A Decentralized Algorithm for Erasure-Coded Virtual Disks

Svend Frolund, Arif Merchant, Yasushi Saito, Susan Spence, Alistair Veitch
Internet Systems and Storage Laboratory
HP Laboratories Palo Alto
HPL-2004-46
March 22, 2004*

quorum systems, erasure-coding, crash-recovery, storage systems, distributed systems

A Federated Array of Bricks is a scalable distributed storage system composed from inexpensive storage bricks. It achieves high reliability with low cost by using erasure coding across the bricks to maintain data reliability in the face of brick failures. Erasure coding generates n encoded blocks from m data blocks (n > m) and permits the data blocks to be reconstructed from any m of these encoded blocks. We present a new fully decentralized erasure-coding algorithm for an asynchronous distributed system. Our algorithm provides fully linearizable read-write access to erasure-coded data and supports concurrent I/O controllers that may crash and recover. Our algorithm relies on a novel quorum construction where any two quorums intersect in m processes.

* Internal Accession Date Only
To be published in and presented at the International Conference on Dependable Systems and Networks, 28 June-1 July 2004, Florence, Italy
Approved for External Publication

A Decentralized Algorithm for Erasure-Coded Virtual Disks*

Svend Frølund, Arif Merchant, Yasushi Saito, Susan Spence, and Alistair Veitch
Storage Systems Department
HP Labs, Palo Alto, CA 94304

Abstract

A Federated Array of Bricks is a scalable distributed storage system composed from inexpensive storage bricks. It achieves high reliability with low cost by using erasure coding across the bricks to maintain data reliability in the face of brick failures. Erasure coding generates n encoded blocks from m data blocks $(n > m)$ and permits the data blocks to be reconstructed from any m of these encoded blocks. We present a new fully decentralized erasure-coding algorithm for an asynchronous distributed system. Our algorithm provides fully linearizable read-write access to erasure-coded data and supports concurrent I/O controllers that may crash and recover. Our algorithm relies on a novel quorum construction where any two quorums intersect in m processes.

1 Introduction

Distributed disk systems are becoming a popular alternative for building large-scale enterprise stores. They offer two advantages to traditional disk arrays or mainframes. First, they are cheaper because they need not rely on highly customized hardware that cannot take advantage of economies of scale. Second, they can grow smoothly from small to large-scale installations because they are not limited by the capacity of an array or mainframe chassis. On the other hand, these systems face the challenge of offering high reliability and competitive performance without centralized control.

This paper presents a new decentralized coordination algorithm for distributed disk systems using deterministic erasure codes. A deterministic erasure code, such as Reed-Solomon [12] or parity code, is characterized by two parameters, m and n.[1] It divides a logical volume into fixed-size *stripes*, each with m *stripe units* and computes $n-m$ *parity units* for each stripe (stripe units and parity units have the same size). It can then reconstruct the original m stripe units from any m out of the n stripe and parity units. By choosing appropriate values of m, n, and the unit size, users can tune the capacity efficiency (cost), availability, and performance according to their requirements. The flexibility of erasure codes has attracted a high level of attention in both the industrial and research communities [14, 2, 13, 11].

The algorithm introduced in this paper improves the state of the art on many fronts. Existing erasure-coding algorithms either require a central coordinator (as in traditional disk arrays), rely on the ability to detect failures accurately and quickly (a problem in real-world systems), or assume that failures are permanent (any distributed system must be able to handle temporary failures and recovery of it's components).

In contrast, our algorithm is completely decentralized, yet maintains strict linearizability [7, 1] and data consistency for all patterns of crash failures and subsequent recoveries without requiring quick or accurate failure detection. Moreover, it is efficient in the common case and degrades gracefully under failure. We achieve these properties by running voting over a

*This is an extended version of a paper, with the same title, that appears in the Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN)

[1] Reed-Solomon code allows for any combination of m and n, whereas parity code only allows for $m = n-1$ (RAID-5) or $m = n-2$ (RAID-6).

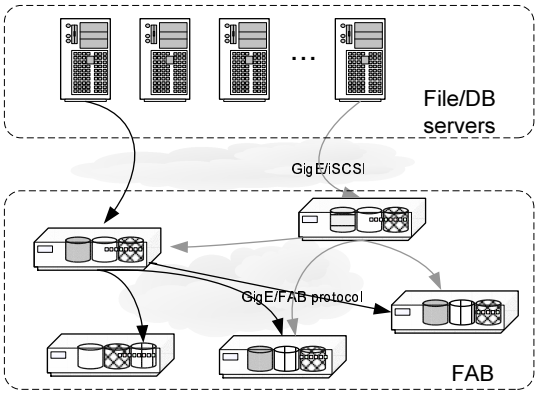

Figure 1: A typical FAB structure. Client computers connect to the FAB bricks using standard protocols. Clients can issue requests to any brick to access any logical volume. The bricks communicate among themselves using the specialized protocol discussed in this paper.

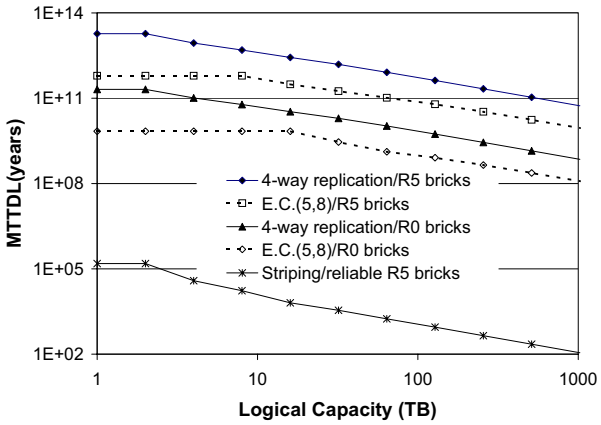

Figure 2: Mean time to first data loss (MTTDL) in storage systems using (1) striping, (2) replication and (3) erasure coding. (1) Data is striped over conventional, high-end, high-reliability arrays, using internal RAID-5 encoding in each array/brick. Reliability is good for small systems, but does not scale well. (2) Data is striped and replicated 4 times over inexpensive, low reliability array bricks. Reliability is highest among the three choices, and scales well. Using internal RAID-5 encoding in each brick improves the MTTDL further over RAID-0 bricks. (3) Data is distributed using 5-of-8 erasure codes over inexpensive bricks. The system scales well, and reliability is almost as high as the 4-way replicated system, using similar bricks.

quorum system which enforces a large-enough intersection between any two quorums to guarantee consistent data decoding and recovery.

In the next two sections, we provide background information on the *FAB* system we have built and quantify the reliability and cost benefits of erasure coding. Section 1.3 articulates the challenge of the coordination of erasure coding in a totally distributed environment and overviews our algorithm. We define the distributed-systems model that our algorithm assumes in Section 2 and outline the guarantees of our algorithm in Section 3. We present our algorithm in Section 4, analyze it in Section 5, and survey related work in Section 6.

1.1 Federated array of bricks

We describe our algorithm in the context of a *Federated Array of Bricks (FAB)*, a distributed storage system composed from inexpensive *bricks* [6]. Bricks are small storage appliances built from commodity components including disks, a CPU, NVRAM, and network cards. Figure 1 shows the structure of a typical FAB system. Bricks are connected together by a standard local-area network, such as Gigabit Ethernet. FAB presents the client with a number of logical volumes, each of which can be accessed as if it were a disk. In order to eliminate central points of failure as well as performance bottlenecks, FAB distributes not only data, but also the coordination of I/O requests. Clients can access logical volumes using a standard disk-access protocol (e.g., iSCSI [8]) via a *coordinator* module running on *any* brick. This decentralized architecture creates the challenge of ensuring single-copy consistency for reads and writes without a central controller. It is this problem that our algorithm solves.

1.2 Why erasure codes?

While any data storage system using large numbers of failure-prone components must use some form of redundancy to provide an adequate degree of reliability, there are several alternatives besides the use of erasure codes. The simplest method for availability is to stripe (distribute) data over conventional, high-

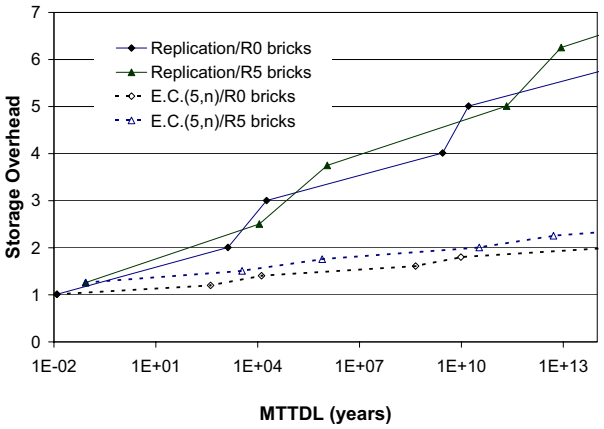

Figure 3: Storage overheads (raw capacity/logical capacity) of systems using replication and erasure coding. The storage overhead of replication-based systems rises much more steeply with increasing reliability requirements than for systems based on erasure-coding. Using RAID-5 bricks reduces the overhead slightly. The MTTDL of a storage system that stripes data over RAID-5 bricks is fixed, and hence this is omitted from this plot; the storage overhead of such a system is 1.25.

reliability array bricks. No redundancy is provided across bricks, but each brick could use an internal redundancy mechanism such as RAID-1 (mirroring) or RAID-5 (parity coding). The second common alternative is to mirror (i.e., replicate) data across multiple bricks, each of which internally uses either RAID-0 (non-redundant striping) or RAID-5. This section compares erasure coding to these methods and show that erasure coding can provide a higher reliability at a lower cost.

