Byzantine-tolerant erasure-coded storage

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

September 2003
CMU-CS-03-187

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This paper describes a decentralized consistency protocol for survivable storage that exploits data versioning within storage-nodes. Versioning enables the protocol to efficiently provide linearizability and wait-freedom of read and write operations to erasure-coded data in asynchronous environments with Byzantine failures of clients and servers. Exploiting versioning storage-nodes, the protocol shifts most work to clients. Reads occur in a single round-trip unless clients observe concurrency or write failures. Measurements of a storage system using this protocol show that the protocol scales well with the number of failures tolerated, and that it outperforms a highly-tuned instance of Byzantine-tolerant state machine replication.

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, Byzantine fault-tolerance, atomic registers, erasure codes

1 Introduction

Survivable storage systems spread data redundantly across a set of decentralized storage-nodes in an effort to ensure its availability despite the failure or compromise of storage-nodes. Such systems require some protocol to maintain data consistency and liveness in the presence of failures and concurrency.

This paper describes and evaluates a new consistency protocol that operates in an asynchronous environment and tolerates Byzantine failures of clients and storage-nodes. The protocol supports a hybrid failure model in which up to t storage-nodes may fail: $b \leq t$ of these failures can be Byzantine the remainder can crash. The protocol requires at least $2t+2b+1$ storage-nodes (i.e., $4b+1$ if $t=b$). The protocol supports m-of-n erasure codes (i.e., m-of-n fragments are needed to reconstruct the data), which usually require less network bandwidth (and storage space) than full replication [40, 41].

Briefly, the protocol works as follows. To perform a write, a client determines the current logical time and then writes time-stamped fragments to at least a threshold quorum 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 a threshold quorum of storage-nodes and determines whether they comprise a completed write; usually, they do. If they do not, additional and historical fragments are fetched, and repair may be performed, until a completed write is observed.

The protocol gains efficiency from three features. First, the space-efficiency of m-of-n erasure codes can be substantial. Second, most read operations complete in a single round trip: reads that observe write concurrency or failures (of storage-nodes or a client write) may incur in additional work. Most studies of distributed storage systems (e.g., [3, 26]) indicate that concurrency is uncommon (i.e., writer-writer and writer-reader sharing occurs in well under 1% of operations). Failures, although tolerated, ought to be rare. Moreover, a subsequent write or read (with repair) will replace a write that has not been completed, thus preventing future reads from incurring any additional cost; when writes do the fixing, additional costs are never incurred. Third, most protocol processing is performed by clients, increasing scalability via the well-known principle of shifting work from servers to clients [17].

This paper describes the protocol in detail and provides a proof sketch of its safety and liveness. It also describes and evaluates a prototype storage system called PASIS [41], that uses the protocol. Measurements of the prototype highlight the benefits of the three features. For example, client response times are only 20% higher when tolerating four Byzantine storage-node failures than when tolerating one. For comparison, a well-performing state machine replication implementation [5] has a higher single-fault response time for write operations and is $2\times$ slower when tolerating four faults. Additionally, the shifting of protocol processing to clients halves storage-node CPU load and thereby doubles throughput as the number of clients scales upwards.

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. To read D, Client B issues read requests to an overlapping subset of storage-nodes. This scheme provides

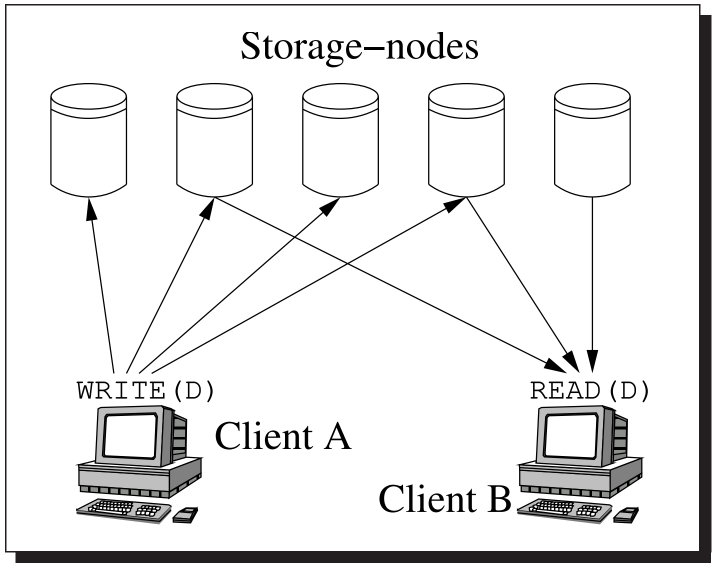

Figure 1: **High-level architecture for survivable storage.** Spreading data redundantly across storage-nodes improves its fault-tolerance. Clients write and (usually) read data from multiple storage-nodes.

access to data-items even when subsets of the storage-nodes have failed.

A common data distribution scheme used in such systems is replication, in which a writer stores a replica of the new data-item value at each storage-node to which it sends a write request. Alternately, more space-efficient erasure coding schemes can be used to reduce network load and storage consumption.

To provide reasonable semantics, the system must guarantee that readers see consistent data-item values. Specifically, the linearizability of operations is desirable for a shared storage system. Our protocol tolerates Byzantine faults of any number of clients and a limited number of storage nodes while implementing linearizable [16] and wait-free [14] read-write objects. Linearizability is adapted appropriately for Byzantine clients, and wait-freedom is as in the model of [18].

Most prior systems implementing Byzantine fault-tolerant services adopt the state machine approach [35], whereby all operations are processed by server replicas in the same order (*atomic broadcast*). While this approach supports a linearizable, Byzantine fault-tolerant implementation of *any* deterministic object, such an approach cannot be wait-free [12, 14, 18]. Instead, such systems achieve liveness only under stronger timing assumptions, such as synchrony (e.g., [8, 29, 37]) or partial synchrony [11] (e.g., [6, 19, 33]), or probabilistically (e.g., [4]). An alternative is Byzantine quorum systems [22], from which our protocols inherit techniques. Protocols for supporting a linearizable implementation of any deterministic object using Byzantine quorums have been developed (e.g., [23]), but also necessarily forsake wait-freedom to do so.

