Appears in the Proceedings of the International Conference on Dependable Systems and Networks, June 2004.

Efficient Byzantine-tolerant erasure-coded storage

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

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 performance compares favorably with an efficient implementation of Byzantine-tolerant state machine replication.

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 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 the 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.

Briefly, the protocol works as follows. To perform a write, a client erasure codes a data-item into a set of fragments, determines the current logical time and then writes the time-stamped fragments to at least a threshold set of storage-nodes. Storage-nodes keep all versions of the fragments they are sent (in practice, until garbage collection frees them). To perform a read, a client fetches the latest fragment versions from a subset 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 fragments are then decoded and the data-item is returned.

The protocol gains efficiency from five features. First, it supports the use of space-efficient m-of-n erasure codes that can substantially reduce communication overheads. 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., [1, 25]) 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 [17]. Fifth, the protocol only requires the use of cryptographic hashes, rather than more expensive digital signatures.

This paper describes the protocol in detail and develops bounds for thresholds in terms of the number of failures tolerated (i.e., the protocol requires at least $2t+2b+1$ storage-nodes). It also describes and evaluates its use in a prototype storage system called PASIS [32]. To demonstrate that our protocol is efficient in practice, we compare its performance

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. This work is also supported by the Army Research Office through grant number DAAD19-02-1-0389 to CyLab at Carnegie Mellon University.

to BFT [3, 4], the Byzantine fault-tolerant replicated state machine implementation that Castro and Rodrigues have made available [5]. Experiments show that PASIS scales better than BFT in terms of server network utilization and in terms of work performed by the server. Experiments also show that response times of BFT (using multicast) and PASIS are comparable.

This protocol is timely because many research storage systems are investigating practical means of achieving high fault tolerance and scalability of which some are considering the support of erasure-coded data (e.g., [8, 10, 19, 24]). Our protocol for Byzantine-tolerant erasure-coded storage can provide an efficient, scalable, highly fault-tolerant foundation for such storage systems.

2. Background and related work

In a fault-tolerant, or survivable, distributed storage system, clients write and (usually) read data from multiple 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. To provide reasonable semantics, storage systems must guarantee that readers see consistent data-item values. Specifically, the linearizability [16] of operations is desirable for a shared storage system.

A common data distribution scheme used in such systems is replication. 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.

Most prior systems implementing Byzantine fault-tolerant services adopt the replicated state machine approach [27], 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 [7, 14, 18]. Instead, such systems achieve liveness only under stronger timing assumptions. An alternative to state machine replication is a Byzantine quorum system [21], from which our protocol inherits 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., [22]), but also necessarily forsake wait-freedom to do so. Additionally, most protocols accessing Byzantine quorum systems utilize computationally expensive digital signatures.

Byzantine fault-tolerant protocols for implementing read-write objects using quorums are described in [15, 21, 23]. Of these related quorum systems, only Martin et al. [23] achieve linearizability in our fault model. 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 [23] "listen" for updates (versions) from storage-nodes until a complete write is observed. Conceptually, our approach differs in that clients read 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 [23] to erasure coded data appears nontrivial. Second, ours provides better message efficiency. Third, ours requires less computation, in that we do not require the use of expensive digital signatures. Advantages of [23] 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 [30], and even under attack this can be managed using the garbage collection mechanism we describe in Section 5.1.1.

Frølund et al. [9] have developed a decentralized access protocol for erasure-coded data under the crash-recovery storage-node failure model. 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 [31].

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 [20]; the remainder are assumed to crash. We make no assumptions about the behavior of Byzantine storage-nodes and Byzantine clients. A client or storage-node that does not exhibit a Byzantine failure (it is either correct or crashes) is *benign*.

Our protocol tolerates Byzantine faults in any number of clients and a limited number of storage nodes while implementing linearizable and wait-free read-write objects. Linearizability is adapted appropriately for Byzantine clients (we discuss the necessary adaptations in [13]). Wait-freedom functions as in the model of Jayanti et al. [18] and assumes no storage exhaustion.

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.

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-items and data-fragments. 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, $\mathbf{0}$, 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, which 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 linearizability and wait-freedom.

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. Data-fragments are generated by erasure-coding data-items. 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 threshold 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 candidate's value is decoded and returned; 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 We use threshold erasure coding schemes, in which 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. Moreover, any m of the data-fragments can deterministically generate the other $N - m$ data-fragments.

