Efficient Byzantine-tolerant erasure-coded storage

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

CMU-PDL-03-104
December 2003

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

Abstract

This paper describes a decentralized consistency protocol for survivable storage that exploits local data versioning within each storage-node. Such 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. By exploiting versioning storage-nodes, the protocol shifts most work to clients and allows highly optimistic operation: reads occur in a single round-trip unless clients observe concurrency or write failures. Measurements of a storage system prototype using this protocol show that it scales well with the number of failures tolerated, and its single request response time compares favorably with an efficient implementation of Byzantine-tolerant state machine replication.

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

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

1 Introduction

Survivable storage systems spread data redundantly across a set of distributed 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 and the remainder can be crash. The protocol also supports use of 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 [51, 52].

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 five features. First, the space-efficiency of m-of-n erasure codes can be substantial, reducing communication overheads significantly. 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 additional work. Most studies of distributed storage systems (e.g., [4, 37]) 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. Third, incomplete writes are replaced by subsequent writes or reads (that perform repair), thus preventing future reads from incurring any additional cost; when subsequent writes do the fixing, additional overheads are never incurred. Fourth, most protocol processing is performed by clients, increasing scalability via the well-known principle of shifting work from servers to clients [24]. Fifth, the protocol only requires the use of cryptographic hashes, rather than more expensive cryptographic primitives (e.g., digital signatures).

This paper describes the protocol in detail, develops bounds for thresholds in terms of the number of failures tolerated (i.e., the protocol requires at least $2t+2b+1$ storage-nodes), and provides a proof sketch of its safety and liveness. It also describes and evaluates its use in a prototype storage system called PASIS [52]. To demonstrate that our protocol is efficient in practice, we compare its performance to BFT [8, 9], the Byzantine fault-tolerant replicated state machine implementation that Castro and Liskov have made available [10]. Experiments show that PASIS scales better than BFT in terms of network utilization at the server and in terms of work performed by the server. Experiments also show that response times of PASIS and BFT are comparable.

This protocol is timely because many research storage systems are investigating practical means of achieving high fault tolerance and scalability. Examples include the FARSITE project at Microsoft Research [2], the Federated Array of Bricks project at HP Labs [17], the IceCube project at IBM [35], the OceanStore project at Berkeley [28], and the Self-* Storage project at CMU [18]. Some of these projects (e.g., [2, 28]) use Castro's Byzantine Fault Tolerant (BFT) library [9]. Many of these projects (e.g., [17, 18, 28, 35]) are considering the use of erasure codes for data storage. Our protocol for Byzantine-tolerant erasure-coded storage can provide an efficient, scalable, highly fault-tolerant foundation for such storage systems.

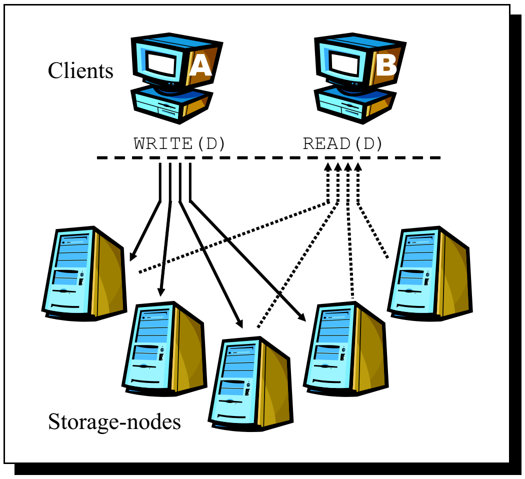

Figure 1: **High-level architecture for survivable storage.** Spreading data redundantly across storage-nodes improves its fault-tolerance. Clients write and (usually) read data from multiple storage-nodes.

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 access to data-items even when subsets of the storage-nodes have failed. One difficulty created by this architecture is the need for a consistent view, across storage-nodes, of the most recent update. Without such consistency, data loss is possible or even likely.