Byzantine fault-tolerant protocols for implementing read-write objects using quorums are described in [15, 22, 24, 28]. Of these, only Martin et al. [24] achieve linearizability in our fault model, and this work is also closest to ours in that it uses a type of versioning. In our protocol, a reader may retrieve fragments for several versions of the data-item in the course of identifying the return value of a read. Similarly, readers in [24] “listen” for updates (versions) from storage-nodes until a complete write is observed. Conceptually, our approach differs by clients reading past versions, versus listening for future versions broadcast by servers. In our fault model, especially in consideration of faulty clients, our protocol has several advantages. First, our protocol works for erasure-coded data, whereas extending [24] to erasure coded data appears nontrivial. Second, ours provides better message efficiency: [24] involves a $\Theta(N^2)$ message exchange among the N servers per write (versus no server-to-server exchange in our case) over and above otherwise comparable (and linear in N) message costs. Third, ours requires less computation, in that [24] requires digital signatures by clients, which in practice is two orders of magnitude more costly than the cryptographic transforms we employ. Advantages of [24] are that it tolerates a higher fraction of faulty servers than our protocol, and does not require servers to store a potentially unbounded number of data-item versions. Our prior analysis of versioning storage, however, suggests that the latter is a non-issue in practice [39], and even under attack this can be managed using a garbage collection mechanism we describe in Section 6.

We contrast our use of versioning to maintain consistency with systems in which each write creates a new, immutable version of a data-item to which subsequent reads are directed (e.g., [25, 31, 32]). Such systems merely shift the consistency and liveness problems to the metadata mechanism that resolves data-item names to a version. So, systems that employ such an approach (e.g., Past [34], CFS [9], Farsite [2], and the archival portion of OceanStore [21]) require a separate protocol mechanism to manage this metadata.

A goal of our work is to demonstrate that our protocol is sufficiently efficient to use in practice. Castro and Liskov [6] have made available for experimentation a well-engineered implementation of a Byzantine fault-tolerant replicated state machine. Their BFT library is used elsewhere in the research community (e.g., Farsite [2]). As such, we use their implementation as a point of comparison to demonstrate that our protocol is efficient and scalable in both network and storage-node CPU utilization.

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.

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 *fails*. 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. We assume that Byzantine storage-nodes can collude with each other and with any Byzantine clients. A client or storage-node that does not exhibit a Byzantine failure (it is either correct or crashes) is *benign*.

The protocol tolerates crash and Byzantine clients. As in any practical storage system, an authorized Byzantine client can write arbitrary values to storage, which affects the value of the data, but not its consistency. We assume that Byzantine clients and storage-nodes are computationally bounded so that we can employ cryptographic primitives.

We assume an asynchronous model of time (i.e., we make no assumptions about message transmission delays or the execution rates of clients and storage-nodes). We assume that communication between a client and a storage-node 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 to it.

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. Clients may encode data-items in an erasure-tolerant manner; thus the distinction between data-item and data-fragment. Requests are *executed* by storage-nodes; a correct storage-node that executes a write request *hosts* that write operation.

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. There is a well known zero time, $\mathbf{0}$, and null value, $\perp$, which storage-nodes can return in response to read requests. Implicitly, all stored data is initialized to $\perp$ at time $\mathbf{0}$.

4 Protocol

This section describes our Byzantine fault-tolerant consistency protocol that efficiently supports erasure-coded data-items by taking advantage of versioning storage-nodes. It presents the mechanisms employed to encode and decode data, and to protect data integrity from Byzantine storage-nodes and clients. It then describes, in detail, the protocol in pseudo-code form. Finally, it develops constraints on protocol parameters to ensure the safety and liveness of the protocol.

4.1 Overview

At a high level, the protocol proceeds as follows. Logical timestamps are used to totally order all write operations and to identify data-fragments pertaining to the same write operation across the set of storage-nodes. For each write, a logical timestamp is constructed by the client that is guaranteed to be unique and greater than that of the *latest complete write* (the complete write with the highest timestamp). This is accomplished by querying storage-nodes for the greatest timestamp they host, and then incrementing the greatest response. In order to verify the integrity of the data, a hash that can verify data-fragment correctness is appended to the logical timestamp.

To perform a read operation, clients issue read requests to a subset of storage-nodes. Once at least a read quorum of storage-nodes reply, the client identifies the *candidate*—the response with the greatest logical timestamp. The set of read responses that share the timestamp of the candidate comprise the *candidate set*. The read operation *classifies* the candidate as *complete*, *repairable*, or *incomplete*. If the candidate is classified as complete, the data-fragments, timestamp, and return value are validated. If validation is successful, the value of the candidate is returned and the read operation is complete; otherwise, the candidate is reclassified as incomplete. If the candidate is classified as repairable it is repaired by writing data-fragments back to the original set of storage-nodes. Prior to performing repair, data-fragments are validated in the same manner as for a complete candidate. If the candidate is classified as incomplete, the candidate is discarded, previous data-fragment versions are requested, and classification begins anew. All candidates fall into one of the three classifications, even those corresponding to concurrent or failed write operations.

4.2 Mechanisms

Several mechanisms are used in our protocol to achieve linearizability in the presence of Byzantine faults.

4.2.1 Erasure codes

In an erasure coding scheme, N data-fragments are generated during a write (one for each storage-node), and any m of those data-fragments can be used to decode the original data-item. Any m of the data-fragments can deterministically generate the other $N-m$ data-fragments. We use a systematic information dispersal algorithm [30], which stripes the data-item across the first m data-fragments and generates erasure-coded data-fragments for the remainder. Other threshold erasure codes (e.g., Secret Sharing [36] and Short Secret Sharing [20]) work as well.

4.2.2 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: Cross checksums [13] are used to detect corrupt data-fragments. A cryptographic hash of each data-fragment is computed. 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 (i.e., it is replicated N times). Cross checksums enable read operations to detect data-fragments that have been modified by storage-nodes.