Figure 2 shows expected reliability of these schemes. We measure the reliability by the mean time to data loss (MTTDL), which is the expected number of years before data is lost for the first time. For example, in a stripe-based system, data is lost when any one brick breaks terminally. On the other hand, in a system using m out of n erasure coding, a piece of data is lost when more than $n-m$ of n bricks that store the data terminally break at the same time. Thus, the system-wide MTTDL is roughly proportional to the number of combinations of brick failures that can lead to a data loss. We used the component-wise reliability numbers reported in [3] to extrapolate the reliability of bricks and networks, and calculated the MTTDL assuming random data striping across bricks. This graph shows that the reliability of striping is adequate only for small systems. Put another way, to offer acceptable MTTDL in such systems, one needs to use hardware components far more reliable and expensive than the ones commonly offered in the market. On the other hand, 4-way replication and 5-of-8 erasure coding both offer very high reliability, but the latter with a far lower storage overhead. This is because reliability depends primarily on the number of brick failures the system can withstand without data loss. Since both 4-way replication and 5-of-8 erasure coding can withstand at least 3 brick failures, they have similar reliability.

Figure 3 compares the storage overhead (the ratio of raw storage capacity to logical capacity provided) for sample 256TB FAB systems using replication and erasure coding, and with the underlying bricks internally using RAID-5 or RAID-0 (non-redundant). In order to achieve a one million year MTTDL, comparable to that provided by high end conventional disk arrays, the storage overhead for a replication-based system is 4 using RAID-0 bricks and approximately 3.2 using RAID-5 bricks. By contrast, an erasure code based system with $m = 5$ can meet the same MTTDL requirement with a storage overhead of 1.6 with RAID-0 bricks, and yet lower with RAID-5 bricks.

The storage efficiency of erasure-coded systems comes at some cost in performance. As in the case of RAID-5 arrays, small writes (writes to a subregion of the stripe) require a read of the old data and each of the corresponding parity blocks, followed by a write to each. Thus, for an m-of-n erasure coded system, a small write engenders $2(n-m+1)$ disk I/Os, which is expensive. Nonetheless, for read-intensive workloads (such as Web server workloads), systems with large capacity requirements, and systems where cost is a primary consideration, a FAB system based on erasure codes is a good, highly reliable choice.

1.3 Challenges of distributed erasure coding

Implementing erasure-coding in a distributed system, such as FAB, presents new challenges. Erasure-

coding in traditional disk arrays rely on a centralized I/O controller that can accurately detect the failure of any component disk that holds erasure-coded data. This assumption reflects the tight coupling between controllers and storage devices—they reside within the same chassis and communicate via an internal bus.

It is not appropriate to assume accurate failure detection or to require centralized control in FAB. Storage bricks serve as both erasure-coding coordinators (controllers) and storage devices. Controllers and devices communicate via a standard shared, and potentially unreliable, network. Thus, a controller often cannot distinguish between a slow and failed device: the communication latency in such networks is unpredictable, and network partitions may make it temporarily impossible for a brick to communicate with other bricks.

Our algorithm relies on the notion of a quorum system, which allows us to handle both asynchrony and recovery. In our algorithm, correct execution of read and write operations only requires participation by a subset of the bricks in a stripe. A required subset is called a quorum, and for an *m*-out-of-*n* erasure-coding scheme the underlying quorum system must only ensure that any two quorums intersect in at least *m* bricks. In other words, a brick that acts as erasure-coding controller does not need to know which bricks are up or down, it only needs to ensure that a quorum executes the read or write operation in question. Furthermore, consecutive quorums formed by the same controller do not need to contain the same bricks, which allows bricks to seamlessly recover and rejoin the system.

Compared to existing quorum-based replication algorithms [4, 9, 10], our algorithm faces new challenges that are partly due to the fact that we use erasure-coding instead of replication, and partly due to the fact that we apply the algorithm to storage systems. Using erasure-coding instead of replication means that any two quorums must intersect in *m* instead of 1 bricks. We define a new type of quorum system, called an *m-quorum system*, that provides this intersection property. Using erasure-coding also means that it is more difficult to handle partial writes where the erasure-coding controller crashes after updating some, but not all, members of a quorum. Existing quorum-based replication algorithms rely on the ability to write-back the latest copy during a subsequent read operation, essentially having read operations complete the work of a partial write. However, with erasure coding, a partial write may update fewer than *m* stripe units, rendering subsequent read operations unable to reconstruct the stripe. We use a notion of *versioning* in our algorithm so that a read operation can access a previous version of the stripe if the latest version is incomplete. In existing quorum-based algorithms, a read operation *always* tries to complete a partial write that it detects. This means that a partially written value may appear at any point after the failed write operation, whenever a read operation happens to detect it. Having partial write operations take effect at an arbitrary point in the future is not appropriate for storage systems. Our algorithm implements a stronger semantics for partial writes: a partial write appears to either take effect before the crash or not at all. Implementing these stronger semantics is challenging because a read operation must now decide whether to complete or roll-back a partial write that it detects.

2 Model

We use the abstract notion of a *process* to represent a brick, and we consider a set U of n processes, $U = \{p_1, \ldots, p_n\}$. Processes are fully connected by a network and communicate by message passing. The system is asynchronous: there is no bound on the time for message transmission or for a process to execute a step. Processes fail by crashing—they never behave maliciously—but they may recover later. A *correct* process is one that either never crashes or eventually stops crashing. A *faulty* process is a process that is not correct.

Network channels may reorder or drop messages, but they do not (undetectably) corrupt messages. Moreover, network channels have a fair-loss property: a message sent an infinite number of times to a correct process will reach the destination an infinite number of times.

2.1 Erasure-coding primitives

We use the term *block* to refer to the unit of data storage. Processes store data using an *m*-out-of-*n*

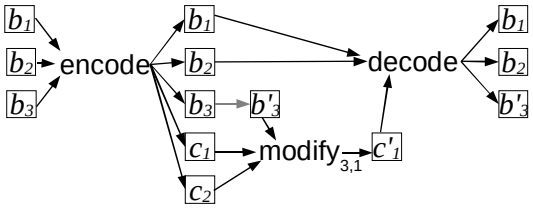

Figure 4: Use of the primitives for a 3-out-of-5 erasure coding scheme. Data blocks b_1 to b_3 form a stripe. The encode function generates two parity blocks c_1 and c_2. When b_3 is updated to become b'_3, we call $\mathsf{modify}_{3,1}(b_3, b'_3, c_1)$ to update c_1 to become c'_1. Finally, we use decode to reconstruct the stripe from b_1, b_2, and c'_1.

erasure-coding scheme. A stripe consists of m data blocks, and we generate $n-m$ parity blocks from these m data blocks. Thus, each stripe results in the storage of n blocks; each process stores one of these n blocks.

The primitive operations for erasure coding are listed in Figure 4:

- encode takes m data blocks and returns n blocks, among which the first m are the original blocks and the remaining $n-m$ are parity blocks. We define encode to return the original data blocks as a matter of notational convenience.
- decode takes any m out of n blocks generated from an invocation of encode and returns the original m data blocks.
- $\mathsf{modify}_{i,j}(b_i, b'_i, c_j)$ re-computes the value of the j'th parity block after the i'th data block is updated. Here, b_i and b'_i are the old and new values for data block i, and c_j is the old value for parity block j.

2.2 m-quorum systems

To ensure data availability, we use a quorum system: each read and write operation requires participation from only a subset of U, which is called a quorum. With m-out-of-n erasure coding, it is necessary that a read and a write quorum intersect in at least m processes. Otherwise, a read operation may not be able to construct the data written by a previous write operation. An m-quorum system is a quorum system where any two quorums intersect in m elements; we refer to a quorum in an m-quorum system as an m-quorum.

Let f be the maximum number of faulty processes in U. An m-quorum system is then defined as follows:

Definition 1 *An m-quorum system $\mathcal{Q} \subseteq 2^U$ is a set satisfying the following properties.*

$$\forall Q_1, Q_2 \in \mathcal{Q} : |Q_1 \cap Q_2| \geq m.$$
$$\forall S \in 2^U \text{ s.t. } |S| = f, \exists Q \in \mathcal{Q} : Q \cap S = \emptyset.$$

The second property ensures the existence of an m-quorum for any combination of f faulty processes. It can be shown that $f = \lfloor (n-m)/2 \rfloor$ is a necessary and sufficient condition for the existence of an m-quorum system (we prove this claim in Appendix A). Thus, we assume that at most $f = \lfloor (n-m)/2 \rfloor$ processes are faulty.