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. Since each storage-node has a complete instance of the data-item, the main difficulty is identifying and retaining the most recent instance. Alternately, more space-efficient erasure coding schemes can be used to reduce network load and storage consumption. With erasure coding schemes, reads require fragments from multiple servers. Moreover, to decode the data-item, the set of fragments read must correspond to the same write operation.

To provide reasonable semantics, storage systems 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 [23] and wait-free [21] read-write objects. Linearizability is adapted appropriately for Byzantine clients, and wait-freedom is as in the model of Jayanti et al. [25].

Most prior systems implementing Byzantine fault-tolerant services adopt the replicated state machine approach [43], 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 [16, 21, 25]. Instead, such systems achieve liveness only under stronger timing assumptions, such as synchrony (e.g., [12, 40, 45]) or partial synchrony [14] (e.g., [9, 26, 42]), or probabilistically (e.g., [6]). An alternative is Byzantine quorum systems [30], from which our protocol inherit techniques (i.e., our protocol can be considered a Byzantine quorum system that uses the threshold quorum construction). Protocols for supporting a linearizable implementation of any deterministic object using Byzantine quorums have been developed (e.g., [33]), but also necessarily forsake wait-freedom to do so. Additionally, most Byzantine quorum systems utilize digital signatures which are computationally expensive.

Byzantine fault-tolerant protocols for implementing read-write objects using quorums are described in [22, 30, 34, 39]. Of these related quorum systems, only Martin et al. [34] 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 [34] "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 [34] to erasure coded data appears nontrivial. Second, ours provides better message efficiency: [34] 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 [34] requires digital signatures by clients, which in practice is two orders of magnitude more costly than the cryptographic transforms we employ. Advantages of [34] 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 [49], and even under attack this can be managed using a garbage collection mechanism we describe in Section 5.

There exists much prior work (e.g., [3, 22, 36]) that combines erasure coded data (e.g., [41, 44]) with quorum systems to improve the confidentiality and/or integrity of data along with its availability. However, these systems do not provide consistency (i.e., a synchronization mechanism is required) and do not cope with Byzantine clients.

We develop our protocol for a hybrid failure model of storage-nodes (i.e., a mix of crash and Byzantine failures). The concept of hybrid failure models was introduced by Thambidurai and Park in [50]; other protocols have been developed for such failure models (e.g., Garay and Perry [19] consider reliable broadcast, consensus and clock synchronization in the hybrid failure model and Malkhi, Reiter and Wool [31] consider the resilience of Byzantine quorum systems to crash faults).

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 [29]; the remainder are assumed to crash. We make no assumptions about the behavior of Byzantine storage-nodes and Byzantine clients (e.g., 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 benefit from 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, except that it is non-zero). 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; correct storage-nodes host a version of the data-fragment for each write request they execute. There is a well known zero time, **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 **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 (note, in [32], repair, for replicas, is referred to as “write-back”). 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 [41], 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 [44] and Short Secret Sharing [27]) 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 [20] 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. [34]. 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 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 reader must validate the cross checksum. 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. Instead of using validated cross checksums, our protocol could use Verifiable Secret Sharing [11, 15]. Verifiable Secret Sharing enables storage-nodes to validate that the client acted correctly on each write request (instead of validating the data-item on each read operation).