4.2.3 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 $\binom{N}{m}$ permutations of data-fragments could "recover" a distinct data-item. Additionally, a Byzantine client could partition the set of N data-fragments into subsets that each decode to a distinct data-item. These attacks are similar to *poisonous writes* for replication as described by Martin et al. [24]. To protect against Byzantine clients, the protocol must ensure that read operations 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, storage-node verification, and validated cross checksums.
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. Note, the hash is used for space-efficiency; instead, the entire cross checksum could be placed in the low bits of the timestamp.
STORAGE-NODE VERIFICATION: On a write, each storage-node verifies its data-fragment against its hash in the cross checksum. The storage-node also verifies the cross checksum against the hash in the 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. It follows, that only Byzantine storage-nodes can return data-fragments that do not verify against the cross checksum.

VALIDATED CROSS CHECKSUMS: Storage-node verification combined with a validating timestamp ensures that the data-fragments being considered by a read operation were written by the client (as opposed to being 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. To validate the cross checksum, all N data-fragments are required. Given any m data-fragments, the full set of N data-fragments a correct client should have written can be generated. The "correct" cross checksum can then be computed from the regenerated set of data-fragments. If the generated cross checksum does not match the verified cross checksum, then a Byzantine client performed the write operation. Only a single-valued write operation can generate a cross checksum that verifies against the validating timestamp.

4.3 Pseudo-code

The pseudo-code for the protocol is shown in Figures 2 and 3. The symbol Q_W defines a complete write threshold—the number of write responses a client must observe to know that the write operation is complete. Thus, a write is *complete* once $Q_W - b$ benign storage-nodes have executed write requests with timestamp ts. Note that Q_W is a *threshold* quorum; as such, it represents a scalar value, not a set. 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 set of N storage-nodes. In the pseudo-code, the operator '|' denotes string concatenation.

4.3.1 Storage-node interface

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. Given the simplicity of the storage-node interface, storage-node pseudo-code has been omitted.

4.3.2 Write operation

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 must be determined. Collecting more than $N + b - Q_W$ responses, on line 8 of `WRITE`, is necessary to guarantee that the response set overlaps a complete write at a minimum of $b + 1$ storage-nodes. Thus, the constraint ensures that the timestamp of the latest complete write is observed.

Next, the `ENCODE` function, on line 10, encodes the data-item into N data-fragments. The data-fragments are used to construct a cross checksum from the concatenation of the hash of each data-fragment (line 11). The function `MAKE_TIMESTAMP`, called on line 12, generates a logical timestamp to be used for the current write operation. This is done by incrementing the high order bits of the greatest observed logical timestamp from the *ResponseSet* (i.e., $LT.TIME$), appending the client's ID, and appending the *Verifier*. The *Verifier* is just the hash of the cross checksum.

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

```
WRITE(Data) :
 1: /* Determine greatest existing logical timestamp */
 2: for all S_i ∈ {S_1, ..., S_N} do
 3:     SEND(S_i, TIME_REQUEST)
 4: end for
 5: ResponseSet := ∅
 6: repeat
 7:     ResponseSet := ResponseSet ∪ RECEIVE(S, TIME_RESPONSE)
 8: until (|ResponseSet| > N + b − Q_W)
 9: /* Generate data-fragments, cross checksum and logical timestamp */
10: {D_1, ..., D_N} := ENCODE(Data)
11: CC := MAKE_CROSS_CHECKSUM({D_1, ..., D_N})
12: LT := MAKE_TIMESTAMP(MAX[ResponseSet.LT], CC)
13: /* Write out the data-fragments */
14: DO_WRITE({D_1, ..., D_N}, LT, CC)

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(LT_max, CC) :
 1: LT.TIME := LT_max.TIME + 1
 2: LT.ID := ID
 3: LT.Verifier := HASH(CC)
 4: 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| = Q_W)
```

Figure 2: **Write operation pseudo-code.**

checksum with the verifier 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 at least Q_W `WRITE_RESPONSE` messages are received (line 7 of `DO_WRITE`).

4.3.3 Read operation

The read operation iteratively identifies and classifies candidates, until a repairable or complete candidate is found. Once a repairable or complete candidate is found, the read operation validates its correctness and returns the data. A client must observe at least Q_W data-fragments with matching logical timestamps in order to classify a candidate as complete. To ensure that a candidate that corresponds to a complete write is classified as such as often as possible, $N - t$ data-fragments are considered for classification (line 10 of `READ`). Note that the read operation returns a $\langle timestamp, value \rangle$ pair; in practice, a client only makes use of the value returned.

The read operation begins by issuing `READ_LATEST_REQUEST` commands to all storage-nodes (via the `DO_READ` function). Each storage-node responds with the data-fragment, logical timestamp,

```
READ() :
 1: ResponseSet := DO_READ(READ_LATEST_REQUEST, ⊥)
 2: loop
 3:   ⟨CandidateSet, LT_candidate⟩ := CHOOSE_CANDIDATE(ResponseSet)
 4:   if (|CandidateSet| ≥ Q_W − t − b) then
 5:     /* Complete or repairable write found */
 6:     {D_1, ..., D_N} := GENERATE_FRAGMENTS(CandidateSet)
 7:     CC_valid := MAKE_CROSS_CHECKSUM({D_1, ..., D_N})
 8:     if (CC_valid = CandidateSet.CC) then
 9:       /* Validated cross checksums match */
10:       if (|CandidateSet| < Q_W) then
11:         /* Repair is necessary */
12:         DO_WRITE({D_1, ..., D_N}, LT_candidate, CC_valid)
13:       end if
14:       Data := DECODE({D_1, ..., D_N})
15:       RETURN(⟨LT_candidate, Data⟩)
16:     end if
17:   end if
18:   /* Partial or data-item validation failed, loop again */
19:   ResponseSet := DO_READ(READ_PREVIOUS_REQUEST, LT_candidate)
20: end loop

DO_READ(READ_COMMAND, LT) :
 1: for all S_i ∈ {S_1, ..., S_N} do
 2:   SEND(S_i, READ_COMMAND, LT)
 3: end for
 4: ResponseSet := ∅
 5: repeat
 6:   Resp := RECEIVE(S, READ_RESPONSE)
 7:   if (VALIDATE(Resp.D, Resp.CC, Resp.LT, S) = TRUE) then
 8:     ResponseSet := ResponseSet ∪ Resp
 9:   end if
10: until (|ResponseSet| = N − t)
11: RETURN(ResponseSet)

VALIDATE(D, CC, LT, S) :
 1: if ((HASH(CC) ≠ LT.Verifier) OR (HASH(D) ≠ CC[S])) then
 2:   RETURN(FALSE)
 3: end if
 4: RETURN(TRUE)
```

Figure 3: **Read operation pseudo-code.**

and cross checksum corresponding to the greatest timestamp it has executed.