We use a non-blocking primitive called quorum() to capture request-reply style communication with an m-quorum of processes. The quorum(msg) primitive ensures that at least an m-quorum receives msg, and it returns the list of replies. From the properties of an m-quorum system defined above, we can implement quorum() in a non-blocking manner on top of fair-lossy channels by simply retransmitting messages periodically.

2.3 Timestamps

Each process provides a non-blocking operation called newTS that returns a totally ordered timestamp. There are two special timestamps, LowTS and HighTS, such that for any timestamp t generated by newTS, $\mathsf{LowTS} < t < \mathsf{HighTS}$. We assume the following minimum properties from newTS.

UNIQUENESS: Any two invocations of newTS (possibly by different processes) return different timestamps.

MONOTONICITY: Successive invocations of newTS by a process produce monotonically increasing timestamps.

PROGRESS: Assume that newTS() on some process returns t. If another process invokes newTS an infinite number of times, then it will eventually receive a timestamp larger than t.

A logical or real-time clock, combined with the issuer's process ID to break ties, satisfies these properties.

3 Correctness

For each stripe of data, the processes in U collectively emulate the functionality of a read-write register, which we call a *storage register*. As we describe below, a storage register is a special type of atomic read-write register that matches the properties and requirements of storage systems.

A storage register is a strictly linearizable [1] atomic read-write register. Like traditional linearizability [7], strict linearizability ensures that read and write operations execute in a total order, and that each operation logically takes effect instantaneously at some point between its invocation and return. Strict linearizability and traditional linearizability differ in their treatment of partial operations. A partial operation occurs when a process invokes a register, and then crashes before the operation is complete. Traditional linearizability allows a partial operation to take effect at any time after the crash. That is, if a storage brick crashes while executing a write operation, the write operation may update the system at an arbitrary point in the future, possibly after the brick has recovered or has been replaced. Such delayed updates are clearly undesirable in practice—it is very complicated, if not impossible, for the application-level logic that recovers from partial writes to take future updates into account.

Strict linearizability ensures that a partial operation appears to either take effect before the crash or not at all. The guarantee of strict linearizability is given relative to external observers of the system (i.e., applications that issue reads and writes). The only way for an application to determine if a partial write actually took effect is to issue a subsequent read. In our algorithm, the fate of a partial write is in fact decided by the next read operation on the same data: the read rolls the write forward if there are enough blocks left over from the write, otherwise the read rolls back the write.

We allow operations on a storage register to *abort* if they are invoked concurrently. It is extremely rare that applications issue concurrent write-write or read-write operations to the same block of data: concurrency is usually resolved at the application level, for example by means of locking. In fact, in analyzing several real-world I/O traces, we have found no concurrent write-write or read-write accesses to the same block of data [6]. An aborted operation returns a special value (e.g., $\perp$) so that the caller can distinguish between aborted and non-aborted operations. The outcome of an aborted operation is non-deterministic: the operation may have taken effect as if it were a normal, non-aborted operation, or the operation may have no effect at all, as if it had never been invoked. Strict linearizability incorporates a general notion of aborted operations.

In practice, it is important to limit the number of aborted operations. Our algorithm only aborts operations if they actually conflict on the same stripe of data (i.e., write-write or read-write operations), and only if the operations overlap in time or generate timestamps that do not constitute a logical clock. Both situations are rare in practice. First, as we have already observed, it is extremely rare for applications to concurrently issue conflicting operations to the same block of data. Moreover, we can make stripe-level conflicts unlikely by laying out data so that consecutive blocks in a logical volume are mapped to different stripes. Second, modern clock-synchronization algorithms can keep clock skew extremely small [5]. Finally, it is important to notice that the absence of concurrency and the presence of clock synchronization only affect the abort rate, not the consistency of data.

4 Algorithm

Our algorithm implements a single storage register; we can then independently run an instance of this algorithm for each stripe of data in the system. The instances have no shared state and can run in parallel.

In Section 4.1, we give describe the basic principles behind the algorithm and the key challenges that the algorithm solves. Section 4.2 describes the data structures used by the algorithm. Section 4.3 gives the pseudo-code for reading and writing stripes of data, and Section 4.4 gives the pseudo-code for reading and

writing individual blocks within a stripe. We prove the algorithm correct in Appendix B.

4.1 Overview

Our algorithm supports four types of operations: *read-stripe* and *write-stripe* to read and write the entire stripe, and *read-block* and *write-block* to read and write individual blocks within the stripe.[2] A read operation returns a stripe or block value if it executes successfully; a write operation returns OK if it executes successfully. Both read and write operations may abort, in which case they return the special value $\perp$.

A process that invokes a register operation becomes the *coordinator* for that operation. Any process can be the coordinator of any operation. The designation of coordinator is relative to a single operation: consecutive operations on the same data can be coordinated by different processes.

Each process stores a single block for each storage register. To simplify the presentation, we assume that process j always stores block j. That is, processes $p_1 \dots p_m$ store the data blocks, and $p_{m+1} \dots p_n$ store the parity blocks. It is straightforward to adapt the algorithm to more sophisticated data-layout schemes. In the following, we refer to $p_{m+1} \dots p_n$ as the *parity processes*.

To implement a total order for operations, each process stores a timestamp along with each block of data. The timestamp denotes the time when the block was last updated. The basic principle of our algorithm is then for a write coordinator to send a message to an m-quorum of processes to store new block values with a new timestamp. A read coordinator reads the blocks and timestamps from an m-quorum and reconstructs the most recent register value.

A key complexity of the algorithm stems from the handling of a partial write operation, which stores a value in fewer than an m-quorum of replicas, either because the coordinator crashes or proposes too small a timestamp. Such a partial write causes two potential problems: inability to recover the previous value, and violation of strict linearizability.

[2]The single-block methods can easily be extended to access multiple blocks, but we omit this extension to simplify the presentation.

4.1.1 Recovering from partial writes

The challenge with erasure coding is that, during a write operation, a process cannot just overwrite its data block with the new data value. For example, consider an erasure-coded register with $m = 5, n = 7$ (the m-quorum size is 6). If a write coordinator crashes after storing the new value on only 4 processes, we have 4 blocks from the new stripe and 3 blocks from the old, which means that it is impossible to construct either the old or the new stripe value.

To handle such situations, each process keeps a log of $\langle \textit{block-value}, \textit{timestamp} \rangle$ pairs of past write requests. A write request simply appends the new value to the log; a read coordinator collects enough of the most recent blocks from the logs to recover the last register value. We discuss log trimming in Section 5.1.

4.1.2 Linearizing partial operations

After a partial write, a read operation cannot simply pick the value with the highest timestamp, since this may violate strict linearizability. For example, consider the execution in Figure 5. To satisfy strict linearizability, a storage-register implementation must ensure the following total order: $\text{write}_1 \rightarrow \text{read}_2 \rightarrow \text{read}_3$. In other words, read_3 must return v even though it finds the value v' with a higher timestamp. That is, we need to detect partial write operations and abort them to handle such a situation. We accomplish this by executing a write operation in two phases. In the first phase, a write operation informs an m-quorum about the intention to write a value; in the second phase, a write operation actually writes the value to an m-quorum. A read operation can then detect a partial write as an unfulfilled intention to write a value.

Our approach of explicit partial-write detection has a pleasant side effect: an efficient single-round read operation in the common case. A read operation first checks if an m-quorum has no partial write; if so, it simply returns the current register value: the value received from the process that stores the requested data, or the stripe value derived from any m processes in the case of a full stripe read. Failing the optimistic phase, the read operation reconstructs the most recent register value and writes it back to an m-quorum.

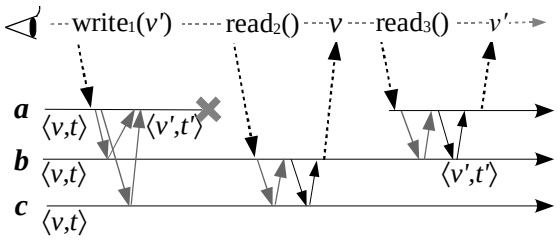

Figure 5: To ensure strict linearizability, read operations cannot simply pick, and possibly write-back, the value with the highest timestamp. In the example, the processes *a*, *b* and *c* implement a storage register; for simplicity, we use an erasure-coding scheme with a stripe size of 1 and where parity blocks are copies of the stripe block (i.e., replication as a special case of erasure coding). The label $\langle v,t\rangle$ indicates that a process stores a value v with timestamp t. The first request $\text{write}_1(v')$ crashes after storing v' on only *a*; the second read_2 request contacts processes *b* and *c* and returns value v. Then *a* recovers, and the subsequent read_3 returns v', even though write_1 seems to have happened before read_2 in the eye of an observer.