4.2.4 Authentication

Clients and storage-nodes must be able to validate the authenticity of messages. We use an authentication scheme based on pair-wise shared secrets (e.g., between clients and storage-nodes), in which RPC argu-

```
WRITE($Data$) :
$Time := $ READ_TIMESTAMP()
$\{D_1, \ldots, D_N\} := $ ENCODE($Data$)
$CC := $ MAKE_CROSS_CHECKSUM($\{D_1, \ldots, D_N\}$)
$LT := $ MAKE_TIMESTAMP($Time$, $CC$)
DO_WRITE($\{D_1, \ldots, D_N\}$, $LT$, $CC$)

READ_TIMESTAMP() :
for all $S_i \in \{S_1, \ldots, S_N\}$ do
    SEND($S_i$, TIME_REQUEST)
end for
$ResponseSet := \emptyset$
repeat
    $ResponseSet :=$
            $ResponseSet \cup$ RECEIVE($S$, TIME_RESPONSE)
until ($|ResponseSet| = N - t$)
$Time := $ MAX[$ResponseSet.LT.Time$]
RETURN($Time$)

MAKE_CROSS_CHECKSUM($\{D_1, \ldots, D_N\}$) :
for all $D_i \in \{D_1, \ldots, D_N\}$ do
    $H_i := $ HASH($D_i$)
end for
$CC := H_1 | \ldots | H_N$
RETURN($CC$)

MAKE_TIMESTAMP($LT_{\mathrm{max}}$, $CC$) :
$LT.Time := LT_{\mathrm{max}}.Time + 1$
$LT.Verifier := $ HASH($CC$)
RETURN($LT$)

DO_WRITE($\{D_1, \ldots, D_N\}$, $LT$, $CC$) :
for all $S_i \in \{S_1, \ldots, S_N\}$ do
    SEND($S_i$, WRITE_REQUEST, $LT$, $D_i$, $CC$)
end for
$ResponseSet := \emptyset$
repeat
    $ResponseSet :=$
            $ResponseSet \cup$ RECEIVE($S$, WRITE_RESPONSE)
until ($|ResponseSet| = N - t$)
```