The integrity of each response is individually validated through the VALIDATE function, called on line 7 of DO_READ. This function checks the cross checksum against the *Verifier* found in the logical timestamp and the data-fragment against the appropriate hash in the cross checksum.

Since, in an asynchronous system, slow storage-nodes cannot be differentiated from crashed storage-nodes, only $N - t$ read responses can be collected (line 10 of DO_READ). Since correct storage-nodes perform the same validation before executing write requests, the only responses that can fail the client's validation are those from Byzantine storage-nodes. For every discarded Byzantine storage-node response, an additional response can be awaited.

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 found 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 validated data-fragments that share a common cross checksum and logical timestamp.

Once a candidate has been chosen, it is classified as either complete, repairable, or incomplete. Given that only $N-t$ read responses can be collected, a completed write at Q_{W} storage-nodes, b of which may be Byzantine (and capable of "hiding" the write), may have as few as $Q_{\text{W}}-t-b$ data-fragments visible to a later read. Accounting for this lack of information, the classification rules are:

- $|CandidateSet| \geq Q_{\text{W}}$, the candidate is classified as complete;
- $Q_{\text{W}}-t-b \leq |CandidateSet| < Q_{\text{W}}$, the candidate is classified as repairable;
- $|CandidateSet| < Q_{\text{W}}-t-b$, the candidate is classified as incomplete;

If the candidate is classified as incomplete, a `READ_PREVIOUS_REQUEST` 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 either complete or repairable, the candidate set contains sufficient data-fragments written by the client to decode the original data-item. To validate the observed write's integrity, the candidate set is used to generate a new set of data-fragments (line 6 of `READ`). A validated cross checksum, CC_{valid}, is computed from the newly generated data-fragments. The validated cross checksum is compared to the cross checksum of the candidate set (line 8 of `READ`). 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 by a correct client 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, repair is performed: write requests are issued with the generated data-fragments, the validated cross checksum, and the logical timestamp (line 10 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`, on line 14 of `READ`, decodes m data-fragments, returning the data-item.

It should be noted that, even after a write completes, it may be classified as repairable by a subsequent read, but it will never be classified as incomplete. For example, this will occur if the read set (of $N-t$ storage-nodes) does not fully encompass the write set (of Q_{W} storage-nodes).

4.4 Protocol constraints

To ensure that linearizability and liveness are achieved, Q_{W} and N must be constrained with regard to b, t, and each other. As well, the parameter m, used in `DECODE`, must be constrained. We sketch the proof that the protocol under these constraints is safe and live in Appendix I.

WRITE TERMINATION: To ensure write operations are able to complete in an asynchronous environment,

$$Q_{\text{W}} \leq N-t. \tag{1}$$

Since slow storage-nodes cannot be differentiated from crashed storage-nodes, only $N-t$ responses can be awaited.

OVERLAP: To ensure linearizability, the function `CHOOSE_CANDIDATE` must select the latest complete write as the candidate. For this to occur, the response set of a read operation (or a get logical time operation) must intersect with at least one correct (non-Byzantine) storage-node that executed a write request of a complete write. Let Q_{R} be the size of the read response set,

$$\begin{aligned} N+b &< Q_{\mathrm{W}}+Q_{\mathrm{R}}, \\ N+b-Q_{\mathrm{W}} &< Q_{\mathrm{R}}. \end{aligned} \tag{2}$$

This is the source of the constraint on line 8 of `WRITE`.

REAL REPAIRABLE CANDIDATES: If Byzantine storage-nodes, through collusion, can fabricate a candidate that a client deems repairable, then the storage-nodes can "trick" a client into repairing a value that was never entered into the system by an authorized client. To ensure that Byzantine storage-nodes can never fabricate a repairable candidate, a candidate set of size b must be classifiable as incomplete. Since a candidate is repairable if $Q_{\mathrm{W}}-t-b$ or more responses are observed (recall the classification rules), then

$$\begin{aligned} b &< Q_{\mathrm{W}}-t-b, \\ t+2b &< Q_{\mathrm{W}}. \end{aligned} \tag{3}$$

CONSTRAINT SUMMARY: Constraints (1) and (3) yield overall constraints on Q_{W} and N:

$$\begin{aligned} t+2b+1 \le Q_{\mathrm{W}} &\le N-t, \\ 2t+2b+1 &\le N. \end{aligned}$$

REPAIRABLE CANDIDATES: For repair to be possible, a repairable candidate must be decodable. The fewest responses a repairable candidate can have is $Q_{\mathrm{W}}-t-b$, therefore,

$$1 \le m \le Q_{\mathrm{W}}-t-b.$$

So, m can always be at least $b+1$ (note: larger values of m yield more space-efficient erasure coding). If a repairable candidate is decodable, then a complete candidate is clearly also decodable.

5 Evaluation

This section evaluates the consistency protocol's performance in the context of a prototype storage system called PASIS [41]. We also compare the PASIS implementation of our protocol with the BFT library implementation [7], an efficient implementation of the BFT protocol for replicated state machines [5].

5.1 Prototype implementation

PASIS consists of clients and storage-nodes. Storage-nodes store data-fragments and their versions. Clients execute the protocol to read and write data-items.

5.1.1 Storage-node implementation

Our storage-nodes use the Comprehensive Versioning File System (CVFS) [38] to retain data-fragments and their versions. CVFS uses a log-structured data organization to reduce the cost of data versioning. Experience indicates that retaining every version and performing local garbage collection comes with minimal performance cost (a few percent) and that it is feasible to retain complete version histories for several days [38, 39].