4.2 Persistent data structures

Each process has persistent storage that survives crashes. In general, the $\mathsf{store}(var)$ primitive atomically writes the value of variable *var* to the persistent storage. When a process recovers, it automatically recovers the most recently stored value for each variable.

The persistent state of each process consists of a timestamp, *ord-ts*, and a set of timestamp-block pairs, called the *log*. The initial values for *ord-ts* and *log* are LowTS and $\{[\mathsf{LowTS}, \mathsf{nil}]\}$, respectively. (Remember that, for any timestamp t generated by newTS, $\mathsf{LowTS} < t < \mathsf{HighTS}$.) The log captures the history of updates to the register as seen by an individual process. To update the timestamp information in the log without actually storing a new value, we sometimes store a pair $[ts, \perp]$ in the log. We define three functions on the log:

- The "max-ts(*log*)" function returns the highest timestamp in *log*.
- The "max-block(*log*)" function returns the non-$\perp$ value in *log* with the highest timestamp.
- The "max-below(*log, ts*)" function returns the non-$\perp$ value in *log* with the highest timestamp smaller than *ts*.

Variable *ord-ts* shows the logical time at which the most recent write operation was started, establishing its place in the ordering of operations. As such, max-ts(*log*) $<$ *ord-ts* indicates the presence of a partial operation.

4.3 Reading and writing the whole stripe

Algorithm 1 describes the methods for reading and writing a stripe. Algorithm 2 describes the handlers invoked upon receipt of messages from a coordinator.

The write-stripe method triggers a two-phase interaction. In the first phase, the coordinator sends "[Order, *ts*]" messages to replicas with a newly generated timestamp. A replica updates its *ord-ts* and responds OK if it has not already seen a request with a higher timestamp. This establishes a place for the operation in the ordering of operations in the system, and prevents a concurrent write operation with an older timestamp from storing a new value between the first and second phases. In the second round, the coordinator sends "[Write,..]" messages and stores the value.

The read-stripe method first optimistically assumes that an *m*-quorum of processes stores blocks with the same value and timestamp, and that there are no partial writes. If these assumptions are true, the method returns after one round-trip without modifying the persistent state of any process (line 9). Otherwise, the two-phase recovery method is invoked, which works like the write-stripe method except that it dynamically discovers the value to write using the read-prev-stripe method. This method finds the most recent version with at least *m* blocks. Its loop ends when it finds the timestamp of the most recent complete write. The recovery method ensures that the completed read operation appears to happen after the partial write operation and that future read operations will return values consistent with this history.

Algorithm 1 Methods for accessing the entire stripe.

```
 1: procedure read-stripe()
 2:     val ←fast-read-stripe()
 3:     if val = ⊥ then val ←recover()
 4:     return val
 5: procedure fast-read-stripe()
 6:     targets ←Pick m random processes
 7:     replies ←quorum([Read, targets])
 8:     if status in all replies is true
            and val-ts in all replies is the same
            and all processes in targets replied then
 9:         return decode(blocks in replies from targets)
10:     else
11:         return ⊥
12: procedure write-stripe(stripe)
13:     ts ←newTS()
14:     replies ←quorum([Order, ts])
15:     if status in any reply is false then return ⊥
16:     else return store-stripe(stripe, ts)
17: procedure recover()
18:     ts ←newTS()
19:     s ←read-prev-stripe(ts)
20:     if s ≠ ⊥ and store-stripe(s, ts) = OK then
21:         return s
22:     else
23:         return ⊥
24: procedure read-prev-stripe(ts)
25:     max ←HighTS
26:     repeat
27:         replies ←quorum([Order&Read, ALL, max, ts])
28:         if status in any reply is false then
29:             return ⊥
30:         max ←the highest timestamp in replies
31:         blocks ←the blocks in replies with
                timestamp max
32:     until | blocks | ≥ m
33:     return decode(blocks)
34: procedure store-stripe(stripe, ts)
35:     replies ←quorum([Write, encode(stripe), ts])
36:     if status in all replies is true then return OK
37:     else return ⊥
```

Algorithm 2 Register handlers for process p_i

```
38: when receive [Read, targets] from coord
39:     val-ts ←max-ts(log)
40:     status ←val-ts ≥ ord-ts
41:     b ←⊥
42:     if status and i ∈ targets then
43:         b ←max-block(log)
44:     reply [Read-R, status, val-ts, b] to coord
45: when receive [Order, ts] from coord
46:     status ←(ts > max-ts(log) and ts ≥ ord-ts)
47:     if status then ord-ts ←ts; store(ord-ts)
48:     reply [Order-R, status] to coord
49: when receive [Order&Read, j, max, ts] from coord
50:     status ←(ts > max-ts(log) and ts ≥ ord-ts)
51:     lts ← LowTS; b ← ⊥
52:     if status then
53:         ord-ts ←ts; store(ord-ts)
54:         if j = i or j = ALL then
55:             [lts, b] ←max-below(log, max)
56:     reply [Order&Read-R, status, lts, b] to coord
57: when receive [Write, [b_1, ..., b_n], ts] from coord
58:     status ←(ts > max-ts(log) and ts ≥ ord-ts)
59:     if status then log ←log ∪{[ts, b_i]}; store(log)
60:     reply [Write-R, status] to coord
```

4.4 Reading and writing a single block

Algorithm 3 defines the methods and message handlers for reading and writing an individual block.

The read-block method, which reads a given block number (j), is almost identical to the read-stripe method except that, in the common case, only p_j performs a read. The write-block method updates the parity blocks as well as the data block at process p_j. This is necessary when an I/O request has written to a single block of the stripe, in order to maintain consistency of the whole stripe. In the common case without any partial write, this method reads from, and writes to, process p_j and the parity processes (fast-write-block). Otherwise, it essentially performs a recovery (Line 17), except that it replaces the jth block with the new value upon write-back.

5 Discussion

5.1 Garbage collection of old data

Our algorithm relies on each process keeping its entire history of updates in a persistent log, which is not practical. For the correctness of the algorithm, it is sufficient that each process remember the most recent timestamp-data pair that was part of a complete write. Thus, when a coordinator has successfully updated a full quorum with a timestamp *ts*, it

Algorithm 3 Block methods and handlers for p_i

```
61: procedure read-block(j)
62:     replies ←quorum([Read, {j}])
63:     if status is all true and p_j replied
            and val-ts in all replies is the same then
64:         return the block in p_j's reply
65:     s ← recover()
66:     if s ≠ ⊥ then
67:         return s[j]
68:     else
69:         return ⊥
70: procedure write-block(j, b)
71:     ts ←newTS()
72:     if fast-write-block(j, b, ts) = OK then return OK
73:     else return slow-write-block(j, b, ts)
74: procedure fast-write-block(j, b, ts)
75:     replies ←quorum([Order&Read, j, HighTS, ts])
76:     if status contains false or p_j did not reply then
77:         return ⊥
78:     b_j ←the block in p_j's reply
79:     ts_j ←the timestamp in p_j's reply
80:     replies ←quorum([Modify, j, b_j, b, ts_j, ts])
81:     if status is all true then return OK
82:     else return ⊥
83: procedure slow-write-block(j, b, ts)
84:     data ←read-prev-stripe(ts)
85:     if data = ⊥ then return ⊥
86:     data[j] ←b
87:     return store-stripe(data, ts)
88: when receive [Modify, j, b_j, b, ts_j, ts] from coord
89:     status ←(ts_j = max-ts(log) and ts ≥ ord-ts)
90:     if status then
91:         if i = j then
92:             b_i ←b
93:         else if i > m then
94:             b_i ←modify_{j,i}(b_j, b, max-block(log))
95:         else
96:             b_i ←⊥
97:         log ←log ∪{[ts, b_i]}; store(log)
98:     reply [Modify-R, status] to coord
```

can safely send a garbage-collection message to all processes to garbage collect data with timestamps older than *ts*. Notice that the coordinator can send this garbage-collection message asynchronously after it returns OK.

5.2 Algorithm complexity

Table 1 compares the performance of our algorithm and state-of-the-art atomic-register constructions [9, 10]. We improve on previous work in two ways: efficient reading in the absence of failures or concurrent accesses, and support of erasure coding.

In describing our algorithm, we have striven for simplicity rather than efficiency. In particular, there are several straight-forward ways to reduce the network bandwidth consumed by the algorithm for block-level writes: (a) if we are writing block j, it is only necessary to communicate blocks to p_j and the parity processes, and (b) rather than sending both the old and new block values to the parity processes, we can send a single coded block value to each parity process instead.

6 Related work

As we discussed in Section 1.3, our erasure-coding algorithm is based on fundamentally different assumptions than traditional erasure-coding algorithms in disk arrays.