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 [11] 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. These attacks are similar to *poisonous writes* for replication as described by Martin et al. [23]. 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. Via this approach, 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: To ensure that the client that performed a 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.

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 arguments and replies are accompanied by an HMAC [2] (using the shared secret as the key). We assume an infrastructure is in place to distribute shared secrets. Our implementation uses an existing Kerberos [29] infrastructure.

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

READ_TIMESTAMP() :
  for all S_i ∈ {S_1, ..., S_N} do
     SEND(S_i, TIME_REQUEST)
  end for
  ResponseSet := ∅
  repeat
     ResponseSet := ResponseSet ∪ RECEIVE(S, TIME_RESPONSE)
  until (UNIQUE_SERVERS(ResponseSet) = N − t)
  Time := MAX[ResponseSet.LT.Time]
  RETURN(Time)

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

MAKE_TIMESTAMP(LT_max, CC) :
  LT.Time := LT_max.Time + 1
  LT.Verifier := HASH(CC)
  RETURN(LT)

DO_WRITE({D_1, ..., D_N}, LT, CC) :
  for all S_i ∈ {S_1, ..., S_N} do
     SEND(S_i, WRITE_REQUEST, LT, D_i, CC)
  end for
  ResponseSet := ∅
  repeat
     ResponseSet := ResponseSet ∪ RECEIVE(S, WRITE_RESPONSE)
  until (UNIQUE_SERVERS(ResponseSet) = N − t)
```

Figure 1. Write operation pseudo-code.

4.3. Pseudo-code

The pseudo-code for the protocol is shown in Figures 1 and 2. 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 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 with a specific logical time (`WRITE`); query the greatest logical time of a hosted data-fragment (`TIME_REQUEST`); read the hosted data-fragment with the greatest logical time (`READ_LATEST`); and read the hosted data-fragment with the greatest logical time at or before some logical time (`READ_PREV`). Due to its simplicity, storage-node pseudo-code is omitted.

4.3.2. Write operation The `WRITE` operation consists of determining the greatest logical timestamp, constructing

```
READ() :
 1: ResponseSet := DO_READ(READ_LATEST_REQUEST, ⊥)
 2: loop
 3:   ⟨CandidateSet, LT_candidate⟩ :=
          CHOOSE_CANDIDATE(ResponseSet)
 4:   if (|CandidateSet| ≥ INCOMPLETE 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:       /* Cross checksum is validated */
10:       if (|CandidateSet| < COMPLETE) 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(Data)
16:     end if
17:   end if
18:   /* Incomplete or cross checksum did not validate, loop again */
19:   ResponseSet := DO_READ(READ_PREV_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 (UNIQUE_SERVERS(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 2. Read operation pseudo-code.

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. In practice, the timestamp of the latest complete write can be observed with fewer responses.

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 WRITE_RESPONSE messages are received from $N-t$ storage-nodes (line 7 of DO_WRITE). Since the environment is asynchronous, a client can wait for no more than $N-t$ responses. The function UNIQUE_SERVERS determines how many unique servers are present in the candidate set; it ensures that only a single response from each Byzantine storage-node is counted.

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.

The read operation begins by issuing READ_LATEST requests 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. Although not shown in the pseudo-code, the client only considers responses from storage-nodes to READ_PREV requests that have timestamps strictly less than that given in the request.

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 *CandidateSet*. 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 candidate set. The definitions of INCOMPLETE and COMPLETE are given in the following subsection. If the candidate is classified as incomplete, a READ_PREV request 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 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 then 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 for a candidate set classified as complete 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 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.

4.4. Protocol constraints

The symbol Q_{C} denotes a complete write operation: the threshold number of benign storage-nodes that must execute write requests for a write operation to be complete. To ensure that linearizability and wait-freedom 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 present safety and liveness proofs for the protocol and discuss the concept of linearizability in the presence of Byzantine clients in [13].

