragments. The IDA+VCC Decode line, representing validated cross checksums, shows the cost of generating $N - m$ data-fragments and taking their SHA1 digests on decode.

The two dashed lines show the cost of encoding and decoding a data-item with short secret sharing. The difference between the SSS Encode line and IDA Encode line, as well as the SSS Decode line and the IDA Decode line, is the cost of AES encryption of a 16 KB block, as well as key generation and secret

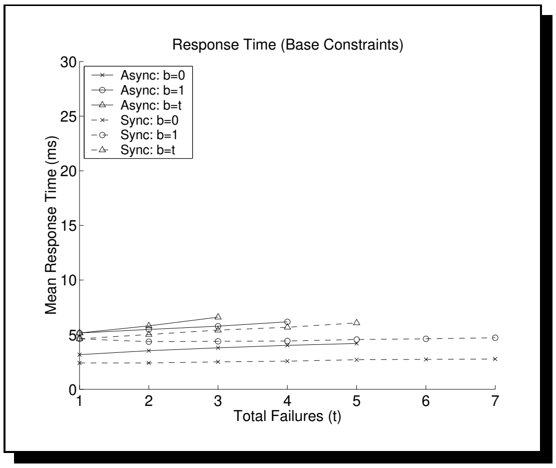

Figure 4: **Mean response time vs. Total failures (t) (Base Constraints).** Compares the mean response time of requests given different failure and timing models using non-repair thresholds. The number of storage-nodes grows with t, b, and the timing model.

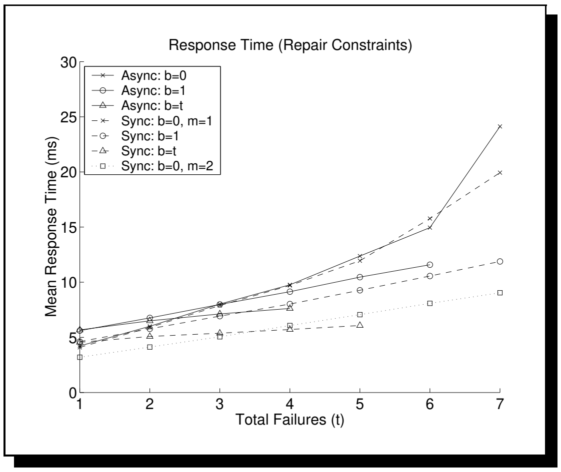

Figure 5: **Mean response time vs. Total failures (t) (Repair Constraints).** Compares the mean response time of requests given different failure and timing models using repair constraints.

sharing.

The cryptographic algorithms we use have the following performance characteristics on the testbed systems: AES 17.6 MB/s, SHA1 66.4 MB/s, and random number generation 4.6 MB/s.

6.3 Protocol overheads

To evaluate the overhead of the protocol under different system models, we consider asynchronous and synchronous members of the protocol family over a range of failure tolerance. We examine the mean response time of a single request from a single client, as well as the system throughput of multiple clients with multiple requests outstanding as failure tolerance scales.

6.3.1 Response time

Figure 4 shows mean response times for several system configurations using the base constraints (i.e., those without repair in Table 1). Figure 5 shows mean response times using the repair constraints.

The mean response times are plotted as a function of t, the total number of failures the system can handle. Notice, as t increases, so do $R_{\min}$, $W_{\min}$, and N (see Table 1 for equations). Three lines are shown for each of the asynchronous and synchronous models; those that tolerate $b = 0$, $b = 1$, and $b = t$ Byzantine failures out of the t tolerated failures. To put these parameters in context, consider the asynchronous protocol with $b = t$ in Figure 4. The line for this protocol is only plotted until $t = 3$. At this point, there are $N = 3t + 3b + 1 = 9 + 9 + 1 = 19$ storage-nodes.

The focus of these plots is on slope of the lines. The slope shows the protocol overheads in terms of network cost, since the number of storage-nodes that are accessed is increased. The flatness of the lines shows that most network transmissions can be fully overlapped. The slope of the lines is dictated by both the number of nodes and the blowup of the encoding.