We extended CVFS to provide an interface for retrieving the logical timestamp of a data-fragment. Each write request contains a data-fragment, a logical timestamp, and a cross checksum. In a normal read response, storage-nodes return all three. To improve performance, read responses contain a limited version history containing logical timestamps of previously executed write requests. The version history allows clients to identify and classify additional candidates without issuing extra read requests. The storage-node can also return read responses that contain no data other than version histories (this makes candidate classification more network efficient).

5.1.2 Client implementation

The client module provides a block-level interface to higher level software, and uses a simple RPC interface to communicate with storage-nodes. The RPC mechanism uses TCP/IP. The client module is responsible for the execution of the consistency protocol and for encoding and decoding data-items.

Initially, read requests are issued to the first Q_W storage-nodes. Only m of these request the data-fragment, while all request version histories; this makes the read operation more network efficient. If the read responses do not yield a candidate that is classified as complete, read requests are issued to the remaining storage-nodes (and a total of up to $N-t$ responses are awaited). If the initial candidate is classified as incomplete, subsequent rounds of read requests fetch only version histories until a candidate is classified as either repairable or complete. If necessary, after classification, extra data-fragments are fetched according to the candidate timestamp. Once the data-item is successfully validated and decoded, it is returned to the client.

5.1.3 Mechanism implementation

We measure the space-efficiency of an erasure code in terms of *blowup*—the amount of data stored over the size of the data-item. We use an information dispersal algorithm [30] which has a blowup of $\frac{N}{m}$. Our information dispersal implementation stripes the data-item across the first m data-fragments (i.e., each data-fragment is $\frac{1}{m}$ of the original data-item's size). These *stripe-fragments* are used to generate the *code-fragments* via polynomial interpolation within a Galois Field, which treats the stripe-fragments and code-fragments as points on some $m-1$ degree polynomial. Our implementation of polynomial interpolation was originally based on publicly available code for information dispersal [10]. We modified the source to make use of stripe-fragments and added an implementation of Galois Fields of size 2^8 that use a lookup table for multiplication.

Our implementation of cross checksums closely follows Gong [13]. We use a publicly available implementation of MD5 for all hashes [1]. Each MD5 hash is 16 bytes long; thus, each cross checksum is $N \times 16$ bytes long.

5.2 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.

In all experiments, clients keep a fixed number of read and write operations outstanding. Once an operation completes, a new operation is issued (there is no think time). Unless otherwise specified, requests are for random 16 KB blocks. For all experiments, the working set fits into memory and all caches are warmed up beforehand.

Most experiments focus on configurations where $b = t$ and $t = \{1,2,3,4\}$. Thus, for our protocol, $N = 4b+1$, $Q_{\mathrm{W}} = 3b+1$, and $m = b+1$. For BFT, $N = 3b+1$ (i.e., $N = \{4,7,10,13\}$).

5.2.1 PASIS configuration

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 experiments on the overheads introduced by the protocol and not those introduced by the disk subsystem. No authentication is currently performed in the PASIS prototype (authentication is expected to have little performance impact).

5.2.2 BFT configuration

Operations in BFT [5] require agreement among the replicas (storage-nodes in PASIS). Agreement is performed in four steps: (i) the client broadcasts requests to all replicas; (ii) the *primary* broadcasts pre-prepare messages to all replicas; (iii) all replicas broadcast prepare messages to all replicas; and, (iv) all replicas send replies back to the client and then broadcast commit messages to all other replicas. Commit messages are piggy-backed on the next pre-prepare or prepare message to reduce the number of messages on the network. *Authenticators*, chains of MACs, are used to ensure that broadcast messages from clients and replicas cannot be modified by a Byzantine replica. All clients and replicas have public and private keys that enables them to exchange symmetric cryptography keys used to create MACs. Logs of commit messages are checkpointed (garbage collected) periodically. View changes, in which a new primary is selected, are performed periodically.

A fast path for read operations is implemented in BFT. The client broadcasts its request to all replicas. Each replica replies once all messages previous to the request are committed. Only one replica sends the full reply (i.e., the data and digest), and the remainder just send digests that can verify the correctness of the data returned. If the replies from replicas do not agree, the client re-issues the read operation—for the replies to agree, the read-only request must arrive at $2b+1$ of the replicas in the same order (with regard to other write operations). The re-issued read operation performs agreement before a single replica replies.

The BFT configuration does not store data to stable storage, instead it stores all data in memory and accesses it via memory offsets. For the experiments, checkpointing and view changes are suppressed. BFT uses UDP connections rather than TCP. The BFT implementation defaults to using IP multicast; however, IP multicast was disabled for all experiments. The BFT implementation authenticates broadcast messages via authenticators, and point to point messages with MACs.

	b=1	b=2	b=3	b=4
Erasure coding (ms)	1.25	1.50	1.73	1.99
Cross checksum (ms)	0.36	0.44	0.48	0.51
Verifier (μs)	1.64	2.31	3.61	4.28
Validate (μs)	81.5	58.0	48.3	40.1

Table 1: Computation costs in PASIS (1 GHz CPU)

5.3 Performance and scalability

5.3.1 Mechanism costs

Client and storage-node computation costs in PASIS are listed in Table 1. For every read and write operation, clients perform erasure coding (i.e., they compute $N-m$ data-fragments given m data-fragments), generate a cross checksum, and generate a verifier. Recall that writes generate the first m data-fragments by striping the data-item into m fragments. Similarly, reads must generate $N-m$ fragments, from the m they have, in order to verify the cross checksum. The cost of erasure encoding, cross checksumming, and creating the verifier grow with m and N.

Storage-nodes validate each write request they receive. This validation requires a comparison of the data-fragment's hash to the hash within the cross checksum, and a comparison of the cross checksum's hash to the verifier within the timestamp. The cost of storage-node validation decreases with data-fragment sizes ($\frac{1}{m}$); as such, it decreases with t.