The algorithm in [14] also provides erasure-coded storage in a decentralized manner using a combination of a quorum system and log-based store. The algorithm in [14] handles Byzantine as well as crash failures, but does not explicitly handle process recovery (i.e., failures are permanent). In contrast, our algorithm only copes with crash failures, but incorporates an explicit notion of process recovery. Another difference is that the algorithm in [14] implements (traditional) linearizability where partial operations may take effect at an arbitrary point in the future, whereas our algorithm implements strict linearizability where partial operations are not allowed to remain pending. Finally, the algorithm in [14] only implements full-stripe reads and writes, whereas our algorithm implements block-level reads and writes as well.

The goal of [2] is to allow clients of a storage-area network to directly execute an erasure-coding algorithm when they access storage devices. The resulting distributed erasure-coding scheme relies on the ability for clients to accurately detect the failure of storage devices. Moreover, the algorithm in [2] can result

	Our algorithm							LS97	
	Stripe access			Block access					
	read/F	write	read/S	read/F	write/F	read/S	write/S	read	write
latency	2δ	4δ	6δ	2δ	4δ	6δ	8δ	4δ	4δ
# messages	$2n$	$4n$	$6n$	$2n$	$4n$	$6n$	$8n$	$4n$	$4n$
# disk reads	m	0	$n+m$	1	$k+1$	$n+1$	$k+n+1$	n	0
# disk writes	0	n	n	0	$k+1$	n	$k+n+1$	n	n
Network b/w	mB	nB	$(2n+m)B$	B	$(2n+1)B$	$(2n+1)B$	$(4n+1)B$	$2nB$	nB

Table 1: Performance comparison between our algorithm and the one by Lynch and Shvartsman [9]. The suffix "/F" denotes the operations that finishes without recovery. The suffix "/S" indicates the operations that execute recovery. We assume that recovery only requires a single iteration of the repeat loop. Parameter n is the number of processes, and $k = n - m$ (i.e., k is the number of parity blocks). We pessimistically assume that all replicas are involved in the execution of an operation. δ is the maximum one-way messaging delay. B is the size of a block. When calculating the number of disk I/Os, we assume that reading a block from *log* involves a single disk read, writing a block to *log* involves a single disk write, and that timestamps are stored in NVRAM.

in data loss when certain combinations of client and device failures occur. For example, consider a 2 out of 3 erasure-coding scheme with 3 storage devices: if a client crashes after updating only a single data device, and if the second data device fails, we cannot reconstruct data. In contrast, our algorithm can tolerate the simultaneous crash of all processes, and makes progress whenever an m-quorum of processes come back up and are able to communicate.

Several algorithms implement atomic read-write registers in an asynchronous distributed system based on message passing [4, 9, 10]. They all assume a crash-stop failure model, and none of them support erasure-coding of the register values.

References

[1] M. K. Aguilera and S. Frolund. Strict linearizability and the power of aborting. Technical Report HPL-2003-241, HP Labs, November 2003.

[2] K. Amiri, G. A. Gibson, and R. Golding. Highly concurrent shared storage. In *International Conference on Distributed Computing Systems (ICDCS)*, 2000.

[3] S. Asami. *Reducing the cost of system administration of a disk storage system built from commodity components*. PhD thesis, University of California, Berkeley, May 2000. Tech. Report. no. UCB-CSD-00-1100.

[4] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, 1995.

[5] J. Elson, L. Girod, and D. Estrin. Fine-grained network time synchronization using reference broadcasts. In *Proceedings of the 5th symposium on operatings systems design and implementation (OSDI)*, pages 147–163. USENIX, 2002.

[6] S. Frolund, A. Merchant, Y. Saito, S. Spence, and A. Veitch. FAB: enterprise storage systems on a shoestring. In *Proceedings of the Ninth Workshop on Hot Topics in Operating Systems (HOTOS IX)*. USENIX, 2003. to appear.

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

[8] iSCSI draft 20 specifi cation. http://www.diskdrive-.com/reading-room/standards.html, 2003.

[9] N. A. Lynch and A. A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of the IEEE Symposium on Fault-Tolerant Computing Systems (FTCS)*, pages 272–281, 1997.

[10] N. A. Lynch and A. A. Shvartsman. Rambo: A reconfi gurable atomic memory service for dynamic networks. In *16th Int. Conf. on Dist. Computing (DISC)*, October 2002.

[11] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (raid). In Harran Boral and Per-Ake Larson, editors, *Proceedings of 1988 SIGMOD International Conference on Management of Data*, pages 109–16, Chicago, IL, 1–3 June 1988.

[12] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software—Practice and Experience*, 27(9), 1997.

[13] Sean Reah, Patrik Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowics. Pond: the OceanStore prototype. In *Conference and File and Storage Technologies (FAST)*. USENIX, mar 2003.

[14] J. J. Wylie, G. R. Goodson, G. R. Ganger, and M. K. Reiter. Efficient byzantine-tolerant erasure-coded storage. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2004.

A Existence of *m*-quorum systems

We consider a set U of n processes, $U = \{p_1, \ldots, p_n\}$; up to f of these processes may be faulty. We we use $\mathcal{F}$ to denote the set of all subsets of U that contain f processes: $\mathcal{F} = \{F \in 2^U \mid |F| = f\}$. As in Section 2.2, we define an *m*-quorum system Q as a set of subsets of U that satisfies the following properties:

CONSISTENCY: $\forall Q_1, Q_2 \in Q : |Q_1 \cap Q_2| \geq m$.

AVAILABILITY: $\forall F \in \mathcal{F}, \exists Q \in Q : Q \cap F = \emptyset$.

We show the following Theorem abort *m*-quorum systems:

Theorem 2 *There exists an m-quorum system for U if and only if* $n \geq 2f + m$.

We prove the Theorem through the following two lemmas:

Lemma 3 *There exists an m-quorum system for U if and only if the set* $Q = \{Q \in 2^U \mid |Q| \geq n - f\}$ *is an m-quorum system for* U.

PROOF: Consider first the "if" part of the lemma. If Q is an *m*-quorum system for U, then it is obvious that an *m*-quorum system exists.

To prove the "only if" part of the lemma, assume that there exists an *m*-quorum system Q', yet Q is not an *m*-quorum system. Since Q clearly satisfies the AVAILABILITY property, it cannot satisfy the CONSISTENCY property. Thus, let Q_1 and Q_2 be two elements in Q such that $|Q_1 \cap Q_2| < m$. Define next two sets F_1 and F_2 as follows: $F_1 = U \setminus Q_1$ and $F_2 = U \setminus Q_2$. The sets F_1 and F_2 have f elements, and are thus elements of $\mathcal{F}$. This means that there are quorums Q'_1 and Q'_2 in Q' such that $Q'_1 \subseteq Q_1$ and $Q'_2 \subseteq Q_2$. Because $(Q'_1 \cap Q'_2) \subseteq (Q_1 \cap Q_2)$, we can conclude that $|Q'_1 \cap Q'_2| \leq |Q_1 \cap Q_2| < m$, which contradicts the assumption that Q' is an *m*-quorum system. □

Lemma 4 *The set* $Q = \{Q \in 2^U \mid |Q| \geq n - f\}$ *is n m-quorum system for U if and only if* $n \geq 2f + m$.

PROOF: We first prove that $n \geq 2f+m$ implies that $\mathcal{Q}$ is an m-quorum system. The set $\mathcal{Q}$ clearly satisfies the AVAILABILITY property. For CONSISTENCY, consider two sets Q_1 and Q_2 in $\mathcal{Q}$:

$$\begin{aligned} |Q_1 \cap Q_2| &= |Q_1| + |Q_2| - |Q_1 \cup Q_2| \\ &= 2(n-f) - |Q_1 \cup Q_2| \\ &\geq 2(n-f) - n \\ &= n - 2f \\ &\geq m \end{aligned}$$

To prove that $n \geq 2f+m$ is a necessary condition for $\mathcal{Q}$ being an m-quorum system, consider two sets Q_1 and Q_2 in $\mathcal{Q}$ such that $Q_1 \cup Q_2 = U$. We know that $Q_2 = (U \setminus Q_1) \cup (Q_1 \cap Q_2)$. Because $(U \setminus Q_1) \cap (Q_1 \cap Q_2) = \emptyset$, we have the following:

$$|Q_2| = |U \setminus Q_1| + |Q_1 \cap Q_2|$$

Since $|Q_2| = n-f$ and $|U \setminus Q_1| = f$, we have:

$$n - 2f = |Q_1 \cap Q_2| \geq m$$

□

B Algorithm correctness

We show that Algorithms 1, 2, and 3 correctly implement a strictly linearizable read-write register. Strict linearizability is defined in [1], and we do not repeat the definition here. Although the definition in [1] only covers crash-stop processes, it is straightforward to extend the definition to cover crash-recovery processes: we can reason about strict linearizability in a crash-recovery model by changing the well-formedness constraints for histories to allow multiple crash events in a given per-process history; in particular, the history transformation rules that deal with linearizing crash events do not assume that each process crashes at most once.

We represent a run of our algorithms as a history of *invocation* and *return* events. An invocation event captures the start of a particular execution of a method on a storage register, and a return event captures the completion of such an execution.