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

```
READ() :
$ResponseSet := $ DO_READ(READ_LATEST_REQUEST, $\perp$)
loop
    $\langle CandidateSet, LT_{\mathrm{candidate}} \rangle :=$
            CHOOSE_CANDIDATE($ResponseSet$)
    if ($|CandidateSet| \geq$ INCOMPLETE then
        /* Complete or repairable write found */
        $\{D_1, \ldots, D_N\} := $ GENERATE_FRAGMENTS($CandidateSet$)
        $CC_{\mathrm{valid}} := $ MAKE_CROSS_CHECKSUM($\{D_1, \ldots, D_N\}$)
        if ($CC_{\mathrm{valid}} = CandidateSet.CC$) then
            /* Cross checksum is validated */
            if ($|CandidateSet| <$ COMPLETE) then
                /* Repair is necessary */
                DO_WRITE($\{D_1, \ldots, D_N\}$, $LT_{\mathrm{candidate}}$, $CC_{\mathrm{valid}}$)
            end if
            $Data := $ DECODE($\{D_1, \ldots, D_N\}$)
            RETURN($\langle LT_{\mathrm{candidate}}, Data \rangle$)
        end if
    end if
    /* Incomplete or cross checksum not validated, loop again */
    $ResponseSet := $ DO_READ(READ_PREV_REQUEST, $LT_{\mathrm{candidate}}$)
end loop

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

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

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

ments and replies are accompanied by an HMAC [5] (using the shared secret as the key). We assume an infrastructure is in place to distribute shared secrets. Our implementation is able to make use of an existing Kerberos [47] infrastructure.

4.3 Pseudo-code

The pseudo-code for the protocol is shown in Figures 2 and 3. The symbol LT denotes logical time and $LT_{\mathrm{candidate}}$ 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 binary operator '$|$' denotes string concatenation. Simplicity and clarity in the presentation of the pseudo-code was chosen over obvious optimizations that are in the actual implementation.

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 $N-t$ responses, on line 7 of `READ_TIMESTAMP`, ensures that the response set intersects a complete write at a correct storage-node. Since the environment is asynchronous, a client can wait for no more than $N-t$ responses. Fewer than $N-t$ responses are actually required to observe the timestamp of the latest complete write, since a single correct response is sufficient.

Next, the `ENCODE` function, on line 2 of `WRITE`, 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 3). The function `MAKE_TIMESTAMP`, called on line 4, 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., $LT.TIME$) 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 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 $N-t$ `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. 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, 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`, called 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 based on the size of the *CandidateSet*. The rules for classifying a candidate as `INCOMPLETE` or `COMPLETE` are given in the following subsection. If the candidate is classified as incomplete, a `READ_PREV_REQUEST` message is sent to each storage-node with its 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 could occur if the read set (of $N-t$ storage-nodes) does not fully encompass the write set (of $N-t$ storage-nodes).

4.4 Protocol constraints

The symbol Q_{C} denotes a complete write operation: the number of benign storage-nodes that must execute write responses for a write operation to be complete. Note that since threshold quorums are used, Q_{C} is a scalar value. To ensure that linearizability and liveness are achieved, Q_{C} 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 safety and liveness proofs for the protocol in Appendix I.

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

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

Since slow storage-nodes cannot be differentiated from crashed storage-nodes, only $N-t$ responses can be awaited. As well, b responses received may be from Byzantine storage-nodes.

READ CLASSIFICATION: To classify a candidate as `COMPLETE`, a candidate set of at least Q_{C} benign storage-nodes must be observed. In the worst case, at most b members of the candidate set may be Byzantine, thus,

$$|\mathit{CandidateSet}|-b \geq Q_{\text{C}} \Rightarrow \texttt{COMPLETE}. \tag{2}$$

To classify a candidate as `INCOMPLETE` a client must determine that a complete write does not exist in the system (i.e., fewer than Q_{C} benign storage-nodes host the write). For this to be the case, the client must have queried all possible storage-nodes ($N-t$), and must assume that nodes not queried host the candidate in consideration. So,

$$|\mathit{CandidateSet}|+t < Q_{\text{C}} \Rightarrow \texttt{INCOMPLETE}. \tag{3}$$

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

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

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