5.3.2 Response time

Figure 4 shows the mean response time of a single request from a single client as a function of tolerated number of storage-node failures. The focus in this plot is the slopes of the response time lines: the flatter the line the more scalable the protocol is with regard to the number of faults tolerated. A key contributor to response time is network cost, which is dictated by both the number of nodes and the space-efficiency of the encoding. PASIS has better response times than BFT for write operations due to the space-efficiency of erasure codes and the nominal amount of work storage-nodes perform to execute write requests.

PASIS has longer response times than BFT for read operations. This can be attributed to three main factors: First, for our protocol, the client computation cost grows as the number of failures tolerated increases because the cost of generating data-fragments grows as m increases. Second, the PASIS storage-nodes store data in a real file system; although the data is serviced from cache, the expenses to fetch such data are larger than a single memory access. The BFT setup used does not incur these costs, but would in an apples-to-apples comparison. Third, BFT's implementation of "fast path" read operations scale extremely well in terms of failures tolerated. Recall, one server returns a replica plus digest, the remainder just return a digest, and no agreement is performed; the operation completes in a single round-trip with little computation cost.

In addition to the $b=t$ case, Figure 4 shows one instance of PASIS assuming a hybrid fault model with $b=1$. For space-efficiency, we set $m=t+1$, which is $t-b$ greater than the minimum allowed value of m. Consequently, Q_W and N must also be set above their minimum values. The $b=1$ line thus corresponds to a configuration with $b=1$, $m=t+1$, $N=3t+2$ and $Q_W=2t+2$.

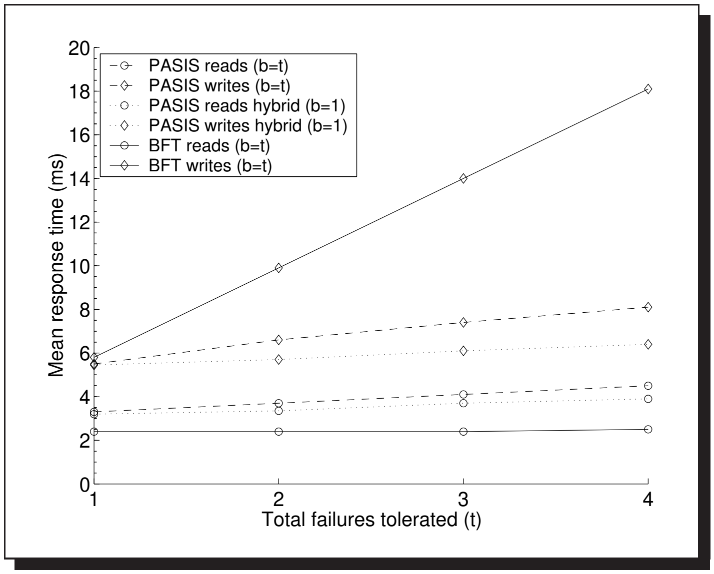

Figure 4: **Mean response time vs. total failures tolerated.** Mean response times of read and write operations of random 16 KB blocks in PASIS and BFT. For PASIS, lines corresponding to both b=1 and b=t are shown.

At $t = 1$, this configuration is identical to the Byzantine-only configuration. As t increases, this configuration is more space-efficient than the Byzantine-only configuration, since it requires $t - 1$ fewer storage-nodes. As such, both read and write operations scale better.

Some read operations in PASIS can require repair. A repair operation must perform a "write" operation to repair the value before it is returned by the read. Interestingly, the response time of a read that performs repair is less than the sum of the response times of a normal read and a write operation. This is because the "write" operation during repair does not need to read logical timestamps before issuing write requests. Additionally, data-fragments need only be written to storage-nodes that do not already host the write operation.

5.3.3 Scalability

Figure 5 breaks mean response times of read and write operations into the costs at the client, on the network, and at the storage-node for $b = 1$ and $b = 4$. Since measurements are taken at the user-level, kernel-level timings for host network protocol processing (including network system calls) are attributed to the "network" cost of the breakdowns. To understand the scalability of these protocols, it is important to understand these breakdowns.

Although write operations for both protocols have similar response times for $b = 1$, the response times of BFT write operations scale poorly. The large network cost for BFT writes is due to the space-inefficiency of replication. For $b = 4$, BFT has a blowup of $13\times$ on the network, whereas our protocol has a blowup of $\frac{17}{5} = 3.4\times$ on the network. Note that BFT can use IP multicast to mitigate this effect on write operation response time, though the aggregate network bandwidth utilization would still grow.

Regardless of whether or not IP multicast is employed, our protocol in PASIS requires much less computation on storage-nodes than BFT requires on replicas. Server cost is broken down to

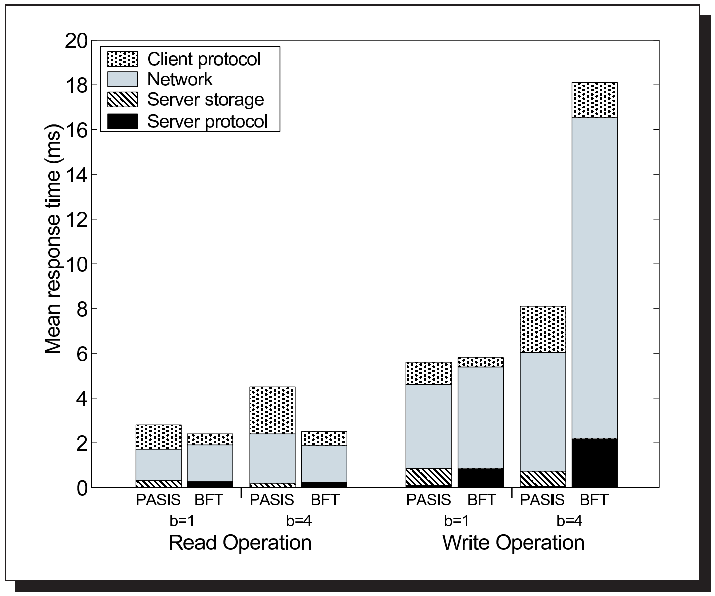

Figure 5: **Protocol cost breakdown.** The bars illustrate the cost breakdown of read and write operations for PASIS and BFT for $b = 1$ and $b = 4$. Each bar corresponds to a single point on the mean response time graph in Figure 4

distinguish protocol costs from the cost of server storage (i.e., stable data storage). Figure 5 shows that as b increases, protocol related server computation on write operations grows significantly for BFT and barely changes for PASIS. In PASIS, the server protocol cost decreases from 90 μs for $b = 1$ to 57 μs for $b = 4$, whereas in BFT it increases from 0.80 ms to 2.1 ms. The server cost in PASIS decreases because m increases as b increases, reducing the size of the data-fragment that is validated. Since the BFT library keeps all data in memory and accesses blocks via memory offsets, it incurs almost no server storage costs. We expect that a BFT implementation with stable data storage would incur server storage costs similar to PASIS (e.g., around 0.7 ms for a write operation, as is shown for $b = 1$ in Figure 5).