A storage register gives read-write access to a stripe of m blocks. To show correctness of our implementation, we show that it provides strictly linearizable read-write access to any block within the register's stripe. That is, we prove correctness on a per-block basis, but do so for both block-level and stripe-level operations that affect a given block. Considering each block separately simplifies the presentation because we do not have to reason about a mixture of stripe and block-level operations.

Given a history H of block-level and stripe-level operations on a storage register, we derive a history H_i that captures the read-write behavior relative to block i in the following manner. We translate a stripe-level operation that reads or writes a stripe value s in H into a corresponding block-level operation that reads or writes the i'th block value in s. We ignore block-level operations that read or write blocks other than i, and we include block-level reads and writes directly if they are targeted at block i. In the following, we only consider such block-level histories. In particular, the following correctness reasoning is relative to a given run R of our algorithm, and a given block i in the stripe maintained by the algorithm. We use H to refer to a possible block-level history H_i that R may give rise to.

As in [1], we use a notion of *operation* in conjunction with H.[3] A *complete* operation consists of an invocation event and a matching return event. If the return event does not contain the value $\perp$, we say that the operation is *successful*. A *partial* operation consists of an invocation event and a matching crash event. Finally, an *infinite* operation consists of an invocation event without any matching return or crash event. We use $\mathsf{write}(v)$ to refer to a operation whose invocation event captures the start of a write operation with parameter v. We assume that that each write operation tries to write a unique value (*"unique-value" assumption*). Moreover, we assume that no write operation tries to write nil, the initial value of any register. We use $\mathsf{read}(v)$ to represent an operation with a return event that captures the completion of executing the read method where v is the returned value.

We introduce a partial order on the operations in H. If the return or crash event of an operation op precedes the invocation event of another operation op' in

[3] What we call operation here is called operation instance in [1]

a history H, we say that op *happens before* op', and we write this as $\mathsf{op} \rightarrow_H \mathsf{op}'$.

We define the following subsets of Value:

- $\mathsf{Written}_H$ is the set of all values that are part of invocation events for write operations in H.
- $\mathsf{Commited}_H$ is the set of all values that are part of an invocation event for a write operation that return a status of OK in H.
- Read_H is the set of all values that are part of a return event for a read operation in H, or the value referenced by a successful write-block operation.
- We define the *observable* values in H as follows:

$$\mathsf{Observable}_H \equiv \mathsf{Read}_H \cup \mathsf{Commited}_H.$$

B.1 A sufficiency condition for strict linearizability of a storage register

Intuitively, a *conforming total order* is a totally-ordered set $(V, <)$ such that (a) V contains all the observable values in H, and (b) the ordering of values in V corresponds to the ordering of operation instances in H. More precisely:

Definition 5 *A totally ordered set $(V, <)$ is a conforming total order for H if $\mathsf{Observable}_H \subseteq V \subseteq \mathsf{Written}_H \cup \{\mathsf{nil}\}$ and if for all $v, v' \in V$ the following holds:*

$$\mathsf{nil} \in V \Rightarrow \mathsf{nil} \leq v \quad (1)$$
$$\mathsf{write}(v) \rightarrow_H \mathsf{write}(v') \Rightarrow v < v' \quad (2)$$
$$\mathsf{read}(v) \rightarrow_H \mathsf{read}(v') \Rightarrow v \leq v' \quad (3)$$
$$\mathsf{write}(v) \rightarrow_H \mathsf{read}(v') \Rightarrow v \leq v' \quad (4)$$
$$\mathsf{read}(v) \rightarrow_H \mathsf{write}(v') \Rightarrow v < v' \quad (5)$$

□

Proposition 6 *If H has a conforming total order then H is strictly linearizable.*

PROOF: Assume that $(V, <)$ is a conforming total order for H. We have to show that there exists a sequential history S such that $H \rightarrow S$ (where $\rightarrow$ is the history transformation relation defined in [1]) and such that S complies with the sequential specification of an atomic register (i.e., read operations in S always return the most recently written value or nil if no value has been written).

For every $v \in V$, construct a sequential history S_v as follows:

$$S_v = \begin{cases} \mathsf{write}(v) \cdot \mathsf{read}_1(v) \cdot \ldots \cdot \mathsf{read}_k(v) & v \neq \mathsf{nil} \\ \mathsf{read}_1(v) \cdot \ldots \cdot \mathsf{read}_k(v) & \text{otherwis e} \end{cases}$$

where k is the number of successful read operations that return v in H ($k \geq 0$). Next, construct the sequential history S in the following way:

$$S = S_{v_1} \cdot \ldots \cdot S_{v_m}$$

where $v_1 < v_2 < \ldots < v_m$ are the elements of V.

First observe that S complies with the sequential specification of an atomic register.

We now show that $H \rightarrow S$. To do so, we start with H and successively explain which rules from the definition of $\rightarrow$ to apply until we obtain S. First, use Rules (6)–(12) in [1] Figures 3, 4, and 5 to remove all partial, aborted, and infinite read operations from H. Second, apply these rules to write operations as follows. Let v be the parameter of a write operation. If $v \in V$, use Rule (8), (10), or (12) to convert the write operation to a successful write; otherwise ($v \notin V$), use Rule (6), (7), (9), or (11) to remove the write operation. We now have a history H' without crashes, aborts, and infinite aborted operation instances. Moreover, $H \rightarrow H'$.

We next show that $H' \rightarrow S$. We first claim that H' and S contain the same operations. To show the claim, note that every successful operation in H is part of both H' and S. Moreover, every unsuccessful read operation (i.e., partial, aborted, and infinite read operation) in H are in neither H' nor S. An unsuccessful write operation $\mathsf{write}(v)$ is part of S if and only if $v \in V$. But if $v \in V$, we convert the unsuccessful write operation in H to a successful operation in H'_O by the above transformation. This shows the claim.

Now, assume for a contradiction that $H' \not\rightarrow S$. Because the histories contain the same set of operations, and because these operations are all successful, there must be two operations op_i and op_j that are ordered differently in H' and S. But this is impossible because the value ordering in V obeys the operation ordering in H and thereby in H'. □

B.2 Constructing a conforming total order

Histories represent the external view of a storage register: they show the view of processes that interact with the register. We also consider internal events called *store events* to reason about the behavior of our algorithm.

A store event happens at a process p when p writes its *log* variable to stable storage in line 59 or line 97. A store event is parameterized by a value and a timestamp, and we use $\mathsf{st}(v,ts)$ to represent a store event parameterized by the value v and the timestamp ts. The parameters are given by the context in which a store event occurs as follows:

- A store event that happens in line 59 in response to a "Write" message of the form $[\text{Write}, [b_1, \ldots, b_i, \ldots b_n], ts]$ has b_i and ts as parameters, and is denoted $\mathsf{st}(b_i, ts)$.
- A store event that happens in line 97 in response to a "Modify" message of the form $[\text{Modify}, j, b_j, b, ts_j, ts]$ has b and ts as parameters, and is denoted $\mathsf{st}(b, ts)$.

We define SE_R to be the set of store events that happen in R. We define SV_R to be the set of values that are part of store events in a run R, and we define SE_R^v to be the (possibly empty) set of store events for a particular value v.

Definition 7 *Let v be a value in $\mathsf{SV}_R \cup \{\mathsf{nil}\}$. The timestamp ts_v is defined as follows:*

$$ts_v = \begin{cases} \mathsf{LowTS} & \text{if } v = \mathsf{nil} \\ \min\{ts \mid \mathsf{st}(v,ts) \in \mathsf{SE}_R^v\} & \text{otherwise} \end{cases}$$

□

We use the ordering on timestamps to define a total order $<_{val}$ on $\mathsf{SV}_R \cup \{\mathsf{nil}\}$ in the following manner:

Definition 8 *Let v and v' be values in $\mathsf{SV}_R \cup \{\mathsf{nil}\}$. We define $<_{val}$ as follows:*

$$v <_{val} v' \Leftrightarrow ts_v < ts_{v'}$$

□

This is a well-defined total order because different values are always stored with different timestamps. In the following, we omit the subscript from $<_{val}$, and simply use "$<$". With this convention, the symbol $<$ is overloaded to order both timestamps and values.

We show that $(\mathsf{Observable}_H, <)$ is a conforming total order for H.

Lemma 9

$$\mathsf{Observable}_H \setminus \{\mathsf{nil}\} \subseteq \mathsf{SV}_R \subseteq \mathsf{Written}_H.$$

PROOF: The read operation always collect at least m blocks with the same timestamp before returning (Lines 8, 32, and 63). Thus decode primitive is always able to construct the original block value. Moreover, that value is in $\mathsf{Written}_H$ by definition of $\mathsf{Written}_H$.□

For an operation op, $\mathsf{coord}(\mathsf{op})$ is the process that coordinates op, and $\mathsf{ts}(\mathsf{op})$ is the timestamp used by $\mathsf{coord}(\mathsf{op})$ (for a read operation, coord is defined only when the recover method is executed). We use the following notations to define the set of processes contacted by a coordinator:

- $\mathsf{Q}_R(\mathsf{op})$ is a quorum of processes after the successful completion of the "Read" messaging phase.
- $\mathsf{Q}_W(\mathsf{op})$ is a quorum of processes after the successful completion of the "Write" or "Read&Write" messaging phase.
- $\mathsf{Q}_O(\mathsf{op})$ is a quorum of processes after the successful completion of the "Order" or "Order&Read messaging phase.