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

CONSTRAINT SUMMARY:

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

5 Evaluation

This section evaluates the performance and scalability of the consistency protocol in the context of a prototype storage system called PASIS [52]. We compare the PASIS implementation of our protocol with the BFT library implementation [10] of the BFT protocol for replicated state machines [7], since it is generally regarded as efficient.

5.1 PASIS 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

PASIS storage-nodes use the Comprehensive Versioning File System (CVFS) [46] 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 [46, 49].

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, which makes candidate classification more network-efficient.

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 for the corresponding data-item. Storage-nodes are able to classify writes by executing the read consistency protocol in the same manner as the client. Further discussion of the implications of bounded storage capacity on the protocol is provided in Section 6.

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 $Q_C + b$ storage-nodes. PASIS utilizes *read witnesses* to make read operations more network efficient; only m of the initial requests request the data-fragment, while all request version histories. 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 total amount of data stored over the size of the data-item. We use an information dispersal algorithm [41] 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 [13]. We modified the source to make use of stripe-fragments and added an implementation of Galois Fields of size 2^8 that use lookup tables for multiplication.

Our implementation of cross checksums closely follows Gong [20]. 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 1 GHz Pentium III machine with 384 MB of memory. Storage-nodes use a 9 GB Quantum Atlas 10K as the storage device. The machines are connected through a 100 Mb switch. All machines run the Linux 2.4.20 SMP kernel.

In all experiments, clients keep a single read or write operation for a random 16 KB block outstanding. Once an operation completes, a new operation is issued (there is no think time). 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 PASIS, $N = 4b+1$, $Q_C = 2b+1$, 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. All messages are authenticated using HMACs; pair-wise symmetric keys are distributed prior to each experiment.

	b=1	b=2	b=3	b=4
Erasure coding	1250	1500	1730	1990
Cross checksum	360	440	480	510
Verifier	1.6	2.3	3.6	4.3
Validate	82	58	48	40
Authenticate	1.5	1.5	2.1	2.1

Table 1: **Computation costs in PASIS in μs.** Client and storage-node computation costs using a 1GHz CPU. Erasure coding, cross checksum, and verifier generation are performed by the client for every operation. Validation is performed by storage-nodes. Authentication is performed by both clients and storage-nodes for every request.

5.2.2 BFT configuration

Operations in BFT [7] 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*, lists 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.

An optimistic fast path for read operations (i.e., operations that do not modify state) 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). Re-issued read operations perform agreement using the base BFT algorithm.

The BFT configuration does not store data to disk, instead it stores all data in memory and accesses it via memory offsets (i.e., we implemented a simple block interface using BFT). For all experiments, view changes are suppressed. BFT uses UDP connections rather than TCP. The BFT implementation defaults to using IP multicast. In our environment, like many, IP multicast broadcasts to the entire subnet, thus making it unsuitable for shared environments. We found that the BFT implementation code is fairly fragile when using IP multicast in our environment, making it necessary to disable IP multicast in some cases (where stated explicitly). The BFT implementation authenticates broadcast messages via authenticators, and point to point messages with MACs.

5.3 Mechanism costs

Client and storage-node computation costs in PASIS are listed in Figure 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.

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

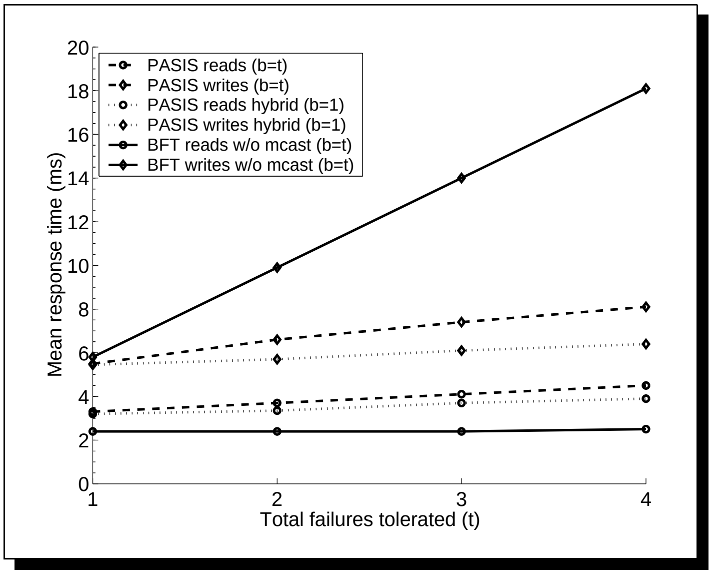

Figure 4: **Mean response time vs. total failures tolerated.** Mean response times of read and write operations of random 16 KB blocks in PASIS and BFT. Lines are shown for PASIS that correspond to both $b = t$ and $b = 1$ (a hybrid fault model). Multicast was not used for these BFT experiments.

to the verifier within the timestamp.

All requests and responses are authenticated via HMACS. The cost of authenticating write requests, listed in the table, is very small. The step-wise increase in computation cost from $b = 2$ to $b = 3$ is due to the set of arguments increasing into a second block. The cost of authenticating read requests and timestamp requests are similar to those listed in the table.

