well when the number of faults tolerated is small or when writes are small, RAID-5 and RAID-6 perform better for large writes, and all three perform well for reads [11, 39]. The Google File System (GFS) [15], for example, uses replication for a mostly-read workload and by default tolerates two faults.

For distributed storage, as the number of faults tolerated grows beyond two or three, erasure coding provides much better write bandwidth [37, 38]. A few distributed storage systems support erasure coding. For example, Zebra [19], xFS [4], and PanFS [29] support parity-based protection of data striped across multiple servers. FAB [34], Ursa Minor [3], and RepStore [40] support more general m-of-n erasure coding.

2.1 Beyond crash faults

A common assumption is that tolerance of crash faults is sufficient for distributed storage systems, but an examination of a modern centralized storage server shows this assumption to be invalid. Most such servers integrate various checksum and scrubbing mechanisms to detect non-crash faults. One common approach is to store a checksum with every block of data and then verify that checksum upon every read (e.g., ZFS [7] and GFS [15]). This checksum can be used to detect problems such as when the device driver silently corrupts data [7, 15] or when the disk drive writes data to the wrong physical location [3, 7].

For example, if a disk drive overwrites the wrong physical block, a checksum may be able to detect this corruption, but only if the checksum is not overwritten as well. To prevent the checksum from being incorrect along with the data, the checksum is often stored separately (e.g., with the metadata [7, 15]).

In short, various mechanisms are applied to detect and recover from non-crash faults in modern storage systems. These mechanisms are chosen and combined in an ad hoc manner based on the collective experience of the organizations that design storage systems. Such mechanisms are inherently ad hoc because no fault model can describe just the types of faults that must be handled in distributed storage; as systems change, the types of faults change. In the absence of a more specific fault model, general Byzantine fault tolerance [25] can be used to cover all possibilities.

2.2 The cost of Byzantine fault tolerance

Byzantine fault-tolerant m-of-n erasure-coded storage protocols require at least $m+2f$ servers to tolerate f faulty servers. Requiring this many servers is less imposing than it sounds; modern non-Byzantine fault-tolerant erasure-coded storage arrays already use a similar number of disk drives. A typical storage array will have several primary disk drives that store unencoded data (e.g., $m = 5$) and a few parity drives for redundancy (e.g., $f = 2$). Beyond these drives, however, an array will often include a pool of hot spares with f or more drives (sometimes shared with neighboring arrays). The reason for this setup is that once a drive fails or becomes otherwise unresponsive, the storage array must either halt or decrease the number of faults that it tolerates unless it replaces that drive with a hot spare. This setup is similar to providing $m+f$ responsive drives to a Byzantine fault-tolerant protocol but making an additional f drives available as needed (for a similar approach, see Rodrigues et al. [33]). In other words, the specter of additional hardware should not scare developers away from Byzantine fault tolerance.

Byzantine fault-tolerant protocols often require additional computational overhead. For example, our protocol requires data to be cryptographically hashed for each write and read operation. This overhead, however, is less significant than it appears for two reasons. First, data must be hashed anyway if it is to be authenticated when sent over the network. Of course, data must be structured properly to use a hash for both authentication and fault tolerance. Second, many modern file systems hash data anyway. For example, ZFS supports hashing all data with SHA-256 [36], and EMC's Centera hashes all data with either MD5 or a concatenation of MD5 and SHA-256 to provide content addressed storage [30].

2.3 Byzantine fault-tolerant storage

Many Byzantine fault-tolerant protocols are used to implement replicated state machines. Implementations of recent protocols can be quite efficient due to several optimizations. For example, Castro and Liskov eliminate public-key signatures from the common case by replacing signatures with message authentication codes (MACs) and lazily retrieving signatures in cases of failure [10]. Abd-El-Malek et al. use aggressive optimistic techniques and quorums to scale as the number of faults tolerated is increased, but their protocol requires $5f+1$ servers to tolerate f faulty servers [1]. Cowling et al. use a hybrid of these two protocols to achieve good performance with only $3f+1$ servers [13]. Kotla and Dahlin further improve performance by using application-specific information to allow parallelism [24]. Though a Byzantine fault-tolerant replicated state machine protocol can be used to implement a block storage protocol, doing so requires writing data to at least $f+1$ replicas.

When writing large blocks of data and tolerating multiple faults, a Byzantine fault-tolerant storage protocol should provide erasure-coded fragments to each server to minimize the bandwidth overhead of redundancy. Writing erasure-coded fragments has been difficult to achieve because servers must ensure that a block is encoded correctly without seeing the entire block. Goodson et al. introduced PASIS, a Byzantine fault-tolerant erasure-coded block storage protocol [18]. In PASIS, servers do not verify that a block is correctly encoded during write operations. Instead, clients verify during read operations that a block is correctly encoded.

This technique avoids the problem of verifying erasure-coded fragments but introduces a few new ones. First, fragments must be kept in versioned storage [35] because clients may need to read several versions of a block before finding a version that is encoded correctly. Second, the read verification process is computationally expensive. Third, PASIS requires $4f+1$ servers to tolerate f faults. Fourth, a separate garbage collection protocol must eventually be run to free old versions of a block. A lazy verification protocol, which also performs garbage collection, was proposed to reduce the impact of read verification by performing it in the background [2], but this protocol consumes significant bandwidth.

Cachin and Tessaro introduced AVID [8], an asynchronous verifiable information dispersal protocol, which they used to build a Byzantine fault-tolerant block storage protocol that requires only $3f+1$ servers to tolerate f faults [9]. In AVID, a client sends each server an erasure-coded fragment. Each server sends its fragment to all other servers such that each server can verify that the block is encoded correctly. This all-to-all communication, however, consumes slightly more bandwidth in the common case than a typical replication protocol. Our paper describing homomorphic fingerprinting provides a protocol that reduces this overhead but still requires all-to-all communication and the encoding and hashing of $3f+1$ fragments [20, Section 4]. We address these shortcomings in Section 3.1.1.

Many of the problems in PASIS are caused by the need to handle Byzantine faulty clients. Faulty clients should be tolerated in a Byzantine fault-tolerant storage system to prevent such clients from forcing the system into an inconsistent state. For example, though a faulty client can corrupt blocks for which it has write permissions, it must not be allowed to write a value that is read as two differ-

ent blocks by two different correct clients; if not, a faulty client at a bank, e.g., could provide one account balance to the auditors but another to the ATM. Liskov and Rodrigues [26] propose that servers provide public-key signatures to vouch for the state of the system. This technique can be used to tolerate Byzantine faulty clients in a quorum system. In the next section, we will adapt this technique to use only MACs and pseudo-random nonce values in a PASIS-like protocol.

3. THE PROTOCOL

This section describes the block storage protocol. A separate instance is executed for each block, so this section does not discuss block numbers. Section 3.1 describes the design of the protocol, including how it builds on prior protocols. Section 3.2 describes the system model. Section 3.3 provides pseudo-code for write and read operations. Section 3.4 discusses liveness and linearizability. Section 3.5 discusses protocol extensions.

3.1 Design

Consider the following replication-based protocol [26], which requires $3f+1$ servers to tolerate f faults. To write a block, a client hashes the block and sends the hash to all servers (the prepare phase). The servers respond with a signed message containing the hash and a logical timestamp, which is always greater than any timestamp that the server has seen. If there are not at least $2f+1$ matching timestamps, the client requests that each server sign a new message using the greatest timestamp found. The client commits the write by sending the entire block along with $2f+1$ signed messages with matching timestamps to each server. The server verifies that the signatures are valid and that the hash of the block matches the hash in the signed message. Because this protocol uses public-key signatures, the client can verify the responses from the prepare phase before it attempts to commit the write.

To read the block, the client queries all servers for their most recent timestamp and the $2f+1$ signatures generated in the prepare phase. The client reads the block with the greatest timestamp and $2f+1$ correctly signed messages. The signatures allow the the client to verify that $2f+1$ servers provided signatures in the prepare phase, which ensures that some client invoked a write of this block at this timestamp (at least one of these servers is correct and, hence, would only provide signatures to a client in the prepare phase). To ensure that other clients see this block, the client writes it back to any servers with older timestamps.

Sending the entire block causes overhead that could be eliminated by sending erasure-coded fragments instead. Writing erasure-coded fragments, however, poses a problem, in that servers can no longer agree on what is being written. A faulty or malicious client can write fragments that decode to different blocks depending upon which subset of fragments is decoded. PASIS [18] uses a cross-checksum [17], which is a set of hashes of the erasure-coded fragments, to detect such inconsistencies. To write a block, a client requests the most recent logical timestamp from all servers in the first round. In the second round, the client sends each server its fragment, the cross-checksum, and the greatest timestamp found. Unfortunately, PASIS requires $4f+1$ servers to tolerate f faults, so $4f+1$ fragments must be encoded and hashed, which is a significant expense. To read a block, the client reads fragments and cross-checksums from each server, starting with the most recent timestamp, until it finds m fragments whose hashes correspond to their respective locations in the cross-checksum. From these fragments, the client decodes the block, re-encodes n fragments, and recomputes the cross-checksum. If this cross-checksum does not match the one provided by the servers, then the write operation for that timestamp was invalid and the client must try reading fragments at an earlier timestamp. If cross-checksums match, the client writes fragments back to servers as needed.

3.1.1 Improvements

The replication-based protocol requires only $3f+1$ servers but relies on public-key signatures and replication. PASIS improves write bandwidth but has a number of drawbacks, as discussed in Section 2.3. Our protocol improves on these approaches with the following four techniques.

No public-key cryptography: As in the replication-based protocol [26], the response to a prepare request in our protocol includes an authenticated timestamp and checksum that allows the client to progress to the commit phase once enough timestamps match. Castro and Liskov [10] avoid using signatures for authentication by using message authentication codes (MACs) in the common case but lazily retrieving signatures when needed. This lazy retrieval technique does not work for the Liskov and Rodrigues protocol, however, because signatures are stored for later use to vouch for the state of the system [26, Section 3.3.2].

Instead, we eschew signatures altogether and rely entirely on MACs and random nonce values. All servers share pairwise MAC keys (clients do not create or verify MACs). Each server provides MACs to the client in the prepare phase, which the client sends in the commit phase to prove that enough servers successfully completed prepare requests at a given timestamp. Of course, a faulty server may provide faulty MACs during the prepare phase, or it may reject valid MACs during the commit phase. To recover from this, a client may need to gather more MACs from the prepare phase after it has entered the commit phase, but a commit eventually completes.

The replication-based protocol also uses signatures to allow servers to prove to a reader that a block was written by some client; that is, to prevent a server from returning fabricated data. Instead of signatures, each server provides a pseudo-random nonce value in the prepare phase of our protocol. The client aggregates these values and provides them to each server in the commit phase. During a read operation, a server provides the client with these nonce values to prove that some client invoked a write at a specific timestamp, as will be described in Section 3.3 and Lemma 5 of Section 3.4.2.