The blowup of encoding directly affects the amount of data put on the wire. The impact of this can most clearly be seen in Figure 5 for the *b=0* lines. Both original lines have $m = 1$, i.e., data-fragment size equals the original data-item size — replication is being used. This gives an increase in the amount of network

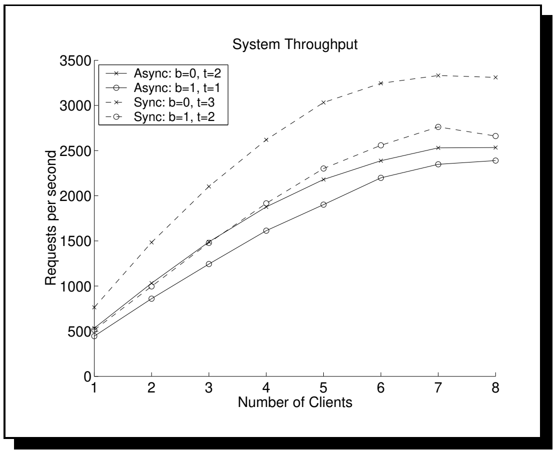

Figure 6: **Throughput vs. Client Load.** Compares the total system throughput of a mixed read/write workload as number of clients increase.

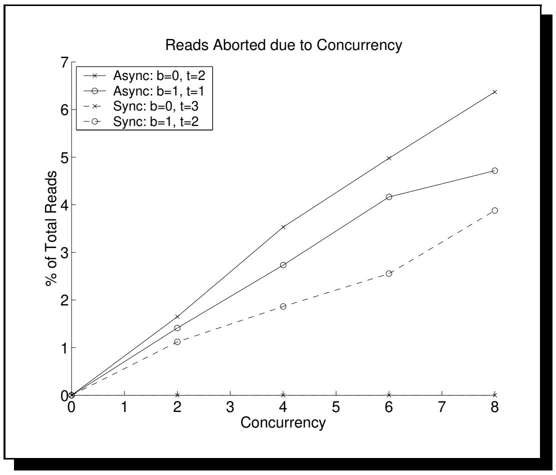

Figure 7: **Percentage of Reads Aborting vs. Concurrency.** Compares the percentage of reads that aborted (out of the total issued) due to the amount of concurrency in the system.

traffic proportional to N. To see the effect of m on response time, compare the following lines: *Sync: b=0, m=1* and *Sync: b=0, m=2*. The $m = 2$ line bumps $R_{\min}$, $W_{\min}$, N, and m up by one, reducing the data moved over the network by approximately a factor of 2. This results in a slightly lower availability, because there is one more storage-node that could fail, but much improved performance (by just over a factor of two for $t = 7$). This is a prime example of why supporting erasure codes is desirable.

The difference in cost between *Sync: b=0* and *Async: b=0* is the sum of the cost of short secret sharing versus IDA and the extra round-trip time to fetch the logical timestamp. The *b=t* lines have higher latencies because $R_{\min}$, $W_{\min}$, and N grow at a larger rate in relation to t.

6.3.2 Throughput

Figure 6 shows the throughput of a system of seven storage-nodes, in terms of requests/second, as a function of client load. Each client keeps four requests outstanding to increase the load generated. Each client accesses a distinct set of blocks, contributing to load without creating concurrent accesses to specific blocks.

As expected, throughput scales with the number of clients, until servers cannot handle more load, because of network saturation. The *b=0* configurations provides higher throughput because the blowup is smaller (i.e., the total network bandwidth consumed per operation is lower). The *Async* configurations generally saturate sooner than *Sync* configurations, because they also query servers for logical timestamp values on write requests.

6.4 Concurrency

To measure the effect of concurrency on the system, we measure multi-client throughput when accessing overlapping block sets. The experiment makes use of four clients; each client has four operations outstanding. Each client accesses a range of eight data blocks, with no two outstanding requests going to the same block. The number of blocks in a client's block range that clients share is denoted as the concurrency level; e.g., at a concurrency of six, six blocks are shared among all clients, while two blocks are reserved exclusively for each client. Blocks are chosen from a uniform random distribution, with a read/write ratio of 50%.

At the highest concurrency level (i.e., all eight blocks are in contention by all clients), we observed neither significant drops in the bandwidth nor significant increases in mean response time. However, the standard deviations of bandwidth and response time did rise slightly. Instead of plotting response time or throughput, the discussion focuses on the subtle interactions between the protocol and concurrency.

At the highest concurrency level, we observed that the initial candidate is classified as complete 89% of the time. In the remaining 11% of the cases, more information is required. The request latencies for the latter cases increased slightly due to the extra round trips necessary to complete classification. Since this occurs so seldom, and since round trip times are fairly small, the effect on mean response time is minimal.

The other case worthy of more detail is that of aborted reads due to read/write concurrency.

6.4.1 Aborted reads

A write operation that is in flight concurrently to a read operation may be observed by the read as partial (although it may later complete). If a read observes a partial write that is unclassifiable (according to the classification constraints), it has to abort. This is even true in the absence of failures. We call these types of aborts, *false aborts*. This section examines the effect of concurrency on false aborts.

Figure 7 shows the percentage of read operations that abort due to read-write concurrency as a function of system concurrency. The *Sync:b=0* configuration never aborts due to read-write concurrency in the case of no storage-node failures. This is because it is always able to obtain perfect information about the system. All writes are classifiable as either partial or as complete.