5.4 Performance and scalability

5.4.1 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. Due to the fragility of the BFT implementation with $b > 1$, IP multicast was disabled for BFT during this experiment. 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. In our environment, a key contributor to response time is network cost, which is dictated by the space-efficiency of the protocol.

Figure 5 breaks the mean response times of read and write operations, from Figure 4, 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 response time measurements and scalability of these protocols, it is important to understand these breakdowns.

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. For $b = 4$, BFT has a blowup of $13\times$ on the network (due to replication), whereas our protocol has a blowup of $\frac{17}{5} = 3.4\times$ on the network. With IP multicast the response time of the BFT write operation would improve significantly, since the client would not need to serialize 13 replicas over its link. However, IP multicast does not reduce the aggregate server network utilization of BFT—for $b = 4$, 13 replicas must be delivered.

PASIS has longer response times than BFT for read operations. This can be attributed to two main

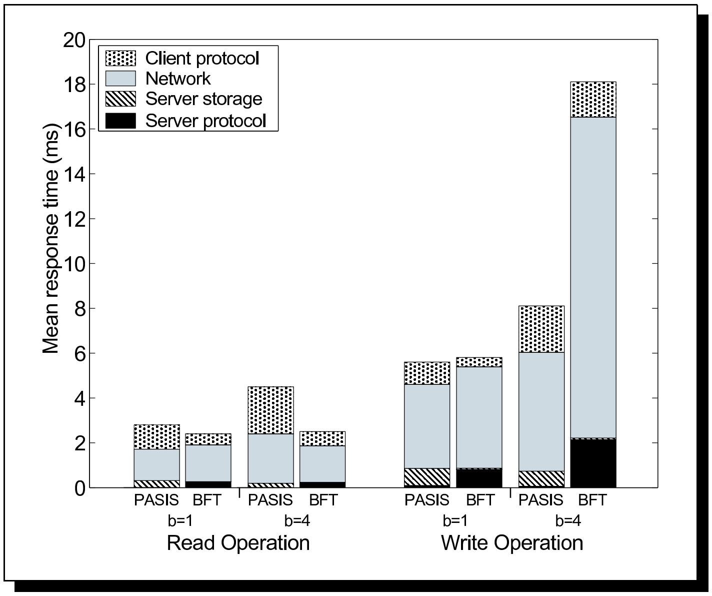

Figure 5: **Protocol cost breakdown.** The bars illustrate the cost breakdown of read and write operations for PASIS and BFT for $b = 1$ and $b = 4$. Each bar corresponds to a single point on the mean response time graph in Figure 4. BFT does not store data to disk, as such no server storage cost is shown for BFT.

factors: First, the PASIS storage-nodes store data in a real file system; since the BFT-based block store 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 actual data storage would incur server storage costs similar to PASIS (e.g., around 0.7 ms for a write and 0.4 ms for a read operation, as is shown for PASIS with $b = 1$ in Figure 5). Indeed, the difference in read response time between PASIS and BFT at $b = 1$ is mostly accounted for by server storage costs. Second, for our protocol, the client computation cost grows as the number of failures tolerated increases because the cost of generating data-fragments grows as N increases.

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 $m = t + 1$. Consequently, $Q_{\text{C}} = 2t + 1$ and $N = 3t + 2$. 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, the response times of 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.

5.4.2 Throughput

Figure 6 shows the throughput in 16 KB requests per second as a function of the number of clients (one request per client) for $b = 1$. Read and write operations are evaluated separately. Since $b = 1$ in this experiment, BFT uses multicast (which greatly improves its network efficiency). PASIS was run in two configurations, one with the thresholds set to that of the minimum system with $m = 2$, $N = 5$ (write blowup of 2.5×), and one, more space-efficient, with $m = 3$, $N = 6$ (write blowup of 2×). Results indicate that, at high client load, throughput is limited by the server network bandwidth.

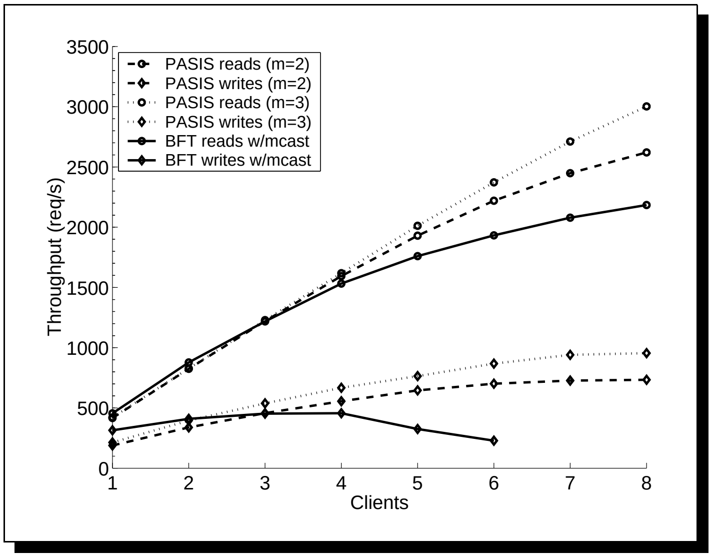

Figure 6: **Throughput vs. number of clients ($b = 1$).** Throughput of read and write operations of random 16 KB blocks in PASIS and BFT for $b = 1$. Each client had one request outstanding. For PASIS, lines corresponding to both $m = 2, N = 4$ and $m = 3, N = 5$ are shown. For BFT, multicast was used.

At high load, PASIS has greater write throughput than BFT. BFT's write throughput flattens out at 456 requests per second. We observed BFT's write throughput drop off as client load increased; likewise, we observed a large increase in request retransmissions. We believe that this is due to the use of UDP and a coarse grained retransmit policy in BFT's implementation. The write throughput of PASIS flattens out at 733 requests per second, significantly outperforming BFT. This is because of the network-efficiency of PASIS. Even with multicast enabled, each BFT server link sees a full 16 KB replica, whereas each PASIS server link sees $\frac{16}{m}$ KB. Similarly, due to network space-efficiency, the PASIS configuration using $m = 3$ outperforms the minimal PASIS configuration (954 requests per second).

Both PASIS and BFT have roughly the same network utilization per read operation (16 KB per operation). To be network-efficient, PASIS uses read witnesses and BFT uses "fast path" read operations. However, PASIS makes use of more storage-nodes than BFT does servers. As such, the aggregate bandwidth available for reads is greater for PASIS than for BFT, and consequently PASIS has a greater read throughput than BFT. Although BFT could add servers to increase its read throughput, doing so would not increase its write throughput (indeed, write throughput would likely drop due to the extra inter-server communication).