Early write: Before committing a write, a correct server in our protocol must ensure that enough other correct servers have fragments for this write, such that a reader will be able to reconstruct the block. The replication-based protocol does not face this problem because each server stores the entire block. Our protocol could solve this problem with another round of communication for servers to confirm receipt of a fragment. Instead, in our protocol and unlike previous protocols, clients send erasure-coded fragments in the first round of the prepare phase, which saves a round of communication. With this approach, a faulty client may send a fragment in the first phase without committing the write. As in PASIS [2], a server may limit this by rejecting a write from a client with too many uncommitted writes.

Partial encoding: PASIS encodes and hashes fragments for all n servers. Encoding this many fragments is wasteful because f servers may not be involved in a write operation. Instead, our protocol encodes and hashes fragments for only the first $n-f$ servers, which lowers the computational overhead. We call this partial encoding because we only partially encode the block for most write operations. The computational savings are significant: for $m=f+1$, our protocol encodes only $2f+1$ fragments, which is the same number encoded by a non-Byzantine fault-tolerant erasure-coded protocol. Many m-of-n erasure codes encode the first m frag-

ments by dividing a block into m fragments (such codes are said to be systematic), which takes little if any computation. Hence, encoding $2f+1$ fragments requires computing $f+1$ values, whereas encoding $4f+1$ fragments (as in PASIS) requires computing $3f+1$ values.

The drawback of this approach is that if one of the first $m+f$ servers is non-responsive or faulty, the client may need to send the entire block to convince another server that its fragment corresponds to the checksum. This procedure is expensive: not only does it consume extra bandwidth, but the server must verify the block against the checksum. To verify a block, the server encodes the block into $m+f$ fragments, hashes each fragment, and compares these hashes to the checksum provided by the client. If the hashes match the checksum, the server encodes its fragment from the block. Fortunately, the first $m+f$ servers should rarely be non-responsive or faulty.

Distributed verification of erasure-coded data: One problem in PASIS is that each server knows only the cross-checksum and its fragment, and so it is difficult for a server to verify that its fragment together with the corresponding fragments held by other servers form a valid erasure coding of a unique block. We recently solved this problem [20]. The solution relies on a data structure called a *fingerprinted cross-checksum*. The fingerprinted cross-checksum includes a cross-checksum, as used in PASIS, along with a set of *homomorphic fingerprints* of the first m fragments of the block. The fingerprints are homomorphic in that the fingerprint of the erasure coding of a set of fragments is equal to the erasure coding of the fingerprints of those fragments. The overhead of computing homomorphic fingerprints is small compared to the cryptographic hashing for the cross-checksum.

The i^{th} fragment is said to be consistent with a fingerprinted cross-checksum if its hash matches the i^{th} index in the cross-checksum and its fingerprint matches the i^{th} erasure coding of the homomorphic fingerprints. Thus, a server can determine if a fragment is consistent with a fingerprinted cross-checksum without access to any other fragments. Furthermore, any two blocks decoded from any two sets of m fragments that are consistent with the fingerprinted cross-checksum are identical with all but negligible probability [20]. A server can check that a fragment is consistent with a fingerprinted cross-checksum shared by other servers on commit, allowing it to overwrite old fragments. Thus, only fragments that are in the process of being written must be versioned, obviating the need for on-disk versioning. This technique also eliminates most of the computational expense of validating the cross-checksum during a read operation.

3.1.2 Protocol overview

This section provides an overview of the protocol. The protocol provides wait-free [21] writes and obstruction-free [22] reads of constant-sized blocks while tolerating a fixed number of Byzantine servers and an arbitrary number of Byzantine clients in an asynchronous environment. Figure 1 provides an outline of the pseudo-code for both write operations and read operations. The line numbers in Figure 1 match those of Figures 2 and 3. Figure 2 provides detailed pseudo-code for a write operation and is described line-by-line in Section 3.3.2. Figure 3 provides detailed pseudo-code for a read operation and is described line-by-line in Section 3.3.3.

To write a block, a client encodes the block into $m+f$ fragments, computes the fingerprinted cross-checksum, and sends each server its fragment and the fingerprinted cross-checksum (lines 100–206). The server responds with a logical timestamp, a nonce, and a MAC for each server of the timestamp, fingerprinted cross-checksum, and nonce (line 505). If timestamps do not match, the client requests new MACs at the greatest timestamp found (line 214). Unlike the signatures in the Liskov and Rodrigues protocol, the client cannot tell if these MACs are valid.

```
c_write(B):
100:  d_1,...,d_{m+f} ← encode_{1,...,m+f}(B)              /* Partial encoding */
101:  for (i ∈ {1,...,m+f}) do fpcc.cc[i] ← hash(d_i)
102:  for (i ∈ {1,...,m}) do fpcc.fp[i] ← fingerprint(hash(fpcc.cc), d_i)
...:  for (i ∈ {1,...,m+f}) do
206:    Prepare[i] ← S_i.s_rpc_prepare_frag(d_i, ts, fpcc)
505:    /* Server returns ⟨ts, nonce, ⟨MAC_{i,j}(⟨ts, fpcc, nonce⟩)⟩_{1≤j≤n}⟩ */
214:    if (Prepare[i].ts ≠ ts) then                       /* Retry prepare */
...:      ...                                              /* See lines 209-217 */
...:  for (i ∈ {1,...,m+f}) do
228:    S_i.s_rpc_commit(ts, fpcc, Prepare)

c_read():
...:  for (i ∈ {1,...,2f+1}) do
705:    ⟨ts, fpcc⟩ ← S_i.s_rpc_find_timestamp()
...:  for (i ∈ {1,...,m}) do
720:    d_i ← S_i.s_rpc_read(ts, fpcc)
917:  B ← decode(d_1,...,d_m)
...:  /* Client verifies the consistency of block B in c_find_block */
726:  return B
```

Figure 1: Pseudo-code outline. Line numbers match Figures 2 and 3.

The client then commits this write by sending the timestamp, fingerprinted cross-checksum, nonces, and MACs to each server (line 228). A correct server may reject a commit with MACs from faulty servers or a faulty server may reject a commit with MACs from correct servers. Faults should be uncommon, but when they occur, the client must contact another server. The client can either try the commit at another server or it can send the entire block to another server in order to garner another prepare response. A write operation returns after at most three rounds of communication with correct servers. Because faults and concurrency are rare, the timestamps received in the first round of prepare will often match, which allows most write operations to complete in only two rounds.

To read a block, a client requests timestamps and fingerprinted cross-checksums from $2f+1$ servers (line 705) and fragments from the first m servers (line 720). If the m fragments are consistent with the most recent fingerprinted cross-checksum, and if the client can determine that some client invoked a write at this timestamp (using nonces as described in Section 3.3.3), a block is decoded (line 917) and returned (line 726). (In Figure 3, the client verifies read responses and decodes a block in **c_find_block**.) Most read operations return after one round of communication with correct servers. If a concurrent write causes a fragment to be overwritten, however, the client may be redirected to a later version of the block, as described in Section 3.3.3.

3.2 System model

The point-to-point communication channel between each client and server is authenticated and in-order, which can be achieved in practice with little overhead. Communication channels are reliable but asynchronous, i.e., each message sent is eventually received, but there is no bound assumed on message transmission delays. Reliability is assumed for presentational convenience only; the protocol can be adapted to unreliable channels as discussed by Martin et al. [27, Section 4.3].

Up to f servers and an arbitrary number of clients are Byzantine faulty, behaving in an arbitrary manner. An adversary can coordinate all faulty servers and clients. To bound the amount of storage used to stage fragments for in-progress writes, there is a fixed upper bound on the number of clients in the system and on the number of prepare requests from each client that are not followed by subsequent commits.

We assume that there is a negligible probability that a MAC can be forged or that a hash collision or preimage can be found. The fingerprinted cross-checksum requires that the hash function acts as a random oracle [6]. All servers share pairwise MAC keys. The value labeled nonce must not be disclosed to parties except as prescribed by the protocol, in order to prove Lemma 5 in Section 3.4.2. This value is small and can be encrypted at little cost.

The protocol tolerates f Byzantine faulty servers and any number of faulty clients given an m-of-n erasure code and $n = m + 2f$ servers, where $m \geq f + 1$. As in PASIS, the protocol may be deployed with $m > f + 1$ to achieve higher bandwidth for fixed f.

3.3 Detailed pseudo-code

This section provides detailed pseudo-code for write and read operations. Pseudo-code for a write operation is described line-by-line in Section 3.3.2. Pseudo-code for a read operation is described line-by-line in Section 3.3.3. Presentation simplicity of the pseudo-code is chosen over optimizations that may be found in an actual implementation.

3.3.1 Notation

The protocol relies on concurrent requests that are described in the pseudo-code by remote procedure calls and coroutines. The **cobegin** and parallel bars represent the forking of parallel threads of execution. Such threads stop at **end cobegin**. The main thread continues to execute after forking threads; that is, the main thread does not wait to join forked threads at **end cobegin**. Threads are not preempted until they invoke a remote procedure call or wait on a semaphore. Semaphores are binary and default to zero. A WAIT operation waits on a semaphore, and SIGNAL releases all waiting threads. A return statement halts all threads and returns a value.

Each operation is assigned a logical timestamp, represented by the pair $\langle \mathsf{ts}, \mathsf{fpcc} \rangle$. Timestamps are ordered according to the value of the integer ts or, if they share the same ts, by comparison of the binary value fpcc. Timestamps that share the same ts and fpcc are equal. The most recent commit at a server is represented as latest_commit; this value is initialized to $\langle 0, \text{NULL} \rangle$ before the protocol starts. Each server stages fragments for concurrent writes and stores committed fragments; both staged and committed fragments are kept in the store table.

A block is represented as B and a fragment as d in the pseudo-code. A cross-checksum, abbreviated to cc in the pseudo-code, contains n hashes, $\mathsf{cc}[1], \ldots, \mathsf{cc}[n]$. The fingerprinted cross-checksum, abbreviated to fpcc, contains m fingerprints, $\mathsf{fpcc.fp}[1], \ldots, \mathsf{fpcc.fp}[m]$, and $m + f$ hashes, $\mathsf{fpcc.cc}[1], \ldots, \mathsf{fpcc.cc}[m+f]$.