The other three configurations, *Async: b=0, b=1, Sync: b=1*, operate under imperfect information. Imperfect information comes from either having to tolerate liars (Byzantine nodes) or from not being able to wait for communication from all storage-nodes (asynchronous model). The number of read aborts in each of the above three configurations is proportional to the amount of information that is unavailable.

The protocol constraints preclude the *Sync: b=1* configuration from believing one response given it hears back from N. The *Async: b=0, t=2* configuration is precluded from communicating with $t = 2$ nodes. Finally the *Async: b=1, t=1* configuration is precluded from communicating with $t = 1$ nodes and believing an additional $b = 1$ nodes. The difference in magnitude between the two *Async* lines comes from the ratio of $W_{\min}$ to $R_{\min}$. In the *Async: b=1, t=1* configuration, there are $\binom{6}{4}$ combinations that a read could observe, while in the *Async: b=0, t=2* configuration there are only $\binom{5}{3}$ such combinations.

6.4.2 Retries

Since false aborts occur due to read/write concurrency (not failures), retrying reads may lead to the determination of a complete value as the system makes progress. Even the case of continuing high concurrency, one can reduce the probability of observing a complete write by varying the retry attempts. We experimentally ran a configuration that retried aborted reads. With a maximum of three retries, we observed one abort in ≈ 16000 sample requests. Continuing retries whenever new responses are observed can reduce this further.

Retries could be used to mimic the "listeners" approach of Martin et al. [39]. Instead of storage-nodes forwarding updates to clients that are listening, clients could poll storage-nodes for updates if a read operation encounters an unclassifiable candidate. In this manner, the client may determine that its read operation is concurrent to some write operations. Such information can be used to return any of the concurrent write operations. Such an approach could allow more read operations to complete or could reduce the number of retries necessary to complete.

7 Summary

This paper describes a family of consistency protocols for configurable, survivable storage systems using erasure codes for data storage. With the same storage-nodes and client-server protocol, thresholds can be configured to allow collections of blocks to be stored consistently and efficiently under a wide range of failure and communcation assumptions. Experiments with a prototype demonstrate efficiency across many points in this range.

Of the space the protocol covers, there is still much to explore. We plan to consider the protocol in a *timed asynchronous* system model [11]. In the timed asynchronous model, in which some assumptions are made about the expected message delay time, we expect to achieve similar properties as in the asynchronous model, but require lower threshold values to achieve such properties. We believe that tighter lower bounds can be derived for threshold values in the synchronous timing model, and we plan to ascertain these lower bounds. Finally, we plan to complete thorough analyses of the protocol performance under many conditions: wide area system model, network failures, storage-node failures, client failures, etc. Since the protocol is broadly applicable, there is much to explore.

8 Acknowledgements

We thank Craig Soules for his timely support of the versioned storage-node code base (the "S4" code). As well, we thank all of the members of the Parallel Data Lab who released cluster machines so that we could run experiments.

References

[1] 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.

[2] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. *ACM SIGMOD International Conference on Management of Data*, pages 23–34, 1995.

[3] Atul Adya and Barbara Liskov. Lazy consistency using loosely synchronized clocks. *ACM Symposium on Principles of Distributed Computing* (Santa Barbara, CA, August 1997), pages 73–82. ACM, 1997.

[4] Khalil Amiri, Garth A. Gibson, and Richard Golding. Highly concurrent shared storage. *International Conference on Distributed Computing Systems* (Taipei, Taiwan, 10–13 April 2000), pages 298–307. IEEE Computer Society, 2000.

[5] 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.

[6] 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.

[7] Elwyn Berlekamp. *Algebraic coding theory*. McGraw-Hill, New York, 1968.

[8] 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.

[9] Fay Chang, Minwen Ji, Shun-Tak A. Leung, John MacCormick, Sharon Perl, and Li Zhang. Myriad: cost-effective disaster tolerance. *Conference on File and Storage Technologies* (Monterey, CA, 28–30 January 2002), pages 103–116. USENIX Association, 2002.

[10] Yuan Chen, Jan Edler, Andrew Goldberg, Allan Gottlieb, Sumeet Sobti, and Peter Yianilos. A prototype implementation of archival Intermemory. *ACM Conference on Digital Libraries* (Berkeley, CA, 11–14 August 1999), pages 28–37. ACM, 1999.

[11] 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.

[12] 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.

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

[14] Deepinder S. Gill, Songian Zhou, and Harjinder S. Sandhu. *A case study of file system workload in a large–scale distributed environment*. Technical report CSRI–296. University of Toronto, Ontario, Canada, March 1994.