5.4.3 Scalability summary

For PASIS and BFT, scalability is limited by either the server network utilization or server cpu utilization. Figure 5 shows that PASIS scales better than BFT in both. Consider write operations. Each BFT server receives an entire replica of the data, whereas each PASIS storage-node receives a data-fragment $\frac{1}{m}$ the size of a replica. The work performed by BFT servers for each write request grows with b. 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. We believe that the server cost for BFT increases because the number of messages that must be sent to all other servers increases.

5.5 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

BYZANTINE CLIENTS: In a storage system, Byzantine clients can write arbitrary values. The use of fine-grained versioning (e.g., self-securing storage [49]) facilitates detection, recovery, and diagnosis from storage intrusions [48]. Once discovered, 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, storage-based intrusion detection [38] is probably sufficient to detect such client actions.

TIMESTAMPS FROM BYZANTINE STORAGE-NODES: Byzantine storage-nodes can fabricate high timestamps that must be classified as incomplete by read operations. Worse, in each subsequent round of a read operation, Byzantine storage-nodes can fabricate more high timestamps that are just a bit smaller than the previous. In this manner, Byzantine storage-nodes can "attack" the performance of the read operation, but not its safety. To protect against such denial-of-service attacks, the read operation can consider all unique timestamps, up to a maximum of $b+1$, present in a *ResponseSet* as candidates before soliciting another *ResponseSet*. In this manner, each "round" of the read operation is guaranteed to consider at least one candidate from a correct storage-node and no more than b candidates from Byzantine storage-nodes.

GARBAGE COLLECTION: The proof of liveness (i.e., of wait-freedom) given in Appendix I assumes unbounded storage capacity. In practice, storage capacity is bounded; if storage capacity is exhausted, wait-freedom cannot be guaranteed. Prior experience indicates that it takes weeks of normal activity to exhaust the capacity of modern disk systems that version all write requests [49].

Garbage collection is used to avoid storage exhaustion. In doing so, it can interact with concurrent read operations and concurrent write operations in such a manner that a read operation must be retried. Specifically a read operation could classify a concurrent write operation as incomplete, the write operation could then complete, and garbage collection could then delete all previous complete writes. If this occurs, the read operation's next round will observe an incomplete write with no previous history. Effectively, the read operation has "missed" the complete write operation that it would have classified as such. When it discovers this fact, the read operation retries (i.e., restarts by requesting a new *ResponseSet*). Thus, in theory, a read operation faced with perpetual write concurrency and garbage collection may never complete. In practice, such perpetual interaction of garbage collection and read-write concurrency for a given data-item is not realistic.

7 Conclusion

Building highly scalable, fault-tolerant storage systems is an active area of research. We have developed an efficient Byzantine-tolerant protocol for reading and writing blocks of data. Experiments demonstrate that PASIS, a prototype storage system that uses our protocol, scales well in the number of faults tolerated, supports 60% greater write throughput than BFT, and requires significantly less server computation than BFT.

Acknowledgements

We thank Craig Soules for his timely support of CVFS (the Comprehensive Versioning File System). We thank Carla Geisser and Terrence Wong for their help getting BFT to run in our environment and building the experimental infrastructure. 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] D. Agrawal and A. El Abbadi. Integrating security with fault-tolerant distributed databases. *Computer Journal*, **33**(1):71–78, 1990.

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

[5] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. *Advances in Cryptology - CRYPTO* (Santa Barbara, CA, 18–22 August 1996), pages 1–15. Springer-Verlag, 1996.

[6] 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.

[7] 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.

[8] Miguel Castro and Barbara Liskov. Byzantine fault tolerance can be fast. *Dependable Systems and Networks*, pages 513–518, 2001.

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

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

[11] B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch. Verifiable Secret Sharing in the Presence of Faults. *IEEE Symposium on Foundations of Computer Science*, pages 335–344. IEEE, 1985.