The $\mathsf{encode}_j(\mathsf{B})$ function encodes block B into its j^{th} erasure-coded fragment. The $\mathsf{decode}(\ldots)$ function will decode the first m fragments provided in its arguments, and the index of each fragment is passed implicitly. We abbreviate $\mathsf{encode}_j(\mathsf{decode}(\mathsf{d}_{i_1}, \ldots, \mathsf{d}_{i_m}))$ to $\mathsf{encode}_j(\mathsf{d}_{i_1}, \ldots, \mathsf{d}_{i_m})$. The $\mathsf{fingerprint}(\mathsf{h}, \mathsf{d})$ function fingerprints fragment d given random value h. The random value in our protocol is provided by a hash of the cross-checksum, which is secure so long as the hash function acts as a random oracle [6]. The homomorphism of the fingerprints provides the following property [20]: if $\mathsf{d}_1, \ldots, \mathsf{d}_n \leftarrow \mathsf{encode}_{1,\ldots,n}(\mathsf{B})$ and $\mathsf{fp}_i \leftarrow \mathsf{fingerprint}(\mathsf{h}, \mathsf{d}_i)$, then $\mathsf{fp}_{i_0} = \mathsf{encode}_{i_0}(\mathsf{fp}_{i_1}, \ldots, \mathsf{fp}_{i_m})$ for any set of indices $i_0, \ldots, i_m$.

3.3.2 Write

Pseudo-code for write is provided in Figure 2. A write operation is invoked with a block of data as its argument and returns SUCCESS as its response. Write is divided into prepare and commit phases. The pseudo-code breaks write into a wrapper function, **c_write**, and a main function, **c_dowrite**, such that the main function can be reused for writing back fragments during a read operation. The wrapper function encodes the block into $m + f$ fragments and computes a fingerprinted cross-checksum (lines 100–102). It then calls **c_dowrite** (line 103).

Prepare, lines 200–221: The prepare phase is described in the top half of **c_dowrite**. The client invokes **s_rpc_prepare_frag** at each of the first $m + f$ servers with its fragment and the fingerprinted cross-checksum (line 206); ts will be NULL. A correct server S_i verifies that this fragment is consistent with the fingerprinted cross-checksum. To do so, it first computes the fingerprint and the hash of the fragment (line 300). It then computes the i^{th} erasure coding of the homomorphic fingerprints in the fingerprinted cross-checksum (line 301). Finally, it ensures that this erasure coding is equal to the fingerprint of this fragment, and that the hash is equal to the i^{th} hash in the cross-checksum (line 302). If the fragment is consistent, the server prepares a response in **s_prepare_common** (line 303).

Because of partial encoding, there are only $m + f$ erasure-coded fragments, so if one of the first $m + f$ servers is not responsive or if commit fails, the client may need to invoke **s_rpc_prepare_block** with the entire block (line 207). A correct server S_i verifies that the erasure coding of the block contains at least m fragments that are consistent with the fingerprinted cross-checksum. To do so, it encodes the block into each of $m + f$ fragments (line 403), computes the fingerprint and the hash of each fragment (line 404), and computes the appropriate erasure coding of the homomorphic fingerprints in the fingerprinted cross-checksum (line 405). It counts the number of fragments for which the fingerprint is equal to the erasure coding of the homomorphic fingerprints and the hash is equal to the appropriate hash in the fingerprinted cross-checksum (line 406). If there are at least m such consistent fragments, server S_i computes the i^{th} fragment d_i and the rest of the cross-checksum of all n fragments and prepares a response in **s_prepare_common** (lines 407–409).

Invoking **s_rpc_prepare_block** allows a client to write a fragment that is not consistent with the fingerprinted cross-checksum, so long as this fragment can be erasure-coded from a block with m erasure-coded fragments that are consistent with the fingerprinted cross-checksum. This will be useful in the read protocol to ensure that any fragment can be written back as needed.

The response prepared in **s_prepare_common** consists of a ts, a nonce, and n MACs (one for each server). The ts may be provided by the client; if not, it is assigned to one greater than the ts portion of the logical timestamp used in the most recent commit (line 500). The nonce is a pseudo-random value that is unique for each timestamp (line 501); we use a MAC of the timestamp to ensure this property. An array of n MACs is computed with the shared pairwise MAC keys (line 505). The MACs are used in the commit phase to authenticate the timestamp and nonce, as well as to prove that enough correct servers stored consistent fragments.

If this timestamp is more recent than the most recently committed timestamp (line 502), the nonce_hash, a preimage-resistant hash of the nonce, is computed (line 503), and the fragment and nonce_hash are stored for future reads (line 504). If **s_prepare_common** was called by **s_rpc_prepare_block**, the correct cross-checksum of all n fragments is also stored. (The NULL value on line 504 is a placeholder that will be filled in

```
c_write(B):                                          /* Wrapper function for c_dowrite */
100: d_1,...,d_{m+f} ← encode_{1,...,m+f}(B)
101: for (i ∈ {1,...,m+f}) do fpcc.cc[i] ← hash(d_i)
102: for (i ∈ {1,...,m}) do fpcc.fp[i] ← fingerprint(hash(fpcc.cc), d_i)
103: c_dowrite(B, NULL, fpcc)

c_dowrite(B, ts, fpcc):                              /* Do a write */
200: Prepare[*] ← NULL
201: cobegin
202: ||_{i∈{1,...,n}}                                /* Start worker threads */
203:    /* Send fragment and fpcc to server, get MAC of fpcc and latest ts */
204:    d_i ← encode_i(B)
205:    if (fpcc.cc[i] = hash(d_i)) then
206:       Prepare[i] ← S_i.s_rpc_prepare_frag(d_i, ts, fpcc)
207:    else Prepare[i] ← S_i.s_rpc_prepare_block(B, ts, fpcc)  /* Bad fpcc.cc[i] */
208:
209:    /* Choose latest ts, if needed */
210:    if (ts = NULL ∧ |{j : Prepare[j] ≠ NULL}| = 2f+1) then
211:       ts ← max{ts' : ⟨ts', *⟩ ∈ Prepare}
212:       SIGNAL(found_largest_ts)
213:    if (ts = NULL) then WAIT(found_largest_ts)       /* Wait for chosen ts */
214:    if (Prepare[i].ts ≠ ts) then                     /* Need new prepare for chosen ts */
215:       if (fpcc.cc[i] = hash(d_i)) then
216:          Prepare[i] ← S_i.s_rpc_prepare_frag(d_i, ts, fpcc)
217:       else Prepare[i] ← S_i.s_rpc_prepare_block(B, ts, fpcc)
218:
219:    /* Attempt commit */
220:    if (|{j : Prepare[j].ts = ts}| ≥ m+f) then SIGNAL(prepare_ready)
221: end cobegin
222:
223: UnwrittenSet ← {1,...,n}
224: while (TRUE) do
225:    WAIT(prepare_ready)
226:    cobegin
227:    ||_{i∈UnwrittenSet}                           /* Start worker threads */
228:       if (SUCCESS = S_i.s_rpc_commit(ts, fpcc, Prepare)) then
229:          UnwrittenSet ← UnwrittenSet \ {i}
230:       if (|UnwrittenSet| ≤ f) then return SUCCESS
231:    end cobegin

S_i.s_rpc_prepare_frag(d, ts, fpcc):                 /* Grant permission to write fpcc at ts */
300: fp ← fingerprint(hash(fpcc.cc), d);   h ← hash(d)
301: fp' ← encode_i(fpcc.fp[1],...,fpcc.fp[m])
302: if (fp = fp' ∧ h = fpcc.cc[i]) then             /* Fragment is consistent with fpcc */
303:    return S_i.s_prepare_common(ts, fpcc, d, NULL)
304: else return FAILURE                             /* Faulty client */

S_i.s_rpc_prepare_block(B, ts, fpcc):                /* Grant permission to write fpcc at ts */
400: /* Called when partial encoding fails or when writing back fragments */
401: cnt ← 0
402: for (j ∈ {1,...,m+f}) do                        /* Validate block */
403:    d_j ← encode_j(B)
404:    fp ← fingerprint(hash(fpcc.cc), d_j);   cc[j] ← hash(d_j)
405:    fp' ← encode_j(fpcc.fp[1],...,fpcc.fp[m])
406:    if (fp = fp' ∧ cc[j] = fpcc.cc[j]) then cnt ← cnt+1
407: if (cnt ≥ m) then                               /* Found m fragments consistent with fpcc */
408:    for (j ∈ {m+f+1,...,n}) do cc[j] ← hash(encode_j(B))
409:    return S_i.s_prepare_common(ts, fpcc, encode_i(B), cc)
410: else return FAILURE                             /* Faulty client */

S_i.s_prepare_common(ts, fpcc, d, cc):               /* Create the prepare response */
500: if (ts = NULL) then ts ← latest_commit.ts+1
501: nonce ← MAC_{i,i}(⟨ts, fpcc⟩)
502: if (⟨ts, fpcc⟩ > latest_commit) then
503:    nonce_hash ← hash(nonce)
504:    store[⟨ts, fpcc⟩] ← ⟨d, cc, nonce_hash, NULL⟩
505: return ⟨ts, nonce, ⟨MAC_{i,j}(⟨ts, fpcc, nonce⟩)⟩_{1≤j≤n}⟩

S_i.s_rpc_commit(ts, fpcc, Prepare):                 /* Commit write of fpcc at ts */
600: if (⟨ts, fpcc⟩ ≤ latest_commit) then return SUCCESS   /* Overwritten */
601: Nonces ← {⟨j, nonce⟩ : Prepare[j] = ⟨ts, nonce, ⟨tag_k⟩_{1≤k≤n}⟩ ∧
602:                        tag_i = MAC_{j,i}(⟨ts, fpcc, nonce⟩)}
603: if (|Nonces| ≥ m+f) then
604:    ⟨d, cc, nonce_hash, *⟩ ← store[⟨ts, fpcc⟩]
605:    store[⟨ts, fpcc⟩] ← ⟨d, cc, nonce_hash, Nonces⟩
606:    for (⟨ts', fpcc'⟩ < ⟨ts, fpcc⟩) do store[⟨ts', fpcc'⟩] ← NULL
607:    latest_commit ← ⟨ts, fpcc⟩
608:    return SUCCESS
609: else return FAILURE
```

Figure 2: Detailed write pseudo-code.

s_rpc_commit.) The nonces and nonce_hashes are used to prove that a client invoked a write at this timestamp. Other protocols ensure this property in ways that would require more communication [8], public-key signatures [26], or $4f+1$ servers [18]; we use nonces to avoid these mechanisms.

The client must wait for $2f+1$ responses before assigning a timestamp to this write. (The first $2f$ threads wait on line 213 until a timestamp is assigned.) The timestamp is the pair $\langle \mathsf{ts}, \mathsf{fpcc} \rangle$, where ts is the greatest ts value from the $2f+1$ responses. If a server provides a response with a different timestamp, the client must retry that request (lines 214–217).

Commit, lines 223–231: After $m+f$ servers have provided MACs in responses with matching timestamps, commit may be attempted (line 220). The commit may fail if a faulty server provided one of these MACs or rejects a MAC from a correct server. But, eventually, at least $m+f$ correct servers will return responses with MACs that will be accepted by at least $m+f$ servers in commit. As prepare responses arrive, they are forwarded to all servers (line 228). Thus, the threads from the prepare phase do not stop until all servers return responses or the commit phase completes. The commit phase completes and the client can return once $m+f$ servers (all but f) return SUCCESS (line 230).