By offloading work from the storage-nodes to the client, PASIS greatly increases its scalability in terms of supported client load. Unfortunately, a direct throughput comparison to our BFT setup was impossible due to its poor stability under heavy load. We have observed that the throughput in PASIS scales according to network and server CPU utilization.

Under heavy load, the response time of read operations grows in PASIS due to resource contention. The same is true of BFT. Moreover, we believe that under load, the "fast path" for read operations in BFT becomes less effective: Replicas delay read replies until all previous requests have committed, and the likelihood that read requests are similarly ordered at distinct replicas diminishes.

5.3.4 Concurrency

In PASIS, both read-write concurrency and client crashes during write operations can lead to client read operations observing repairable writes. 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 with four operations outstanding. Each client accesses a range of eight data blocks, with 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 bandwidth nor significant increases in mean response time. Even at this high concurrency level, the initial candidate was classified as complete 88.8% of the time, and that repair was necessary only 3.3% of the time. Since repair occurs so seldom, the effect on response time and throughput is minimal.

6 Discussion

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 nature of the protocol, cannot determine which local data-fragment versions are safe to garbage-collect. An individual storage-node can garbage collect a data-fragment version if there exists a later complete write. Storage-nodes are able to classify writes by executing the consistency protocol in the same manner as the client.

BYZANTINE CLIENTS: In a storage system, Byzantine clients can write arbitrary values. The use of fine-grained versioning (i.e., self-securing storage [39]), facilitates detection, recovery, and diagnosis from storage intrusions. Arbitrarily modified data can be rolled back to its pre-corruption state.

Byzantine clients can also attempt to exhaust the resources available to the PASIS protocol. Issuing an inordinate number of write operations could exhaust storage space. However, continuous garbage collection frees storage space prior to the latest complete write. If a Byzantine client were to intentionally issue incomplete write operations, then garbage collection may not be able to free up space. In addition, incomplete writes require read operations to roll-back behind them, thus consuming client computation and network resources. In practice, relying on storage-based intrusion detection [27] is probably sufficient to detect clients that exhibit such behavior.

7 Summary

Storage-node versioning enables an efficient protocol that provides strong consistency and liveness in the face of Byzantine failures and concurrency. The protocol achieves scalability by offloading work to the client and using space-efficient erasure coding. Measurements of the PASIS prototype demonstrate these benefits.

Acknowledgements

We thank Craig Soules for his timely support of the versioned storage-node code base (the “S4/CVFS” code). We thank Carla Geisser and Terrence Wong for their help building the experimental infrastructure and getting BFT to run in our environment. We thank Miguel Castro for making the implementation of BFT publicly available. 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] *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] 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.

[4] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. *Advances in Cryptology - CRYPTO* (Santa Barbara, CA, 19–23 August 2001), pages 524–541. IEEE, 2001.

[5] 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.

[6] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, **20**(4):398–461. IEEE, November 2002.

[7] Miguel Castro and Rodrigo Rodrigues. *BFT library implementation*. http://www.pmg.lcs.mit.edu/bft/#sw.

[8] 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.

[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] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the Association for Computing Machinery*, **35**(2):288–323. IEEE, April 1988.

[12] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, **32**(2):374–382. ACM Press, April 1985.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] Kim P. Kihlstrom, Louise E. Moser, and P. Michael Melliar-Smith. The SecureRing group communication system. *ACM Transactions on Information and Systems Security*, **1**(4):371–406. IEEE, November 2001.

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

[21] 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.

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

[23] 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.

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

[25] 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.

[26] 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.

[27] 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. *USENIX Security Symposium* (Washington, DC, 06–08 August 2003), 2003.

[28] 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.

[29] Frank M. Pittelli and Hector Garcia-Molina. Reliable scheduling in a TMR database system. *ACM Transactions on Computer Systems*, **7**(1):25–60, February 1989.

[30] 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.

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

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

[33] Michael K. Reiter and Kenneth P. Birman. How to securely replicate services. *ACM Transactions on Programming Languages and Systems*, **16**(3):986–1009, May 1994.

[34] 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.

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

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

[37] Santosh K. Shrivastava, Paul D. Ezhilchelvan, Neil A. Speirs, and Alan Tully. Principal features of the voltan family of reliable node architectures for distributed systems. *IEEE Transactions on Computr*, **41**(5):542–549. IEEE, May 1992.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

8 Appendix I: Proofs

8.1 Proof of safety

This section sketches a proof that our protocol implements linearizability [16] as adapted appropriately for a fault model admitting operations by Byzantine clients. Intuitively, linearizability requires that each read operation return a value consistent with some execution in which each read and write is performed at a distinct point in time between when the client invokes the operation and when the operation returns. The adaptations necessary to reasonably interpret linearizability in our context arise from the fact that Byzantine clients need not follow the read and write protocols. The first adaptation is necessary because return values of reads by Byzantine clients obviously need not comply with any correctness criteria. As such, we disregard read operations by Byzantine clients in reasoning about linearizability, and define the duration of reads only for those executed by benign clients only.

DEFINITION 1 A read operation executed by a benign client *begins* when the client invokes `READ` locally, and *completes* when this invocation returns $\langle timestamp, value \rangle$.