Lemma 10 *For any process p, the value of ord-ts increases monotonically.*

PROOF: Variable *ord-ts* is modified only in Lines 47 and 53, both of which check beforehand if the new timestamp is larger than the current one. □

Lemma 11 *Let v be a non-*nil *value. ts_v, if it exists, it is the timestamp generated by write-block or write-stripe.*

PROOF: We have to show that ts_v is never generated by a recover method. Assume for a contradiction that a recover method generates ts_v for some value v. Consider the execution of read-prev-stripe during this recover method. Since the recover method results in a store event $\mathsf{st}(v, ts_v)$, the invocation of read-prev-stripe must return a stripe that contains v. The stripe is reconstructed from the replies to "Order&Read" messages. Thus, the processes that store this previous stripe value must have executed store events for v with some timestamp ts. From the algorithm and Lemma 10, we conclude that $ts < ts_v$, which is a contradiction. □

Lemma 12 *If a process executes* $\mathsf{st}(v, ts)$ *during some operation* op*, then an m-quorum has stored ts as the value of ord-ts during* op*.*

PROOF: A store event $\mathsf{st}(v, ts)$ happens only after the coordinator has collected either successful "Order-R" or successful "Order&Read-R" replies from an m-quorum of processes. The handler for successful messages "Order" or "Order&Read" messages set *ord-ts* to *ts* and store the value of *ord-ts.* □

Lemma 13 *Consider two distinct values* $v, v' \in \mathsf{SV}_R$*. If there exists a timestamp* $ts > ts_{v'}$ *such that an m-quorum of processes execute* $\mathsf{st}(v, ts)$*, then* $\forall\, \mathsf{st}(v', ts') \in \mathsf{SE}_R^{v'} : ts' < ts$*. That is, every store event for* v' *has a timestamp smaller than ts*

PROOF: Assume for a contradiction that a store event for v' with a timestamp larger than ts exists. Let ts'_{min} be the smallest timestamp among such store events:

$$ts'_{min} \equiv \min(\{\hat{ts} \mid \hat{ts} > ts \wedge \mathsf{st}(v', \hat{ts}) \in \mathsf{SE}_R^{v'}\}).$$

We first claim that the event $\mathsf{st}(v', ts'_{min})$ must be triggered by a recover method. To show the claim, consider first the case where $v' = \mathsf{nil}$. In this case, the claim follows from the fact that nil is never written. Consider next the case where $v' \neq \mathsf{nil}$. From Lemma 11, only the write-stripe or write-block methods use $ts_{v'}$ when storing v'. Moreover, we know that $ts_{v'} < ts < ts_{min}$, which implies that $ts_{v'} \neq ts_{min}$, and hence proves the claim.

Consider now the recover method that triggers $\mathsf{st}(v', ts'_{min})$. Let ts'' be the value of *max* when the read-prev-stripe method returns. We claim that $ts'' \geq ts$. To prove the claim, let Q be the m-quorum of processes that execute $\mathsf{st}(v, ts)$, and let op be the read-block or read-stripe operation that calls recover and triggers $\mathsf{st}(v', ts'_{min})$. We know that $|Q \cap \mathsf{Q}_O(\mathsf{op})| \geq m$. Thus, the value of *max* is at least *ts*, which proves the claim. Because the recover method triggers a store event for v', v' must be part of the (reconstructed) stripe returned by read-prev-stripe. Since $ts'' > \mathsf{LowTS}$, we know that the processes that store this stripe must have executed $\mathsf{st}(v', ts'')$. This contradicts the assumption that ts'_{min} is the smallest timestamp bigger than *ts* for which v' is stored. □

Lemma 14 *If* $\mathsf{read}(v) \in H$ *with* $v \neq \mathsf{nil}$*, then there exists a timestamp ts such that (a) an m-quorum executes* $\mathsf{st}(v, ts)$ *and (b) an m-quorum of processes have max-ts*$(log) = ts$ *sometime during the execution of* $\mathsf{read}(v)$*.*

PROOF: Consider the two ways in which $\mathsf{read}(v)$ can be executed:

- $\mathsf{read}(v)$ does not involve the recover method. In this case, processes in $\mathsf{Q}_R(\mathsf{read}(v))$ must have returned the same timestamp *ts* with no pending write or recover invocations. Thus, an m-quorum executed $\mathsf{st}(v, ts)$, and an m-quorum have max-ts$(log) = ts$ during $\mathsf{read}(v)$.
- $\mathsf{read}(v)$ involves the recover method. Let $ts = \mathsf{ts}(\mathsf{read}(v))$. For the recovery to succeed, $\mathsf{Q}_W(\mathsf{read}(v))$ must have executed $\mathsf{st}(v, ts)$ and replied "Write-R" to the coordinator, in which case an m-quorum has max-ts$(log) = ts$. □

Lemma 15 *If* $\mathsf{write}(v)$ *is in H and* $v \in \mathsf{Observable}_H$*, then (a) some process executes* $\mathsf{st}(v, ts_v)$ *during* $\mathsf{write}(v)$*, and (b) there is a timestamp ts such that an m-quorum executes* $\mathsf{st}(v, ts)$*.*

PROOF: If $v \in \mathsf{Commited}_H$, then all the properties hold vacuously. Suppose otherwise. Because $v \in \mathsf{Observable}_H$, we know that $v \in \mathsf{Read}_H$ and $\mathsf{read}(v)$ is in H. Moreover, we know that $v \neq \mathsf{nil}$ because nil is never written.

(a) Because $\mathsf{read}(v)$ is in H, some process must have executed $\mathsf{st}(v,ts)$ for some timestamp ts. The timestamp ts_v is merely the smallest such timestamp. Since ts_v exists, we know from Lemma 11 that $\mathsf{st}(v,ts_v)$ happens during $\mathsf{write}(v)$.

(b) From Lemma 14, an m-quorum executes $\mathsf{st}(v,ts)$ for some ts.

□

Lemma 16 *If* $\mathsf{op}_1 \rightarrow_H \mathsf{op}_2$ *and both operation trigger some store events, then the store events for* op_1 *have smaller timestamps than those for* op_2.

PROOF: Assume the contrary: $\mathsf{op}_1 \rightarrow_H \mathsf{op}_2$, op_1 executes $\mathsf{st}(v,ts_1)$, op_2 executes $\mathsf{st}(v',ts_2)$, yet $ts_1 > ts_2$.

Since op_1 executes $\mathsf{st}(v,ts_1)$, processes in $\mathsf{Q}_O(\mathsf{op}_1)$ store ts_1 for *ord-ts* at some point (happens in Lines 47 and 53). Similarly, processes in $\mathsf{Q}_O(\mathsf{op}_2)$ store ts_2 as their *ord-ts* at some point. Consider a process $p \in \mathsf{Q}_O(\mathsf{op}_1) \cap \mathsf{Q}_O(\mathsf{op}_2)$. Since $\mathsf{op}_1 \rightarrow_H \mathsf{op}_2$, this process stores ts to *ord-ts* before it stores ts' to *ord-ts*. Since processes only assign monotonically increasing values to *ord-ts* (Lemma 10), we have a contradiction. □

Lemma 17 *For* $v,v' \in \mathsf{Observable}_H$, *the following condition holds:*

$$\mathsf{write}(v) \rightarrow_H \mathsf{write}(v') \Rightarrow v < v'$$

PROOF: Assume otherwise: $\mathsf{write}(v) \rightarrow_H \mathsf{write}(v')$, $v,v' \in \mathsf{Observable}_H$, yet $v > v'$. From Lemma 15, we know that $\mathsf{st}(v,ts_v)$ happens during $\mathsf{write}(v)$ and that $\mathsf{st}(v',ts_{v'})$ happens during $\mathsf{write}(v')$. From Lemma 16, we conclude that $ts_v < ts_{v'}$. □

Lemma 18 *For* $v,v' \in \mathsf{Observable}_H$, *the following condition holds:*

$$\mathsf{read}(v) \rightarrow_H \mathsf{read}(v') \Rightarrow v \leq v'$$

PROOF: Assume for a contradiction that $\mathsf{read}(v) \rightarrow_H \mathsf{read}(v')$, yet $v > v'$. This means that $v \neq \mathsf{nil}$. From Lemma 14, for some timestamp ts, either $\mathsf{Q}_R(\mathsf{read}(v))$ or $\mathsf{Q}_W(\mathsf{read}(v))$ has their max-ts$(log) = ts$ some time during the execution of $\mathsf{read}(v)$. Let Q_v be this m-quorum, and consider the following two cases:

- $v' = \mathsf{nil}$. We consider two cases: (a) $\mathsf{read}(v')$ executes recover or (b) $\mathsf{read}(v')$ does not execute recover. In case (a), let $ts' = \mathsf{ts}(\mathsf{read}(v')$ and let $Q_{v'} = \mathsf{Q}_W(\mathsf{read}(v))$. Let p be a process in $Q_v \cap Q_{v'}$. From Lemma 10, we know that $ts' \geq ts$. The fact that the recover method executes $\mathsf{st}(\mathsf{nil},ts')$ contradicts Lemma 13. In case (b), $\mathsf{Q}_R(\mathsf{read}(v'))$ has some timestamp ts' as their value of max-ts(log). We know there is a process p in $Q_v \cap \mathsf{Q}_R(\mathsf{read}(v'))$, and from Lemma 10, we conclude that $ts' \geq ts$. Thus, $ts' > \mathsf{LowTS}$, and there must be some store event $\mathsf{st}(\mathsf{nil},ts')$, which contradicts Lemma 13.
- $v' \neq \mathsf{nil}$. From Lemma 14, for some timestamp ts', an m-quorum executes $\mathsf{st}(v',ts')$ and either $\mathsf{Q}_R(\mathsf{read}(v'))$ or $\mathsf{Q}_W(\mathsf{read}(v'))$ has their max-ts$(log) = ts'$ sometime during the execution of $\mathsf{read}(v')$. Let Q'_v be this m-quorum. Let $p \in Q_v \cap Q'_v$. Because $\mathsf{read}(v)$ precedes $\mathsf{read}(v')$, and from Lemma 10, we conclude that $ts < ts'$.
  Moreover, $ts_{v'} < ts_v < ts < ts'$. ($ts_{v'} < ts_v$ because $v' < v$; $ts_v < ts$ from the definition of ts_v). Since Q_v executes $\mathsf{st}(v,ts)$, Lemma 13 implies that all store events for v' have a timestamp that is smaller than ts. But this contradicts the fact that Q'_v executes $\mathsf{st}(v',ts')$ with $ts < ts'$.

□

Lemma 19 *For* $v,v' \in \mathsf{Observable}_H$, *the following condition holds:*

$$\mathsf{write}(v) \rightarrow_H \mathsf{read}(v') \Rightarrow v \leq v'$$

PROOF: Assume for a contradiction that $\mathsf{write}(v) \rightarrow_H \mathsf{read}(v')$, $v \in \mathsf{Observable}_H$, yet $v > v'$.

We first show that there exists a timestamp $ts' > ts_v$ such that a m-quorum executes $\mathsf{st}(v',ts')$. We consider two situations:

(a) $\mathsf{read}(v')$ executes the recover method.
Let $ts' = \mathsf{ts}(\mathsf{read}(v'))$. We know that an m-quorum executes $\mathsf{st}(v',ts')$. Furthermore, from Lemma 15, we know that at least one store event $\mathsf{st}(v,ts_v)$ happens during $\mathsf{write}(v)$. From Lemma 16, $ts_v < ts'$.

(b) $\mathsf{read}(v')$ executes only the read method.
Then $\mathsf{Q}_R(\mathsf{read}(v'))$ has some timestamp ts' as their value for both *ord-ts* and max-ts(*log*). According to Lemmas 12 and 15, $\mathsf{Q}_O(\mathsf{write}(v))$ has ts_v as their value for *ord-ts* some time during $\mathsf{write}(v)$. Consider a process $p \in \mathsf{Q}_R(\mathsf{read}(v')) \cap \mathsf{Q}_O(\mathsf{write}(v))$. From Lemma 10 and the fact that $v \neq v'$, $ts_v < ts'$. In particular, we then know that $ts' \neq \mathsf{LowTS}$, and therefore that $v' \neq \mathsf{nil}$. Since $v' \neq \mathsf{nil}$, we conclude that $\mathsf{Q}_R(\mathsf{read}(v'))$ executed $\mathsf{st}(v', ts')$.

Let ts be a timestamp such that an m-quorum executes $\mathsf{st}(v, ts)$—Lemma 15 guarantees the existence of ts. Per definition, we know that $ts_v < ts$. From the above reasoning, we also have a timestamp ts' such that an m-quorum executes $\mathsf{st}(v', ts')$ and such that $ts_v < ts'$.

We now have one of two situations: (c) $ts > ts'$ or (d) $ts < ts'$. For (c), we have $ts_v < ts' < ts$, which contradicts Lemma 13. For (d), we have $ts_{v'} < ts_v < ts'$, which also contradicts Lemma 13. □

Lemma 20 *For $v, v' \in \mathsf{Observable}_H$, the following condition holds:*

$$\mathsf{read}(v) \rightarrow_H \mathsf{write}(v') \Rightarrow v < v'$$

PROOF: Assume for a contradiction that $\mathsf{read}(v) \rightarrow_H \mathsf{write}(v')$, $v' \in \mathsf{Observable}_H$, yet $v \geq v'$. Because $v' \neq \mathsf{nil}$ (nil is never written), we can conclude that $v \neq \mathsf{nil}$. We consider two cases:

(a) $\mathsf{read}(v)$ executes only the read method.
We know that all the processes in $\mathsf{Q}_R(\mathsf{read}(v))$ has some timestamp ts for their *ord-ts* and max-ts(*log*) during $\mathsf{read}(v)$. From Lemmas 12 and 15, $\mathsf{Q}_O(\mathsf{write}(v'))$ has $ts_{v'}$ as their value for *ord-ts* during $\mathsf{write}(v')$. Consider a process $p \in \mathsf{Q}_R(\mathsf{read}(v)) \cap \mathsf{Q}_O(\mathsf{write}(v'))$. From Lemma 10, we know that $ts < ts_{v'}$. Since an m-quorum executes $\mathsf{st}(v, ts)$, $ts_v < ts$. Thus, we conclude that $ts_v < ts < ts_{v'}$, which contradicts the assumption that $v > v'$.

(b) $\mathsf{read}(v)$ executes both the read and recover methods.
Let $ts = \mathsf{ts}(\mathsf{read}(v))$. From Lemma 15, a store event $\mathsf{st}(v', ts_{v'})$ happens during $\mathsf{write}(v')$. From Lemma 16, we know that $ts < ts_{v'}$. As for case (a), we can now derive a contradiction based on the fact that $ts_v < ts$. □

Proposition 21 *The set $(\mathsf{Observable}_H, <)$ is a conforming total order for H.*

PROOF: Lemma 9 shows that $<$ is a total order for $\mathsf{Observable}_H$. Moreover, the lemma also shows that $\mathsf{Observable}_H \subseteq \mathsf{Written}_H \cup \{\mathsf{nil}\}$. Because LowTS is the smallest timestamp, we have that nil is the smallest value in $\mathsf{Observable}_H$. Condition (2)–(5) follow from Lemma 17–20. □

B.3 Proof of liveness

We show that our implementatation satisfies weak progress [1]: for every process p, if eventually only p issues operations then there is a time after which operations do not abort.

Proposition 22 *For any process p, if $H|p$ contains an invocation event, then $H|p$ either contains a subsequent return event or a subsequent crash event.*

PROOF: We show the proposition for the read-stripe method, the proof of termination for the other methods follow from similar reasoning.

Assume for a contradiction that a process history $H|p$ contains an invocation event, but no subsequent return nor crash event.

Because quorum is non-blocking, it is sufficient to prove termination for the recover method. Similarly, because newTS is non-blocking, termination of read-prev-stripe implies termination of the recover method. Consider now termination of the read-prev-stripe method. Since p does not crash, we conclude that the **repeat** loop never terminates. However, in successive loop iterations, the *max* timestamp takes on decreasing values, which means that *max* eventually will take on LowTS. When *max* is equal to LowTS, all returned blocks will have nil as their value. In particular, there will be at least m blocks returned with nil as value, which contradicts the conclusion that the loop never terminates. □

Proposition 23 *If only a single process p has a history $H|p$ that contains an infinite number of invocation events, and if p is correct, then $H|p$ contains an infinite number of successful return events.*

PROOF: Because p is the only process with an infinite number of invocation events, all other processes generate only a finite number of timestamps. Let ts be the maximum timestamp generated by processes other than p.

Assume that $H|p$ contains an infinite number of unsuccessful return events. From Algorithm 1 and Algorithm 3, we can observe that each invocation with an unsuccessful return event causes the generation of a timestamp. Thus, we know that p generates an infinite number of timestamps. The PROGRESS property of timestamp ensures that p eventually generates a timestamp ts' that is higher than ts. Because p is correct, there is a time t such that (a) p does not crash after t and (b) p invokes a method after t and generates a timestamp ts_c that is greater than ts. Consider this invocation. No replica will reply NO during this invocation because ts_c is higher than any timestamp in the system. This means that the invocation will return successfully, which is a contradiction. □