If a write has a lower timestamp than a previously committed write, a server can ignore it (line 600). A correct server aggregates nonces from valid prepare responses (lines 601–602). A prepare response is valid if the MAC included for this server is a MAC of the timestamp and nonce computed with the proper pairwise key. If there are at least $m+f$ nonces from valid prepare requests, then at least m correct servers stored a fragment, so commit will succeed. The NULL value from line 504 is filled in with these nonces (lines 604–605). This will become the new most recent write (line 607). If a client tries to read a fragment with a lower timestamp, it can be redirected to this write, so earlier fragments can be garbage collected (line 606). Hence, a server must stage fragments for concurrent writes but store only the most recently committed fragments.

3.3.3 Read

Pseudo-code for a read operation is provided in Figure 3. A read operation is invoked with no arguments and returns a block as its response. A read operation is divided into two phases, "find timestamps" and "read timestamp." The client searches for the timestamps of the most recently committed write at each server. As timestamps arrive, the client tries reading at any timestamp greater than or equal to $2f+1$ other timestamps.

Find timestamps, lines 700–715: The client queries each of the first $3f+1$ servers for the timestamp of its most recently committed write (lines 702–707). As timestamps arrive, the client tries reading at any timestamp that it has yet to try already and that is greater than or equal to $2f+1$ other timestamps (lines 710–714). If no writes have been committed, the value latest_commit (line 800) defaults to $\langle 0, \text{NULL} \rangle$; if $2f+1$ or more servers return $\langle 0, \text{NULL} \rangle$, no writes have returned yet so a NULL block is returned (line 715).

Read timestamp, lines 717–732: To read a fragment, the client invokes **s_rpc_read** at each server (line 720). A correct server returns the fragment along with the other data stored during write (line 1002). The client processes each response from **s_rpc_read** with the helper function **c_find_block** (line 721). This function verifies that the fragment is valid (lines 900–906), determines whether a client invoked this write (lines 910–913), and decodes a

```
c_read():                                                    /* Read a block */
700: Timestamp[*] ← NULL;   State[*] ← ∅
701:
702: /* Search for write timestamps */
703: cobegin
704: ||_{i∈{1,...,3f+1}}                                      /* Start worker threads */
705:    Timestamp[i] ← S_i.s_rpc_find_timestamp()
706:    SIGNAL(found_timestamp)
707: end cobegin
708:
709: while (TRUE) do
710:    WAIT(found_timestamp)
711:    /* Try any timestamp greater or equal to m+f timestamps */
712:    for (⟨ts, fpcc⟩ : ⟨ts, fpcc⟩ = Timestamp[i] ∧ ⟨ts, fpcc⟩ ∉ Tried ∧
713:                      |{j : ⟨ts, fpcc⟩ ≥ Timestamp[j]}| ≥ 2f+1) do
714:       Tried ← Tried ∪ {⟨ts, fpcc⟩}
715:       if (ts = 0) then return NULL                        /* No writes yet */
716:
717:       cobegin
718:       ||_{i∈{1,...,n}}                                    /* Start worker threads */
719:          ⟨ts, fpcc⟩ ← ⟨ts, fpcc⟩                          /* Thread local copy of variables */
720:          ⟨data, gc_redirect⟩ ← S_i.s_rpc_read(ts, fpcc)
721:          B ← c_find_block(i, State, ts, fpcc, data)
722:
723:          /* Write back fragments as needed and return the block */
724:          if (B ≠ NULL) then
725:             c_dowrite(B, ts, fpcc)
726:             return B
727:
728:          /* Follow garbage collection redirection */
729:          if (gc_redirect ≠ NULL ∧ gc_redirect > ⟨ts, fpcc⟩) then
730:             Timestamp[i] ← gc_redirect
731:             SIGNAL(found_timestamp)
732:       end cobegin

S_i.s_rpc_find_timestamp():                                  /* Return the latest commit */
800: return latest_commit
```

```
c_find_block(i, State, ts, fpcc, ⟨d, cc, nonce_hash, Nonces⟩):   /* Classify read */
900: if (d ≠ NULL ∧ cc = NULL) then     /* Verify fragment-encoded arguments */
901:    fp ← fingerprint(hash(fpcc.cc), d)
902:    fp' ← encode_i(fpcc.fp[1], ..., fpcc.fp[m])
903:    if (fp ≠ fp' ∨ hash(d) ≠ fpcc.cc[i]) then return NULL
904:
905: if (d ≠ NULL ∧ cc ≠ NULL) then        /* Verify block-encoded arguments */
906:    if (hash(d) ≠ cc[i]) then return NULL
907:
908: /* Update state and count preimages */
909: State[⟨ts, fpcc⟩] ← State[⟨ts, fpcc⟩] ∪ {⟨i, d, cc, nonce_hash, Nonces⟩}
910: npreimages ← |{j : ⟨j, *, *, nonce_hash', *⟩ ∈ State[⟨ts, fpcc⟩] ∧
911:                ⟨*, *, *, *, {*, ⟨j, nonce'⟩, *}⟩ ∈ State[⟨ts, fpcc⟩] ∧
912:                nonce_hash' = hash(nonce')}|
913: if (npreimages < f+1) then return NULL
914:
915: /* Try to decode */
916: Frags ← {d' ≠ NULL : ⟨*, d', NULL, *, *⟩ ∈ State[⟨ts, fpcc⟩]}
917: if (|Frags| ≥ m) then return decode(Frags)
918: else for (cc' ≠ NULL : ⟨*, *, cc', *, *⟩ ∈ State[⟨ts, fpcc⟩]}) do
919:    Frags' ← {d' ≠ NULL : ⟨*, d', cc', *, *⟩ ∈ State[⟨ts, fpcc⟩]}
920:    if (|Frags ∪ Frags'| ≥ m) then
921:       cnt ← 0;   B ← decode(Frags ∪ Frags')
922:       for (j ∈ {1, ..., m+f}) do                          /* Validate block */
923:          d_j ← encode_j(B)
924:          fp ← fingerprint(hash(fpcc.cc), d_j);   h ← hash(d_j)
925:          fp' ← encode_j(fpcc.fp[1], ..., fpcc.fp[m])
926:          if (fp = fp' ∧ h = fpcc.cc[j]) then cnt ← cnt+1
927:       if (cnt ≥ m) then                  /* Found m fragments consistent with fpcc */
928:          return B
929:
930: return NULL                                               /* No block found */

S_i.s_rpc_read(ts, fpcc):                                    /* Read the fragment at ⟨ts, fpcc⟩ */
1000: if (store[⟨ts, fpcc⟩] = NULL ∧ latest_commit > ⟨ts, fpcc⟩) then
1001:    return ⟨⟨NULL, NULL, NULL, NULL⟩, latest_commit⟩
1002: else return ⟨store[⟨ts, fpcc⟩], NULL⟩
```

Figure 3: Detailed read pseudo-code.

block if possible (lines 915–928). If a correct server has no record of this fragment but knows of a more recent write, it returns the timestamp of the more recent write (line 1001). The client follows such garbage collection redirections if a block is not found (lines 728–731).

If a correct server received a fragment in a successful call to **s_rpc_prepare_frag**, the value cc will be set to NULL and the client will verify that the fragment is consistent with the fingerprinted cross-checksum (lines 901–903). If a correct server received a fragment in a successful call to **s_rpc_prepare_block**, the value cc will be the cross-checksum of all n fragments. The client verifies that the cross-checksum cc matches at least this fragment (line 906). If either verification fails, the response is ignored (lines 903 and 906). Otherwise, the client records this fragment in the state for this timestamp (line 909) and tries to determine whether a client invoked this write (lines 910–913). If there are at least $f+1$ nonces that are the preimages of nonce_hashes, one was generated by a correct server, which implies that a client invoked a write with this timestamp (i.e., the write was not fabricated by faulty servers) and so it is eligible to be examined further. Otherwise, the client waits for more nonces before trying to decode a block.

If enough nonces are found, the client tries to reconstruct the block. A block can always be decoded given m fragments consistent with the fingerprinted cross-checksum (line 917). If any fragments were provided with an additional cross-checksum value cc, the client can reconstruct a block and check if the erasure-coding of that block includes m fragments consistent with the fingerprinted cross-checksum. Since all correct servers will produce the same cross-checksum in **s_rpc_prepare_block**, it suffices to check each value of cc in turn (line 918). If m fragments were returned with the same value cc or are consistent with the fingerprinted cross-checksum, the client decodes a block (line 921). If at least m fragments in the erasure-coding of this block are consistent with the fingerprinted cross-checksum (lines 922-927), this block will be returned. Note that the check in **c_find_block** (lines 922-927) is identical to that in **s_rpc_prepare_block** (lines 402–407).

If a block is found, it is returned as the response of the read (line 726). To ensure that this block is seen by subsequent reads, the client writes back fragments as needed (line 725). In practice, the client can skip write back if any $2f+1$ of the first $3f+1$ servers claim to have committed this timestamp or a more recent one.

3.4 Correctness

This section provides arguments for the safety and liveness properties of the protocol.

3.4.1 Liveness

In this section, we argue the liveness properties of write and read operations. We consider two notions of liveness, namely *wait freedom* [21] and *obstruction freedom* [22]. Informally, an operation is wait-free if the invoking client can drive the operation to completion in a finite number of steps, irrespective of the behavior of other clients. An operation is obstruction-free if the invoking client can drive the operation to completion in a finite number of steps once all other clients are inactive for sufficiently long. That is, an obstruction-free operation may not complete, but only due to continual interference by other clients.

THEOREM 1. Write operations are wait-free.

PROOF. **c_dowrite** invokes **s_rpc_prepare_frag** (line 206) or **s_rpc_prepare_block** (line 207) at each server. Each such call at a correct server returns successfully (line 303 or 409), implying

that the client receives at least $n - f \geq 2f + 1$ responses. Consequently, if ts as input to **c_dowrite** is NULL then the largest ts returned by servers is chosen (line 211) and *found_largest_ts* is signalled (line 212). An **s_rpc_prepare_frag** (line 216) or **s_rpc_prepare_block** (line 217) call is then placed at each server that did not return this ts. If ts as input to **c_dowrite** is not NULL, all correct servers will return this ts. By the time the last of the threads that will reach line 220 does so (if not sooner), all correct servers have contributed a response for the same timestamp to *Prepare* from **s_rpc_prepare_frag** or **s_rpc_prepare_block**, causing *prepare_ready* to be signalled. The collected set of prepare responses in *Prepare* is then sent to all servers in an **s_rpc_commit** (line 228). Because at least $m + f$ of the prepare responses in *Prepare* are from correct servers, at least $m + f$ of the prepare responses contain correct MAC values (line 601) and so *Nonces* will include at least $m + f$ tuples (line 603). Hence, these **s_rpc_commit** calls to correct servers return SUCCESS (line 608), and so the write operation completes (line 230). □

DEFINITION 2. If S_i.**s_rpc_commit**(ts, fpcc, *Prepare*) returns SUCCESS, then this commit at S_i is said to *rely on* S_j if $Prepare[j] = \langle \mathsf{ts}, \mathsf{nonce}, \langle \mathsf{tag}_k \rangle_{1 \leq k \leq n} \rangle$ and $\mathsf{tag}_i = \mathsf{MAC}_{j,i}(\langle \mathsf{ts}, \mathsf{fpcc}, \mathsf{nonce} \rangle)$.