WRITE COMPLETION: To ensure that a correct client can complete a write operation,

$$Q_{\text{C}} \leq N - t - b. \quad (1)$$

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

$$|CandidateSet| - b \geq Q_{\text{C}}, \text{ so } \texttt{COMPLETE} = Q_{\text{C}} + b. \quad (2)$$

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,

$$|CandidateSet| + t < Q_{\text{C}}, \text{ so } \texttt{INCOMPLETE} = Q_{\text{C}} - t. \quad (3)$$

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),

$$b + t < Q_{\text{C}}. \quad (4)$$

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):

$$1 \leq m \leq Q_{\text{C}} - t. \quad (5)$$

CONSTRAINT SUMMARY:

$$t + b + 1 \leq Q_{\text{C}} \leq N - t - b;$$
$$2t + 2b + 1 \leq N;$$
$$1 \leq m \leq Q_{\text{C}} - t.$$

5. Evaluation

This section evaluates the performance and scalability of the consistency protocol in the context of a prototype storage system called PASIS [32]. We compare the PASIS implementation of our protocol with the BFT library implementation [5] of Byzantine fault-tolerant replicated state machines [4], 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) [28] 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 [28, 30].

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. 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. Storage-nodes 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 storage-nodes. A storage-node in isolation, by the nature of the protocol, cannot determine which local data-fragment versions are safe to remove. 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. We discuss garbage collection more fully in [12].

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.

Initially, read requests are issued to $Q_{\mathrm{C}} + b$ 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 [26] 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. Our implementation of polynomial interpolation was originally based on publicly available code [6] for information dispersal [26]. 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 [11]. We use MD5 for all hashes; thus, each cross checksum is $N \times 16$ bytes long. Note, that if very small blocks are used with a large N, then the overhead due to size of the cross checksum could be substantial.

5.2. Experimental setup

We use a cluster of 20 machines to perform our 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.

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.

5.2.2. BFT configuration Operations in BFT [4] require agreement among the replicas (storage-nodes in PASIS). BFT requires $N = 3b + 1$ replicas to achieve agreement. 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 enable them to exchange symmetric cryptographic keys used to create MACs. Logs of commit messages are checkpointed (garbage collected) periodically.

An optimistic fast path for read-only operations is implemented in BFT. The client broadcasts its request to all replicas. Each replica replies once all previous requests have 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. Re-issued read operations perform agreement using the base BFT algorithm.

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

Table 1. Computation costs in PASIS in μs.

The BFT configuration does not store data to disk, instead it stores all data in memory and accesses it via memory offsets. For all experiments, BFT view changes are suppressed. BFT uses UDP 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 for operations on a 16 KB block 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.

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.

All requests and responses are authenticated via HMACS. The cost of authenticating write requests, listed in the table, is very small. The cost of authenticating read requests and timestamp requests are similar.

5.4. Performance and scalability

5.4.1. Response time Figure 3 shows the mean response time of a single request from a single client as a function of the 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 4 breaks down the mean response times of read and write operations, from Figure 3, 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 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 4). 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 t increases because the cost of generating data-fragments grows as N increases.

In addition to the $b = t$ case, Figure 3 shows one instance of PASIS assuming a hybrid fault model with $b = 1$. For space-efficiency, we set $m = t + 1$. Consequently, $Q_C = 2t + 1$ 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.

5.4.2. Throughput Figure 5 shows the throughput in 16 KB requests per second as a function of the number of clients (one request per client) for $b = 1$. In this experiment, BFT uses multicast, which greatly improves its network efficiency (BFT with multicast is sta-

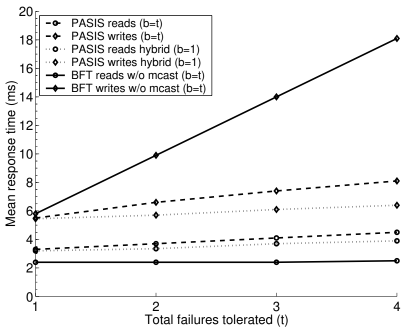

Figure 3. Mean response time.

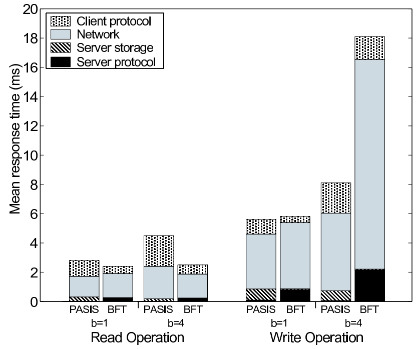

Figure 4. Response breakdown.

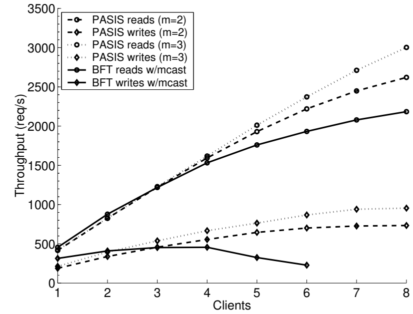

Figure 5. Throughput ($b = 1$).

ble for $b = 1$). 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 $2.5\times$), and one, more space-efficient, with $m = 3$, $N = 6$ (write blowup of $2\times$). Results show that throughput is limited by the server network bandwidth.