[15] Brian Gladman. *AES implementation*. http://fp.gladman.plus.com/cryptography_technology/rijndael/index.htm.

[16] Richard Golding and Elizabeth Borowsky. Fault-tolerant replication management in large-scale distributed storage systems. *Symposium on Reliable Distributed Systems* (Lausanne, Switzerland, 19–22 October 1999), pages 144–155. IEEE Computer Society, 1999.

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

[18] Garth R. Goodson, Jay J. Wylie, Gregory R. Ganger, and Michael K. Reiter. *Decentralized storage consistency via versioning servers*. Technical Report CMU–CS–02–180. September 2002.

[19] 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.

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

[21] John H. Hartman and John K. Ousterhout. The Zebra striped network file system. *ACM Transactions on Computer Systems*, **13**(3):274–310. ACM Press, August 1995.

[22] 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.

[23] 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.

[24] 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.

[25] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Symposium on Operating System Principles* (Asilomar, Pacific Grove, CA). Published as *Operating Systems Review*, **25**(5):213–225, 13–16 October 1991.

[26] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, **10**(1):3–25. ACM Press, February 1992.

[27] Donald Ervin Knuth. *Seminumerical algorithms*, volume 2. Addison-Wesley, 1981.

[28] Hugo Krawczyk. Distributed fingerprints and secure information dispersal. *ACM Symposium on Principles of Distributed Computing* (Ithaca, NY, 15–18 August 1993), pages 207–218, 1993.

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

[30] 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.

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

[32] Leslie Lamport. *On interprocess communication*. Technical report 8. Digital Equipment Corporation, Systems Research Center, Palo Alto, Ca, December 1985.

[33] 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.

[34] 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.

[35] Darrell D. E. Long, Bruce R. Montague, and Luis-Felipe Cabrera. Swift/RAID: a distributed RAID system. *Computing Systems*, **7**(3):333–359. Usenix, Summer 1994.

[36] Michael G. Luby, Michael Mitzenmacher, M. Amin Shokrollahi, and Daniel A. Spielman. Efficient Erasure Correcting Codes. *IEEE Transactions on Information Theory*, **47**(2):569–584. IEEE, February 2001.

[37] D. Malkhi, M. K. Reiter, D. Tulone, and E. Ziskind. Persistent objects in the Fleet system. *DARPA Information Survivability Conference and Exposition* (Anaheim, CA, 12–14 June 2001), pages 126–136. IEEE Computer Society, 2001.

[38] Dahlia Malkhi and Michael Reiter. Byzantine Quorum Systems. *Distributed Computing*, **11**(4):203–213. Springer–Verlag, 1998.

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

[40] David L. Mills. Improved algorithms for synchronizing computer network clocks. *IEEE-ACM Transactions on Networking*, **3**(3), June 1995.

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

[42] 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.

[43] 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), pages 31–44. USENIX Association, 2002.

[44] Brian D. Noble and M. Satyanarayanan. An empirical study of a highly available file system. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Nashville, TN, 16–20 May 1994). Published as *Performance Evaluation Review*, **22**(1):138–149. ACM, 1994.

[45] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). *ACM SIGMOD International Conference on Management of Data* (Chicago, IL), pages 109–116, 1–3 June 1988.

[46] Adam G. Pennington, John D. Strunk, John Linwood Griffin, Craig A. N. Soules, Garth R. Goodson, and Gregory R. Ganger. *Storage-based intrusion detection: Watching storage activity for suspicious behavior*. Technical report CMU–CS–02–179. Carnegie Mellon University, October 2002.

[47] 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.

[48] 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.

[49] 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.

[50] D. P. Reed and L. Svobodova. SWALLOW: a distributed data storage system for a local network. *International Workshop on Local Networks* (Zurich, Switzerland), August 1980.

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

[52] 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.

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

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

[55] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Greg Ganger. Metadata efficiency in versioning file systems. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–02 April 2003), pages 43–57. USENIX Association, 2003.

[56] John D. Strunk, Garth R. Goodson, Adam G. Pennington, Craig A. N. Soules, and Gregory R. Ganger. *Intrusion detection, diagnosis, and recovery with self-securing storage*. Technical report CMU–CS–02–140. Carnegie Mellon University, 2002.

[57] 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.

[58] 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.

[59] Jay J. Wylie, Mehmet Bakkaloglu, Vijay Pandurangan, Michael W. Bigrigg, Semih Oguz, Ken Tew, Cory Williams, Gregory R. Ganger, and Pradeep K. Khosla. *Selecting the right data distribution scheme for a survivable storage system*. Technical report CMU–CS–01–120. CMU, May 2001.

[60] 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.