LEMMA 3. In a correct client's **c_read**, suppose that for a fixed $\langle \mathsf{ts}, \mathsf{fpcc} \rangle$ the following occurs: From some correct S_i that previously returned SUCCESS to **s_rpc_commit**(ts, fpcc, $*$), and from each of m correct servers S_j on which the first such commit at S_i relies, the client receives $\langle \mathsf{data}, * \rangle$ in response to an **s_rpc_read**(ts, fpcc) call on that server (line 720) where $\mathsf{data} \neq \langle$NULL, NULL, NULL, NULL$\rangle$. Then, the call to **c_find_block** (line 721) including the last such response returns a block $\mathsf{B} \neq$ NULL.

PROOF. Consider such a $\mathsf{data} = \langle \mathsf{d}, \mathsf{cc}, \mathsf{nonce\_hash}, Nonces \rangle$ received from a correct server S_j. d either is consistent with fpcc as verified by S_j in **s_rpc_prepare_frag** (lines 300–302) and verified by the client in **c_find_block** (lines 900–903), or cc matches this fragment, as generated by S_j in **s_rpc_prepare_block** (lines 404 and 408) and verified by the client in **c_find_block** (line 906). In the latter case, the fact that S_j reached line 504 (where it saved $\langle \mathsf{d}, \mathsf{cc}, \mathsf{nonce\_hash}, * \rangle$) implies that previously $\mathsf{cnt} \geq m$ in line 407, and so by the properties of fingerprinted cross-checksums [20, Theorem 3.4], all such servers received the same input block B in calls **s_rpc_prepare_block**(B, ts, fpcc) and so constructed the same cc in **s_rpc_prepare_block**.

Now consider the response $\mathsf{data} = \langle \mathsf{d}, \mathsf{cc}, \mathsf{nonce\_hash}, Nonces \rangle$ from the correct server S_i. Recall that S_i previously returned SUCCESS to **s_rpc_commit**(ts, fpcc, $*$), and that the first such commit relied on the m correct servers S_j. Since we focus on the first such commit at S_i, and since $\mathsf{data} \neq \langle$NULL, NULL, NULL, NULL$\rangle$, the SUCCESS response was generated in line 608, not 600. In this case, *Nonces* $\neq$ NULL by lines 603 and 605, and in fact includes $\langle j, \mathsf{nonce} \rangle$ pairs for the m correct servers S_j on which this commit relies. When the last data from S_i and these m servers S_j is passed to **c_find_block** (line 721), each nonce_hash present in each S_j's data will have a matching nonce in S_i's *Nonces*, i.e., such that $\mathsf{nonce\_hash} = \mathsf{hash}(\mathsf{nonce})$. Hence, $\mathsf{npreimages} \geq m \geq f + 1$ (lines 910–913), and so the client will try to decode a block in lines 915–928.

If the responses from the m servers S_j on which the commit at S_i relies have $\mathsf{cc} =$ NULL, a block is decoded and returned (line 917). Otherwise, the client eventually tries to decode these fragments accompanied by $\mathsf{cc} =$ NULL (the set *Frags*) together with those accompanied by $\mathsf{cc} \neq$ NULL provided by these correct servers S_j (the set *Frags*′); see line 921. Each S_j contributing a fragment of the latter type verified that fpcc was consistent with m fragments derived from the block B input to **s_rpc_prepare_block** (lines 402–407), and generated its fragment to be a valid fragment of B. Each fragment of the former type was verified by S_j to be consistent with fpcc (lines 300–302), and so is a valid fragment of B (with overwhelming probability [20, Corollary 2.12]). Consequently, upon decoding any m of these fragments, the client obtains B, will find m fragments of the resulting block to be consistent with fpcc (lines 922–927), and so will return B (line 928). □

THEOREM 4. The read protocol is obstruction-free.

PROOF. In **c_read**, a call to **s_rpc_find_timestamp** is made to servers $1, \ldots, 3f + 1$ (line 705), to which each of at least $2f + 1$ correct servers responds with its value of latest_commit (line 800). Consider the greatest timestamp $\langle \mathsf{ts}, \mathsf{fpcc} \rangle$ returned by a correct server, say S_i. This timestamp is greater than or equal to the timestamp from at least the $2f + 1$ correct servers that responded (checked in lines 712–713), so the client tries to read fragments via **s_rpc_read** at this timestamp (line 720). This timestamp was previously committed by S_i, as a correct server updates latest_commit only in **s_rpc_commit** at line 607. Moreover, this commit relies on at least m correct servers S_j; see line 603. Now consider the following two possibilities for each of these correct servers S_j on which the commit relies:

- S_j assigned to $\mathsf{store}[\langle \mathsf{ts}, \mathsf{fpcc} \rangle]$ in line 504 because the condition in line 502 evaluated to true, and has not subsequently deleted $\mathsf{store}[\langle \mathsf{ts}, \mathsf{fpcc} \rangle]$ in line 606. In this case, S_j returns the contents of $\mathsf{store}[\langle \mathsf{ts}, \mathsf{fpcc} \rangle]$ in response to the **s_rpc_read** call (line 1002).
- S_j either did not assign to $\mathsf{store}[\langle$ts, fpcc$\rangle]$ in line 504 because the condition in line 502 evaluated to false, or deleted $\mathsf{store}[\langle \mathsf{ts}, \mathsf{fpcc} \rangle]$ in line 606. In this case, $\mathsf{store}[\langle \mathsf{ts}, \mathsf{fpcc} \rangle] =$ NULL and $\mathsf{latest\_commit} > \langle \mathsf{ts}, \mathsf{fpcc} \rangle$ in line 1000 (due to lines 502 and 607), and so S_j returns latest_commit in response to the **s_rpc_read** call (line 1001).

If all m correct servers S_j fall into the first case above, then one of the client's calls to **c_find_block** (line 721) returns a non-NULL block (Lemma 3). The client writes this with a **c_dowrite** call (line 725), which is wait-free (Theorem 1), and then completes the **c_read**. If some S_j falls into the second case above, then the timestamp it returns (or a higher one returned by another correct server) satisfies the condition in lines 712–713 and so the client will subsequently read at this timestamp (line 720) if a non-NULL block is not first returned from **c_find_block** (line 721).

Consequently, for the client to never return a block in a **c_read**, correct servers must continuously return increasing timestamps in response to **s_rpc_read** calls. If there are no concurrent commits, then the client must reach a timestamp at which it returns a block. Hence, the read protocol is obstruction-free. □

3.4.2 Linearizability

Informally, linearizability [23] requires that the responses to read operations are consistent with an execution of all reads and writes in which each operation is performed at a distinct moment in real time between when it is invoked and when it completes. We need only be concerned with reads by correct clients, because we provide no guarantees to faulty clients. Since writes by faulty clients can be read by correct clients, however, we cannot ignore such writes. Consequently, we define the execution of **s_rpc_prepare_frag** or **s_rpc_prepare_block** by a faulty client at a correct server that returns $\langle \mathsf{ts}, \mathsf{nonce}, * \rangle$ (i.e., returns a value on line 505 rather than re-

turning FAILURE on line 304 or 410) to instantiate a write invocation at the beginning of time. The timestamp of the invocation is the pair $\langle \mathsf{ts}, \mathsf{fpcc} \rangle$ used to generate a nonce (line 501). Each operation by a correct client also gets an associated timestamp $\langle \mathsf{ts}, \mathsf{fpcc} \rangle$. For a write operation, the timestamp is that sent to **s_rpc_commit**. For a read operation, the timestamp is that of the write operation from which it read.

Proving that faulty write operations and correct write and read operations are linearizable shows that the protocol guarantees a natural extension to linearizability, limiting faulty clients to invoking writes that they could have invoked anyway at similar expense had they followed the protocol. The following five lemmas are used to prove that such a history is linearizable.

LEMMA 5. A read will share a timestamp with a write that has been invoked by some client.

PROOF. Per line 913, a call to **c_find_block**($*$, *State*, ts, fpcc, $*$) returns a non-NULL value only if *State*[$\langle$ts, fpcc$\rangle$], possibly modified per line 909, includes a nonce_hash from some correct S_j such that some S_i.**s_rpc_read**(ts, fpcc) returned data $= \langle *, *, *, \mathit{Nonces} \rangle$, $\mathit{Nonces} \ni \langle j, \mathsf{nonce} \rangle$ and hash(nonce) = nonce_hash (data was passed to this or a previous **c_find_block**($*$, *State*, ts, fpcc, $*$) call, see lines 720–721, and then added to *State*[$\langle$ts, fpcc$\rangle$] in line 909). This nonce was created by S_j on line 501 with $\langle$ts, fpcc$\rangle$. Since a correct writer keeps each nonce secret unless it is returned from **s_rpc_prepare_frag** or **s_rpc_prepare_block** for the timestamp on which it settles for its write timestamp, this nonce shows that the writer, if correct, adopted $\langle$ts, fpcc$\rangle$ as its timestamp; consequently, the write with this timestamp was invoked, satisfying the lemma. If no correct writer performed a write with timestamp $\langle$ts, fpcc$\rangle$, then the creation of nonce by S_j in line 501 with $\langle$ts, fpcc$\rangle$ implies that the write with timestamp $\langle \mathsf{ts}, \mathsf{fpcc} \rangle$ was invoked by a faulty client. □

LEMMA 6. Consider two invocations

$$\mathsf{B} \leftarrow \textbf{c\_find\_block}(*, *, \mathsf{ts}, \mathsf{fpcc}, *)$$
$$\mathsf{B}' \leftarrow \textbf{c\_find\_block}(*, *, \mathsf{ts}, \mathsf{fpcc}, *)$$

at correct clients for the same timestamp $\langle \mathsf{ts}, \mathsf{fpcc} \rangle$. If $\mathsf{B} \neq$ NULL and $\mathsf{B}' \neq$ NULL, then $\mathsf{B} = \mathsf{B}'$ with all but negligible probability.