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 4 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 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 increases.

5.5. Concurrency

To measure the effect of concurrency on the system, we measure multi-client throughput of PASIS 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 89% of the time, otherwise classification required the traversal of history information. Of these history traversals, repair was only necessary a quarter of the time (i.e., 3% of all reads required repair). Since repair occurs so seldom, the effect on response time and throughput is minimal.

6. Summary

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.

Further evaluation, a proof sketch of the correctness of the protocol, and discussion of additional issues (e.g., garbage collection) can be found in the full technical report [12]. This protocol also extends into a protocol family that includes members for other system models (e.g., asynchronous or synchronous timing model, and crash-recovery failures) [13].

ACKNOWLEDGEMENTS: We would like to thank Craig Soules, Carla Geisser, and Terrence Wong for assistance with the code and experiments. We thank Miguel Castro and Rodrigo Rodrigues for the public implementation of BFT.

References

[1] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a distributed file system. ACM Symposium on Operating System Principles. Published as *Operating Systems Review*, **25**(5):198–212, 13–16 October 1991.

[2] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. Advances in Cryptology - CRYPTO, pages 1–15. Springer-Verlag, 1996.

[3] M. Castro and B. Liskov. Byzantine fault tolerance can be fast. Dependable Systems and Networks, pages 513–518, 2001.

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

[5] M. Castro and R. Rodrigues. *BFT library implementation*. http://www.pmg.lcs.mit.edu/bft/#sw.

[6] W. Dai. *Crypto++*. http://cryptopp.sourceforge.net/docs/ref/.

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

[8] S. Frølund, A. Merchant, Y. Saito, S. Spence, and A. Veitch. FAB: enterprise storage systems on a shoestring. Hot Topics in Operating Systems, pages 133–138. USENIX Association, 2003.

[9] S. Frølund, A. Merchant, Y. Saito, S. Spence, and A. Veitch. A de-centralized algorithm for erasure-coded virtual disks. Dependable Systems and Networks, June 2004.

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

[11] L. Gong. Securely replicating authentication services. International Conference on Distributed Computing Systems, pages 85–91. IEEE Computer Society Press, 1989.

[12] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. *Efficient Byzantine-tolerant erasure-coded storage*. Technical report CMU-PDL-03-104. CMU, December 2003.

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

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

[15] M. P. Herlihy and J. D. Tygar. How to make replicated data secure. Advances in Cryptology - CRYPTO, pages 379–391. Springer-Verlag, 1987.

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

[17] J. H. Howard et al. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, **6**(1):51–81, February 1988.

[18] P. Jayanti, T. D. Chandra, and S. Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM*, **45**(3):451–500. ACM Press, May 1998.

[19] J. Kubiatowicz et al. OceanStore: an architecture for global-scale persistent storage. Architectural Support for Programming Languages and Operating Systems, 2000.

[20] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, **4**(3):382–401. ACM, July 1982.

[21] D. Malkhi and M. Reiter. Byzantine quorum systems. ACM Symposium on Theory of Computing, pages 569–578. ACM, 1997.

[22] D. Malkhi, M. K. Reiter, D. Tulone, and E. Ziskind. Persistent objects in the Fleet system. DARPA Information Survivability Conference and Exposition, pages 126–136. IEEE, 2001.

[23] J.-P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine storage. International Symposium on Distributed Computing, 2002.

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

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

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

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

[28] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata efficiency in versioning file systems. Conference on File and Storage Technologies, pages 43–58. USENIX Association, 2003.

[29] J. G. Steiner, J. I. Schiller, and C. Neuman. Kerberos: an authentication service for open network systems. Winter USENIX Technical Conference, pages 191–202, 9–12 February 1988.

[30] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-securing storage: protecting data in compromised systems. Symposium on Operating Systems Design and Implementation, pages 165–180. USENIX Association, 2000.

[31] P. Thambidurai and Y.-K. Park. Interactive consistency with multiple failure modes. Symposium on Reliable Distributed Systems, pages 93–100. IEEE, 1988.

[32] J. J. Wylie et al. Survivable information storage systems. *IEEE Computer*, **33**(8):61–68. IEEE, August 2000.