The second needed adaptation of linearizability arises from the fact that it is not well defined when a write operation by a Byzantine client begins. Therefore, we settle for merely a definition of when writes by Byzantine operations complete.

DEFINITION 2 Storage-node S, *accepts* a write request with data-fragment D, cross checksum CC, and timestamp ts upon successful return of the function $\texttt{VALIDATE}(D, CC, ts, S)$ at the storage-node.

DEFINITION 3 A write operation with timestamp ts *completes* once $Q_W - b$ benign storage-nodes have accepted write requests with timestamp ts.

In fact, Definition 3 applies to write operations by benign clients as well as "write operations" by Byzantine clients. In this section, we use the label w_{ts} as a shorthand for the write operation with timestamp ts. In contrast to Definition 3, Definition 4 applies only to write operations by benign clients.

DEFINITION 4 w_{ts} *begins* when a benign client invokes the `WRITE` operation locally that issues a write request bearing timestamp ts.

LEMMA 1 *Let c_1 and c_2 be benign clients. If c_1 performs a read operation that returns $\langle ts_1, v_1 \rangle$, c_2 performs a read operation that returns $\langle ts_2, v_2 \rangle$, and $ts_1 = ts_2$, then $v_1 = v_2$.*

Proof sketch: Since $ts_1 = ts_2$, each read operation considers the same verifier. Since each read operation considers the same verifier, each read operation considers the same cross checksum. A read operation does not return a value unless the cross checksum is valid and there are more than b read responses with the timestamp (since only candidates classified as repairable or complete are considered). Thus, only a set of data-fragments resulting from the erasure-coding of the same data-item that are issued as write requests with the same timestamp can validate a cross checksum. As such, v_1 and v_2 must be the same. □

Let v_{ts} denote the value written by w_{ts} which, by Lemma 1, is well-defined. We use r_{ts} to denote a read operation by a benign client that returns $\langle ts, v_{ts} \rangle$.

DEFINITION 5 Let o_1 denote an operation that completes (a read by a benign client, or a write), and let o_2 denote an operation that begins (a read or write by a benign client). o_1 *precedes* o_2 if o_1 completes before o_2 begins. The precedence relation is written as $o_1 \rightarrow o_2$.

Operation o_2 is said to follow, or to be subsequent to, operation o_1. The notation $o_1 \not\rightarrow o_2$ is used to mean operation o_1 does not precede operation o_2.

LEMMA 2 *If $w_{ts'}$ is a write operation by a benign client and if $w_{ts} \rightarrow w_{ts'}$, then $ts < ts'$.*

Proof sketch: Constraint (2), the overlap constraint, ensures that the first phase of `WRITE` achieves a higher timestamp than any preceding write operation. A complete write operation executes at at least $Q_W - b$ benign storage-nodes. Considering Q_R `TIME_RESPONSE` messages ensures at least one such response is from a correct storage-node that executed the preceding write operation (line 8 of `WRITE`). A Byzantine storage-node can return a logical timestamp greater than that of the preceding write operation; however, this still advances logical time and Lemma 2 holds. □

OBSERVATION 1 Timestamp order is a total order on write operations. The timestamps of write operations by benign clients respect the precedence order among writes.

LEMMA 3 *If some read operation by a benign client returns $\langle ts, v_{ts} \rangle$, and if $w_{ts} \rightarrow r_{ts'}$, then $ts \leq ts'$.*

Proof sketch: By Definition 3, since w_{ts} completes, there are $Q_W - b$ benign storage-nodes that accept write-requests with timestamp ts. Storage-node crashes and the asynchronous environment can "hide" up to t of the $Q_W - b$ accepted write requests from $r_{ts'}$. As such, at least $Q_W - t - b$ responses with timestamp ts are observable by $r_{ts'}$; a read operation that observes a candidate with at least $Q_W - t - b$ responses performs repair (line 10 of `READ`). Since r_{ts} returns $\langle ts, v_{ts} \rangle$, v_{ts} can be returned from a read operation performed by a benign client. Thus, $r_{ts'}$ either repairs v_{ts}, observes v_{ts} as complete, or observes some value with a timestamp higher than ts. □

OBSERVATION 2 It follows from Lemma 3 that if $r_{ts} \rightarrow r_{ts'}$, then $ts \leq ts'$. As such, there is a partial order $\prec$ on read operations by benign clients defined by the timestamps associated with the values returned (i.e., of the write operations read). More formally, $r_{ts} \prec r_{ts'} \iff ts < ts'$.

Ordering reads according to the timestamps of the write operations whose values they return yields a partial order on read operations. Lemma 3 ensures that this partial order is consistent with precedence among reads. Therefore, any way of extending this partial order to a total order yields an ordering of reads that is consistent with precedence among reads. Lemmas 2 and 3 guarantee that this totally ordered set of operations is consistent with precedence. This implies the natural extension of linearizability to our fault model (i.e., ignoring reads and durations of writes by Byzantine clients); in particular, it implies linearizability as originally defined [16] if all clients are benign.

8.2 Proof of liveness

Our protocol provides a strong liveness property, namely wait-freedom [14, 18]. Informally, each operation by a correct client completes with certainty, even if all other clients fail, provided that at most b servers suffer Byzantine failures and t servers fail in total.

LEMMA 4 *A write operation by a correct client completes.*

Proof sketch: A write operation by a correct client waits for Q_{W} storage-nodes to execute write requests before returning (line 7 of `DO_WRITE`). Since, $N-t$ storage-nodes are always available and $Q_{\mathrm{W}} \leq N-t$, write operations always terminate. □

LEMMA 5 *A read operation by a correct client completes.*

Proof sketch: Given $N-t$ `READ_RESPONSE` messages, a read operation classifies a candidate as complete, repairable, or incomplete. The read completes if a candidate is classified as complete. As well, the read completes if a candidate is repairable. Repair is initiated for repairable candidates—repair performs a write operation, which by Lemma 4 completes—which lets the read operation complete. In the case of an incomplete, the read operation traverses the version history backwards, until a complete or repairable candidate is discovered. Traversal of the version history terminates if $\perp$ at logical time $\mathbf{0}$ is encountered at Q_{W} storage-nodes. □