PROOF. A block $\mathsf{B} \neq$ NULL is returned by **c_find_block**($*$, $*$, ts, fpcc, $*$) at either line 917 or line 928. B is returned at line 928 only if at least m erasure-coded fragments produced from B are consistent with fpcc, as checked in lines 922–927. Similarly, B is returned at line 917 only after it is reconstructed from at least m fragments d' such that $\langle *, \mathsf{d}', \mathsf{cc}, *, * \rangle \in \mathit{State}[\langle$ts, fpcc$\rangle]$ and $\mathsf{cc} =$ NULL; each such d' was confirmed to be consistent with fpcc in lines 900–903, in either this or an earlier invocation of the form **c_find_block**($*$, $*$, ts, fpcc, $*$). In either case, B has at least m erasure-coded fragments consistent with fpcc. If blocks B and B' each have at least m erasure-coded fragments that are consistent with the same fpcc, they are the same with all but negligible probability [20, Theorem 3.4]. □

Lemma 6 states that two correct clients who read blocks at the same timestamp read the same block, since the block returned from **c_read** is that produced by **c_find_block** (lines 721–726).

Lemmas 7–9 show that timestamp order for operations is consistent with real-time precedence.

LEMMA 7. Consider two write operations performed by correct clients. If the response to one precedes the invocation of the other, then the timestamp of the former is less than the timestamp of the latter.

PROOF. Before the earlier write returns a response in line 230, at least $n - f$ servers returned SUCCESS from **s_rpc_commit**(ts, fpcc, *Prepare*), where $\langle$ts, fpcc$\rangle$ is the timestamp of this write. In doing so, at least $m = n - 2f$ correct servers record latest_commit $\leftarrow \langle$ts, fpcc$\rangle$ (line 607) if not greater (line 600). Consequently, the ts value returned in the prepare phase for the later write by these servers will be greater than the ts value for the earlier write (line 500). Because there are $n = m + 2f$ total servers, any $2f + 1$ servers will include one of these m correct servers, so the ts chosen (line 211) will be greater than the ts value in the timestamp for the earlier write. □

LEMMA 8. Consider a write operation and a read operation, both performed by correct clients. If the response to the write precedes the invocation of the read, then the timestamp of the write is at most the timestamp of the read.

PROOF. Before the write returns a response in line 230, at least $n - 2f$ correct servers returned SUCCESS from **s_rpc_commit**(ts, fpcc, *Prepare*), where $\langle$ts, fpcc$\rangle$ is the timestamp of this write. In doing so, at least $n - 2f$ correct servers record latest_commit $\leftarrow$ $\langle$ts, fpcc$\rangle$ (line 607) if not greater (line 600). These $n - 2f = m$ correct servers include at least $f + 1$ of the servers $1, \ldots, 3f + 1$, and so in the read operation at most $2f$ of servers $1, \ldots, 3f + 1$ respond to **s_rpc_find_timestamp** at line 800 with a lower timestamp than $\langle$ts, fpcc$\rangle$. Because a read considers only timestamps that are at least as large as those returned by $2f + 1$ of the first $3f + 1$ servers (line 713), the read will be assigned a timestamp at least as large as $\langle$ts, fpcc$\rangle$. □

LEMMA 9. If the response of a read operation by a correct client precedes the invocation of another (write or read) operation by a correct client, then the timestamp of the former operation is at most the timestamp of the latter.

PROOF. A read calls **c_dowrite** at its timestamp before returning a response (line 725). This will have the same affect as completing a write at that timestamp. Consequently, the later operation will have a higher timestamp if it is a write (Lemma 7) and a timestamp at least as high if it is a read (Lemma 8). □

THEOREM 10. Write and read operations are linearizable.

PROOF. To show linearizability, we construct a linearization (total order) of all read operations by correct clients and all write operations that is consistent with the real-time precedence between operations such that each read operation returns the block written by the preceding write operation. We first order all writes in increasing order of their timestamps. By Lemma 7, this ordering does not violate real-time precedence. We then place all read operations with the same timestamp immediately following a write operation with that timestamp, ordered consistently with real-time precedence (i.e., each read is placed somewhere after all other operations with the same timestamp that completed before it was invoked). By Lemma 5, each read is placed after some write operation. This placement does not violate real-time precedence with the next (or any) write operation in the linearization, since if the next write had completed before this read began, then by Lemma 8 this read operation could not have the timestamp it does. Real-time precedence between reads with different timestamps cannot be violated by this placement, by Lemma 9. Since all reads with the same timestamp read the same block (Lemma 6)—which is the block written in the write with that timestamp if the writer was correct—write and read operations are linearizable. □

3.5 Protocol enhancements

This section briefly discusses how to extend the protocol to ensure non-skipping timestamps [5] and to allow read-only clients.

Non-skipping timestamps: A faulty client or server could suggest an arbitrarily large value for the ts portion of the timestamp. To prevent this, the protocol can ensure that timestamps do not skip. When a client calls prepare at a server with a NULL ts value, the server returns a ts value (as before) but also a $\mathsf{ts}_{\mathsf{prepare}}$ value. The $\mathsf{ts}_{\mathsf{prepare}}$ is the ts value of the greatest successful prepare at this server. The server also returns n MACs of the $\mathsf{ts}_{\mathsf{prepare}}$ value (one for each server). A server will accept a call to prepare with a non-NULL ts only if that ts is accompanied by valid MACs for $f+1$ $\mathsf{ts}_{\mathsf{prepare}}$ values that are greater than or equal to the requested ts value. Hence, the ts value will advance only if at least some correct server successfully prepared at this timestamp.

A client can retry prepare with a non-NULL timestamp once it has a ts value that is greater than or equal to $2f+1$ ts values (as before) but less than or equal to $f+1$ $\mathsf{ts}_{\mathsf{prepare}}$ values. Because the greatest ts value returned by a correct server must be less than or equal to the $\mathsf{ts}_{\mathsf{prepare}}$ values returned by at least $f+1$ correct servers, a client will eventually get an array of $f+1$ valid MACs for $\mathsf{ts}_{\mathsf{prepare}}$ values that will allow it to successfully complete the prepare phase. During a read operation, servers return $\mathsf{ts}_{\mathsf{prepare}}$ values and MACs in **s_rpc_read**. If a correct server committed the ts at which the client is trying to read, the client will eventually find at least $f+1$ greater or equal $\mathsf{ts}_{\mathsf{prepare}}$ values from correct servers that can be used to write fragments back in **c_dowrite** as needed.

Read-only clients: Because readers must be able to write back fragments as needed, the pseudo-code in Section 3.3 does not allow clients with only read permissions. Read-only access control can be enforced by preventing clients without write permission from issuing a prepare request with a NULL ts value and enforcing non-skipping timestamps as described above. A client without write permission can still write back fragments because $\mathsf{ts}_{\mathsf{prepare}}$ values and MACs are returned in **s_rpc_read**.

4. IMPLEMENTATION

We evaluate the protocol and compare it to competing approaches using a distributed storage prototype. The low-overhead fault-tolerant prototype (named Loft) consists of a client library, linked to directly by client applications, and a storage server application. The prototype supports the protocol described in Section 3 as well as several competing protocols, as described in Section 5.1.

The client library interface consists of two functions, "read block" and "write block." In addition to the parameters described in Section 3, read and write accept a block number as an additional argument, which can be thought of as running an instance of the protocol in parallel for every block in the system. Each server in a pool of storage servers runs the storage server application, which accepts incoming RPC requests and executes as described in Section 3. Clients and servers communicate with remote procedure calls over TCP sockets. Each server has a large NVRAM cache, where non-volatility is provided by battery backup. This allows most writes and many reads to return without disk I/O.

The prototype uses 16 byte fingerprints generated with the evaluation homomorphic fingerprinting function [20, Section 5]. Fingerprinting is fast, and only the first m fragments are fingerprinted (the other fingerprints can be computed from these fingerprints). Homomorphic fingerprinting requires a small random value for each distinct block that is fingerprinted. This random value is provided by a hash of the cross-checksum in the protocol. After computing this random value, our implementation precomputes a 64 kB table, which takes about 20 microseconds. After computing this table, fingerprinting each byte requires one table lookup and a 128-bit XOR. This implementation can fingerprint about 410 megabytes per second on a 3 GHz Pentium D processor.

The client library uses Rabin's Information Dispersal Algorithm [31] for erasure coding. The first m fragments consist of the block divided into m equal fragments (that is, we use a systematic encoding). Since $m > f$ and only $m+f$ fragments must be encoded, this cuts the amount of encoding by more than half.

We use the SHA-1 and HMAC implementations from the Nettle toolkit [28], which can hash about 280 megabytes per second on a 3 GHz Pentium D processor. Each hash value is 20 bytes, and each MAC is 8 bytes. Due to ongoing advances in the cryptanalysis of SHA-1 [14], a storage system with a long expected lifetime may benefit from a stronger hash function. The performance of SHA-512 on modern 64-bit processors has been reported as comparable to that of SHA-1 [16]. Hence, though we did not measure this, we expect that the prototype would achieve similar performance if the hash were upgraded on such systems.

The prototype implements a few simple optimizations for the protocol. For example, during commit, only the appropriate pairwise MAC is sent to each server. The client tries writing at the first $m+f$ servers, considering other servers only if these servers are faulty or unresponsive. Similarly, the client requests timestamps from the first $2f+1$ servers and reads fragments from the first m servers, which allows most reads to return in a single round of communication without requiring any decoding beyond concatenating fragments. Furthermore, if $f+1$ or more servers return matching timestamps, the client does not request nonces because it can conclude that a correct server committed this write and hence some client invoked a write at this timestamp, satisfying Lemma 5.

Also, the client limits the amount of state required for a read operation by considering only the most recent timestamp proposed by each server. It does not consider more timestamps until all but f servers have returned a response for all timestamps currently under consideration. Servers delay garbage collection by a few seconds, thus obviating the need for fast clients to ever follow garbage collection redirection.

5. EVALUATION

This section evaluates the protocol on our distributed storage prototype. We also implemented and evaluated three competing protocols that are described in Section 5.1. The experimental setup is described in Section 5.2. Single client write throughput, read throughput, and response time are evaluated in Sections 5.3, 5.4, and 5.5, respectively. An analysis of the protocol presented in Section 3 suggests that our protocol should perform similarly to a benign erasure-coded protocol with the additional computational expense of hashing and the extra bandwidth required for the fpcc, MACs, and nonces. The experimental results confirm that our protocol is competitive with the benign erasure-coded protocol and that it significantly outperforms competing approaches.

5.1 Competing protocols

To enable fair comparison, our distributed storage prototype supports multiple protocols. We compare protocols within the same framework to ensure that measurements reflect protocol variations rather than implementation artifacts. We evaluate the following three protocols in addition to our protocol.