[12] 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.

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

[14] 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.

[15] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. *IEEE Symposium on Foundations of Computer Science*, pages 427–437. IEEE, 1987.

[16] 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.

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

[18] Gregory R. Ganger, John D. Strunk, and Andrew J. Klosterman. *Self-* Storage: brick-based storage with automated administration*. Technical Report CMU-CS-03-178. Carnegie Mellon University, August 2003.

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

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

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

[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] 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.

[26] 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.

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

[28] 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.

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

[30] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *ACM Symposium on Theory of Computing*, pages 569–578. ACM, 1997.

[31] Dahlia Malkhi, Michael Reiter, and Avishai Wool. The load and availability of byzantine quorum systems. *SIAM Journal of Computing*, **29**(6):1889–1906. Society for Industrial and Applied Mathematics, April 2000.

[32] Dahlia Malkhi and Michael K. Reiter. Secure and scalable replication in Phalanx. *IEEE Symposium on Reliable Distributed Networks* (West Lafayette, IN, 20–23 October 1998), 1998.

[33] 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.

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

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

[36] R. Mukkamala. Storage efficient and secure replicated distributed databases. *IEEE Transactions on Knowledge and Data Engineering*, **6**(2):337–341, 1994.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

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

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

[45] 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.

[46] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger,. Metadata efficiency in versioning file systems. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–02 April 2003), pages 43–58. USENIX Association, 2003.

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

[48] 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.

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

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

[51] 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.

[52] 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.

Appendix I: Proofs

Proof of safety

This section sketches a proof that our protocol implements linearizability [23] 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_C 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 $o_1 \not\to o_2$ 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 $w_{ts} \to w_{ts'}$, then $ts < ts'$.*

Proof sketch: A complete write operation executes at at least Q_C benign storage-nodes (cf. Definition 3). Since $w_{ts} \to w_{ts'}$, w_{ts} is complete. Since the `READ_TIMESTAMP` function collects $N-t$ `TIME_RESPONSE` messages and w_{ts} is complete, $w_{ts'}$ observes at least $b+1$ `TIME_RESPONSE` messages from correct storage-nodes that executed w_{ts} (remember, $b+t < Q_C$). As such, $w_{ts'}$ will observe some timestamp greater than or equal to ts and thus construct ts' to be greater than ts. 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 $w_{ts} \to r_{ts'}$, then $ts \leq ts'$.*

Proof sketch: By Definition 3, since w_{ts} completes, there are Q_C 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_C accepted write requests from $r_{ts'}$. As such, at least $Q_C - t$ responses with timestamp ts are observable by $r_{ts'}$; a read operation that observes a candidate with at least $Q_C - t$ 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 $r_{ts} \to r_{ts'}$, 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 [23] if all clients are benign.

Proof of liveness

Our protocol provides a strong liveness property, namely wait-freedom [21, 25]. 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. Note, we assume that storage-nodes have unbounded storage capacity in this proof (i.e., that the entire version history back to the initial value $\perp$ at time $\mathbf{0}$ is available at each storage-node).

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

Proof sketch: A write operation by a correct client waits for $N-t$ responses from storage-nodes before returning (cf. `DO_WRITE`). Since, $Q_C \leq N-t-b$ (cf. (1) in Section 4), 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_{C} storage-nodes. $\square$