Benign erasure-coded protocol: The prototype implements an erasure-coded storage protocol that tolerates crashes but not Byzantine faulty clients or servers. This protocol uses the same erasure coding implementation used for our protocol. A write operation

encodes a block into $m+f$ fragments and sends these fragments to servers. A read operation reads from the first m servers, avoiding the need to decode. Both complete in a single round of communication. This protocol assumes concurrency is handled by some external locking protocol; a real implementation would require more rounds of communication for some writes, but we ignore this overhead. Hence, this protocol provides an upper bound for the performance of any erasure-coded fault-tolerant storage protocol (Byzantine or not).

Benign replication-based protocol: The prototype implements a replication-based storage protocol that tolerates crashes but not Byzantine faulty clients or servers. A write operation sends the block to $f+1$ servers, and a read operation reads from a single server. As in the benign erasure-coded protocol, this protocol assumes concurrency is handled by an external locking protocol. Hence, though this protocol does not tolerate Byzantine faults, it represents an upper bound for the performance of any replication-based Byzantine (or not) fault-tolerant storage protocols. It is worth noting, however, that this implementation is substantially faster than most replication-based Byzantine fault-tolerant storage protocols found in the literature, which often require all-to-all broadcasts [8, 10], public-key signatures [9, 26], or writing to $2f+1$ or more replicas [8, 10].

Replication-based protocols are excellent for reads, but write performance is reduced due to bandwidth limitations. One method to overcome the network bandwidth limitation between the client and the switch for a single client is to use network-level multicast. Our replication-based protocol does not use multicast for several reasons. Multicast is unavailable, unsuitable, or unstable in many network environments [18], and retransmissions due to congestion cannot take advantage of multicast. Also, multicast does nothing to reduce disk bandwidth and network bandwidth between the switch and the servers. Though each server could encode its own block to reduce disk bandwidth [8], a multicast-based protocol would not scale when multiple clients are writing to the same storage servers.

m+3f Byzantine fault-tolerant erasure-coded protocol: An alternative to Byzantine fault-tolerant replication-based storage is Byzantine fault-tolerant erasure-coded storage. The prototype implements a protocol similar to PASIS [18]. PASIS was engineered to improve server throughput by offloading work to clients. This protocol uses the same erasure coding implementation and SHA-1 library used for our protocol. The protocol implemented by the prototype, however, only emulates a PASIS-like protocol. It does not implement the versioning storage required by PASIS, nor does it run a garbage collection protocol, and, hence, it would not be suitable for storing data in a Byzantine environment. This implementation does, however, provide a comparison point against the approach most similar to our protocol.

To write a block, the client requests the most recent timestamp from each server. It then encodes the block into $m+3f$ fragments and hashes each fragment to create a cross-checksum. (Because we use a systematic encoding, "encoding" the first f fragments does not require computation.) By comparison, our protocol encodes and hashes $m+f$ fragments; f fewer because there are f fewer servers and another f fewer due to the partial encoding optimization, described in Section 3.1.1. To complete the write, the $m+3f$ protocol sends fragments to the first $m+2f$ servers, considering other servers only if some servers are unresponsive. By comparison, our protocol and the benign erasure-coded protocol send f fewer fragments because they need f fewer servers. To read a block, the client requests fragments from the first m servers along with timestamps from the first $3f+1$ servers. Assuming all timestamps match, the client must then verify the cross-checksum, which is embedded in the timestamp. This requires repeating the write computation: the client must encode and hash $m+3f$ fragments to recompute the cross-checksum.

5.2 Experimental setup

All experiments are measured using a single client and a collection of servers. Each machine has a dual-core 3 GHz Pentium D processor, 2 GB of RAM, and an Intel PRO/1000 Gigabit Ethernet controller, and machines run Linux kernel version 2.6.18. Measurements are taken in the absence of concurrency and faults, which is expected to be the normal mode of operation in such a storage system. The client and the servers are connected to the same HP ProCurve Switch 2848 with QoS passthrough mode set to one-queue and flow control enabled for each port. Each experiment was run 10 times for 60 seconds, with the average performance reported in the figures. Standard deviations are all within 2% of the average, and performance matches analytical expectations.

The working set of data for each experiment is chosen to fit within the server caches, and the client does not cache data. The data gets loaded before measurements, ensuring 100% read hits, and the servers use write-back with synchronizing to disk disabled. The systems are battery-backed, but the experimental reason for this setup is to allow the measurement of protocol overhead rather than disk latency. Avoiding disk accesses makes performance dependent on the network and computational behavior of the protocols. If the working set does not fit in server caches, or if durability requirements prevent using NVRAM for write-back, the choice of protocol matters less because system performance will be limited by disk performance. In such a scenario, there is an even stronger argument for using a Byzantine fault-tolerant protocol rather than a protocol that tolerates only crash faults.

The client benchmark program is run on a single machine. It generates a synthetic workload. For throughput measurements, the client spawns several parallel threads, each of which issues a read or write request for a randomly selected 64 kB block, waits for the response, and then issues another request. For response time measurements, a single thread issues a single write request, waits for a response, and then repeats. For erasure-coded protocols (all but replication), $m=f+1$. Because the block size is fixed, the fragment size for erasure-coded protocols decreases as m increases (i.e., fragment size is $64/m$ kB).

5.3 Write throughput

Figure 4 shows the write throughput achieved by a single client executing each of the four protocols as a function of the number of faults tolerated. Our protocol significantly outperforms the Byzantine fault-tolerant $m+3f$ erasure-coded protocol as well as the crash fault-tolerant replication-based protocol, and it nearly matches the performance of the benign erasure-coded protocol. For example, at $f=4$, our protocol achieves a factor of 2.6 higher throughput than replication, a factor of 1.4 higher throughput than the $m+3f$ protocol, and is within 5% of the performance of the erasure-coded protocol that does not tolerate Byzantine faulty servers or clients.

Each protocol requires a different number of servers to tolerate the same number of faults. The benign erasure-coded protocol and our protocol require $m+f$ responsive servers, the replication-based protocol requires $f+1$ servers, and the $m+3f$ protocol requires $m+2f$ responsive servers. (Both Byzantine fault-tolerant protocols must be able to reach an additional f servers if some of these servers are not responsive.) For example, for $f=4$ and $m=5$, the benign erasure-coded protocol and our protocol write data to 9

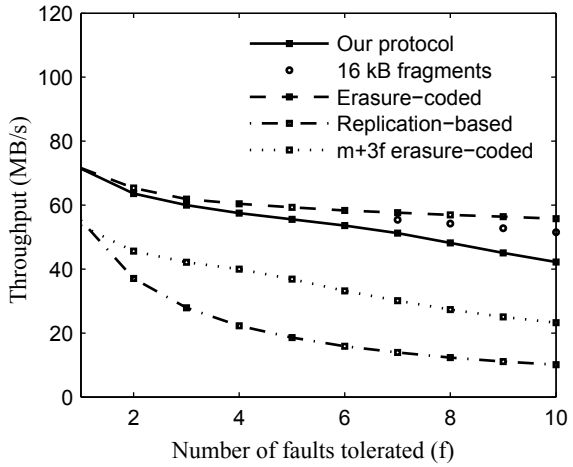

Figure 4: Write throughput for each protocol as a function of faults tolerated. The lines report the performance of each protocol when writing 64 kB blocks. The four circles report the performance of our protocol when writing 16 kB fragments (rather than $64/m$ kB).

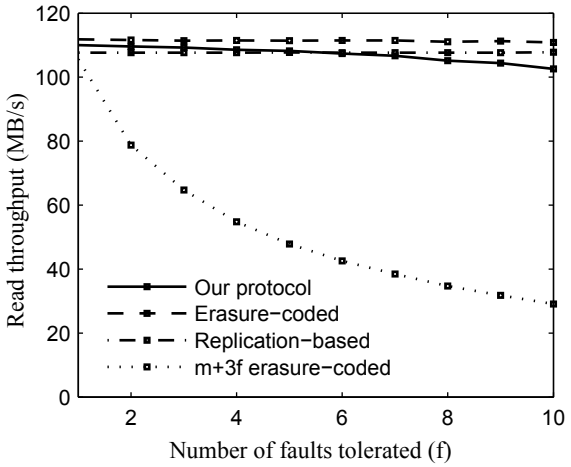

Figure 5: Read throughput as a function of faults tolerated.

servers, the replication-based protocol writes to 5 servers, and the $m+3f$ protocol writes to 13 servers.

Throughput is the amount of useful data written, which is less than the amount of data sent over the network. Our protocol and the benign erasure-coded protocol both send $\frac{|B|}{m}(m+f)$ bytes when writing a block of $|B|$ bytes. Replication must send $|B|(f+1)$ bytes, and the $m+3f$ protocol must send $\frac{|B|}{m}(m+2f)$ bytes. The erasure-coded protocols could increase throughput for constant f by increasing m beyond $f+1$.

The benign erasure-coded protocol performs well, as expected, achieving a write throughput close to $\frac{m}{m+f}$ of the total network bandwidth available. Our protocol performs almost as well. When tolerating up to 6 Byzantine faulty servers, it performs within 10% of the benign protocol that only tolerates server crashes. As the number of servers in the system grows, however, the additional network overhead in our protocol becomes noticeable for two reasons. First, because block size is constant, the size of the fragment written at each server decreases as the number of servers increases. Second, the sizes of the fpcc, MACs, and nonces increase as the number of servers increases. For $f=6$, fragment size is over 9 kB and fpcc, MAC, and nonce overhead is under 700 bytes (overhead is under 7% of data sent). For $f=10$, fragment size is under 6 kB and fpcc, MAC, and nonce overhead is over 1100 bytes (overhead is over 15% of data sent).

One solution to this problem is to increase the block size. For example, the four circles in Figure 4 show the throughput of our protocol when fragment size is 16 kB. Our protocol performs within 10% of the benign protocol when the fragment size is increased to 16 kB for both protocols, even when tolerating 10 faults. (The benign protocol performs less than 3% better when the fragment size is increased to 16 kB.)

The replication-based protocol performs poorly for all but the smallest number of faults tolerated, as expected, because it writes $(f+1)/(\frac{m+f}{m}) > (f+1)/2$ times as much data as the benign erasure-coded protocol. The $m+3f$ Byzantine fault-tolerant erasure-coded protocol writes $\frac{m+2f}{m+f} \approx 1.5$ times as much data as the benign erasure-coded protocol, and up until about $f=4$ it is only a factor of 1.5 times worse. The $m+3f$ protocol, however, must encode and hash $m+3f$ fragments to generate the cross-checksum even though it only writes to $m+2f$ servers because it does not include our partial encoding optimization. Hence, for $f>4$, the $m+3f$ protocol is computationally bound by the client.

5.4 Read throughput

Figure 5 shows the read throughput achieved by a single client executing each of the four protocols as a function of the number of faults tolerated. Our protocol achieves read throughput within 10% of the two benign protocols and significantly outperforms the $m+3f$ protocol. The slight drop for our protocol and the benign erasure-coded protocol as the number of faults tolerated increases is due to network congestion caused by the increasing number of servers providing responses.

All four protocols read the same amount of data. The replication-based protocol reads an entire 64 kB block from a single server, and the other protocols read fragments from m servers. The erasure-coded protocols read from the first m servers to avoid the need to decode. In addition to fragments, the Byzantine fault-tolerant protocols must read timestamps from more servers to check for concurrency. Our protocol reads timestamps from $2f+1-m=f$ more servers, while the $m+3f$ protocol reads timestamps from $3f+1-m=2f$ more servers. Assuming all timestamps match, read completes in a single round of communication.

Once fragments are read, the Byzantine fault-tolerant protocols must verify data. Our protocol requires just a hash and a fingerprint of the fragments. The $m+3f$ protocol, however, must recompute the cross-checksum, which requires encoding and hashing $m+3f$ fragments and is quite expensive for large values of f.

5.5 Response time

Figure 6 shows the response time of a single write for each of the four protocols as a function of the number of faults tolerated. Our protocol requires on average 1.04 ms more to complete a write operation than the benign erasure-coded protocol, which is on average 1.65 times worse. This is, however, a substantial improvement over the $m+3f$ protocol and the replication-based protocol, which both scale worse than our protocol.

Table 1 provides a breakdown of the average latency of each operational component of a write for $f=10$ as seen by the client. The table lists the time each protocol spent encoding, hashing, and fingerprinting fragments (other computational contributions were negligible); it also lists the time spent waiting for the network, which

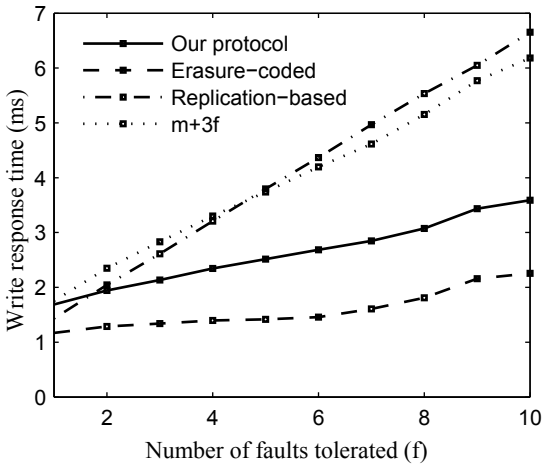

Figure 6: Write response time as a function of faults tolerated.

	RPC	Encode	Hashing	Fingerprinting
Erasure coded	1.46 ms	0.79 ms	–	–
Replication	6.65 ms	–	–	–
Our protocol	2.17 ms	0.79 ms	0.45 ms	0.18 ms
$m+3f$	2.80 ms	2.54 ms	0.88 ms	–

Table 1: Write response time breakdown for $f = 10$.

includes time spent in the kernel. As seen in the table, about half of the additional latency for a write by our protocol as compared to the benign erasure-coded protocol is due to hashing and fingerprinting, and the other half is due to the extra round of communication. The additional latency for the replication-based protocol is, of course, due to the extra bandwidth required to write f replicas. The additional latency for the $m+3f$ protocol is due to the encoding of $2f$ more fragments, the extra round of communication, the sending of 1.5 times as many fragments, and the hashing of $m+3f$ fragments.

Read response time is as expected, so we do not provide a figure. The benign erasure-coded protocol and the replication protocol require on average 0.84 ms and 0.80 ms respectively to read a single block when tolerating between one and ten faults. Our protocol requires on average 1.29 ms, the difference being the time needed to hash and fingerprint fragments. Each of these protocols requires about the same amount of time to read a block when tolerating one fault as when tolerating ten faults. The $m+3f$ protocol requires 3.05 ms on average, and it requires 1.58 ms to read a single block when tolerating one fault but 4.54 ms when tolerating ten faults. It scales worse than the other protocols because it must encode and hash $m+3f$ fragments to recompute the cross-checksum.

6. CONCLUSION

Distributed block storage systems can tolerate Byzantine faults in asynchronous environments with little overhead over systems that tolerate only crashes. Replication-based block storage protocols are effective for workloads that are mostly reads or when tolerating a single fault, but exhibit low throughput and high latency for large writes. Erasure-coded protocols provide higher throughput writes and can increase m for fixed f to realize even higher throughput. Previous Byzantine fault-tolerant erasure-coded protocols, however, exhibit low client throughput for reads and high computational overheads for both reads and writes. This paper presents a Byzantine fault-tolerant erasure-coded protocol that performs well for both reads and large writes. Measurements of a prototype implementation demonstrate that this protocol exhibits throughput within 10% of the ideal crash fault-tolerant erasure-coded protocol for reads and sufficiently large writes. Furthermore, this protocol has little computational overhead other than a cryptographic hash and a homomorphic fingerprint of the data.

7. ACKNOWLEDGMENTS

We thank our shepherd Mike Dahlin and the SOSP reviewers for their feedback as well as the early reviewers of the ideas in this paper, including Michael Abd-El-Malek, Richard Golding, Elie Krevat, Michael Mesnier, Bryan Parno, Raja Sambasivan, Shafeeq Sinnamohideen, Matthew Wachs, Theodore Wong, and Jay Wylie. We thank the members and companies of the CyLab Corporate Partners and the PDL Consortium (including APC, Cisco, EMC, Google, Hewlett-Packard, Hitachi, IBM, Intel, LSI, Network Appliance, Oracle, Panasas, Seagate, and Symantec) for their interest, insights, feedback, and support. This material is based on research sponsored in part by the National Science Foundation, via grants CNS-0326453 and CCF-0424422, and by the Army Research Office, under agreement number DAAD19–02–1–0389. James Hendricks is supported in part by a National Science Foundation Graduate Research Fellowship and was awarded an SOSP student travel scholarship, supported by Infosys, to present this paper at the conference.

8. REFERENCES

[1] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 59–74. ACM Press, 2005.

[2] M. Abd-El-Malek, G. R. Ganger, M. K. Reiter, J. J. Wylie, and G. R. Goodson. Lazy verification in fault-tolerant distributed storage systems. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, pages 179–190. IEEE Computer Society, 2005.

[3] M. Abd-El-Malek, I. William V. Courtright, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa Minor: versatile cluster-based storage. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, pages 59–72. USENIX Association, 2005.

[4] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 109–126. ACM Press, 1995.

[5] R. Bazzi and Y. Ding. Non-skipping timestamps for Byzantine data storage systems. In *Proceedings of the 18th International Symposium on Distributed Computing*, pages 405–419. Springer-Verlag, 2004.

[6] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 62–73. ACM Press, 1993.

[7] J. Bonwick, M. Ahrens, V. Henson, M. Maybee, and M. Shellenbaum. The Zettabyte File System. Technical report, Sun Microsystems.

[8] C. Cachin and S. Tessaro. Asynchronous verifiable information dispersal. In *Proceedings of the 24th IEEE*

Symposium on Reliable Distributed Systems, pages 191–202. IEEE Press, 2005.

[9] C. Cachin and S. Tessaro. Optimal resilience for erasure-coded Byzantine distributed storage. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 115–124. IEEE Computer Society, 2006.

[10] M. Castro and B. Liskov. Authenticated Byzantine fault tolerance without public-key cryptography. Technical Memo MIT-LCS-TM-589, MIT, June 1999.

[11] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, 1994.

[12] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-diagonal parity for double disk failure correction. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 1–14. USENIX Association, 2004.

[13] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: a hybrid quorum protocol for Byzantine fault tolerance. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 177–190. USENIX Association, 2006.

[14] C. De Cannière and C. Rechberger. Finding SHA-1 characteristics: General results and applications. In *Advances in Cryptology – ASIACRYPT*, pages 1–20. Springer-Verlag, 2006.

[15] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 29–43. ACM Press, 2003.

[16] B. Gladman. SHA1, SHA2, HMAC and key derivation in C. http://fp.gladman.plus.com/cryptography_technology/sha.

[17] L. Gong. Securely replicating authentication services. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 85–91. IEEE Computer Society, 1989.

[18] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient Byzantine-tolerant erasure-coded storage. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 135–144. IEEE Computer Society, 2004.

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

[20] J. Hendricks, G. R. Ganger, and M. K. Reiter. Verifying distributed erasure-coded data. In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing*, pages 163–168. ACM Press, 2007.

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

[22] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 522–529. IEEE Computer Society, 2003.

[23] 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, 1990.

[24] R. Kotla and M. Dahlin. High throughput byzantine fault tolerance. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 575–584. IEEE Computer Society, 2004.

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

[26] B. Liskov and R. Rodrigues. Tolerating Byzantine faulty clients in a quorum system. In *Proceedings of the 26th International Conference on Distributed Computing Systems*, pages 34–43. IEEE Computer Society, 2006.

[27] J.-P. Martin, L. Alvisi, and M. Dahlin. Small Byzantine quorum systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 374–388. IEEE Computer Society, 2002.

[28] N. Möller. *Nettle Manual*, 1.15 edition, 2006.

[29] D. Nagle, D. Serenyi, and A. Matthews. The Panasas ActiveScale storage cluster: Delivering scalable high bandwidth storage. In *Proceedings of the ACM/IEEE SC2004 Conference*, page 53. IEEE Computer Society, 2004.

[30] R. Primmer and C. D. Halluin. Collision and preimage resistance of the Centera content address. Technical report, EMC^2 Corporation, 2005.

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

[32] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *SIAM Journal of Applied Mathematics*, 8:300–304, 1960.

[33] R. Rodrigues, P. Kouznetsov, and B. Bhattacharjee. Large-scale Byzantine fault tolerance: Safe but not always live. In *Proceedings of the 3rd Workshop on Hot Topics in System Dependability*. USENIX Association, 2007.

[34] Y. Saito, S. Frølund, A. Veitch, A. Merchant, and S. Spence. FAB: Building distributed enterprise disk arrays from commodity components. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 48–58. ACM Press, 2004.

[35] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata efficiency in versioning file systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 43–58. USENIX Association, 2003.

[36] Sun Microsystems. *ZFS On-Disk Specification Draft*, 2006.

[37] E. Thereska, M. Abd-El-Malek, J. J. Wylie, D. Narayanan, and G. R. Ganger. Informed data distribution selection in a self-predicting storage system. In *Proceedings of the 3rd International Conference on Autonomic Computing*, pages 187–198. IEEE Computer Society, 2006.

[38] H. Weatherspoon and J. D. Kubiatowicz. Erasure coding vs. replication: a quantitative approach. In *International Workshop on Peer-to-Peer Systems*, pages 328–337. Springer-Verlag, 2002.

[39] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, 14(1):108–136, 1996.

[40] Z. Zhang, S. Lin, Q. Lian, and C. Jin. RepStore: A self-managing and self-tuning storage backend with smart bricks. In *Proceedings of the 1st International Conference on Autonomic Computing*, pages 122–129. IEEE Computer Society, 2004.