# 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 [15, 2, 13, 11].

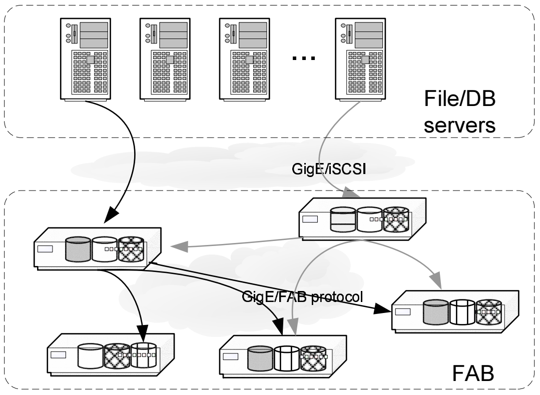

Figure 1: A typical FAB structure. Client computers connect to the FAB bricks using standard protocols. Clients can issue requests to any brick to access any logical volume. The bricks communicate among themselves using the specialized protocol discussed in this paper.

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 [8, 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

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).

by running voting over a 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 [14]) 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-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

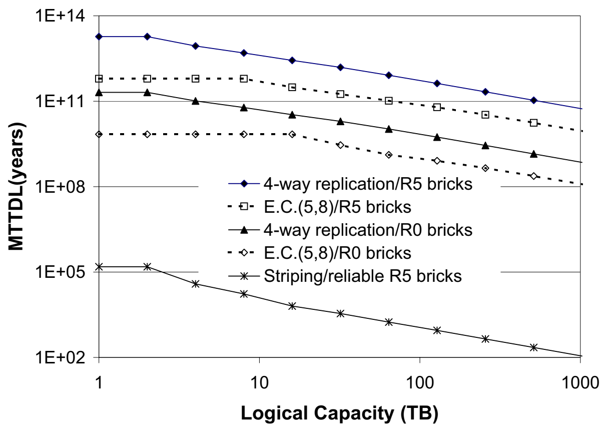

Figure 2: Mean time to first data loss (MTTDL) in storage systems using (1) striping, (2) replication and (3) erasure coding. (1) Data is striped over conventional, high-end, high-reliability arrays, using internal RAID-5 encoding in each array/brick. Reliability is good for small systems, but does not 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.

(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.

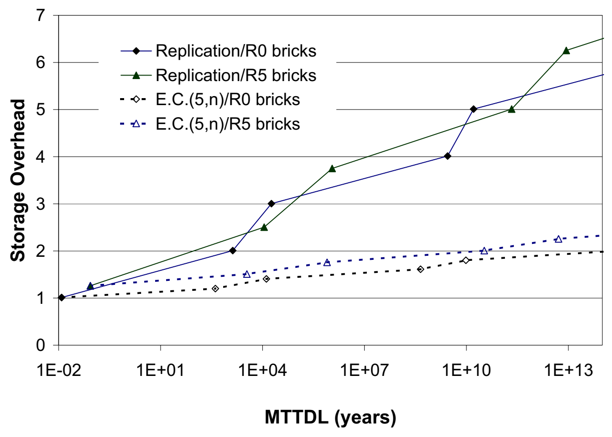

Figure 3: Storage overheads (raw capacity/logical capacity) of systems using replication and erasure coding. The storage overhead of replication-based systems rises much more steeply with increasing reliability requirements than for systems based on erasure-coding. Using RAID-5 bricks reduces the overhead 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.

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, $U = \{p_1, \dots, p_n\}$. 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-$n$ 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 $j$.

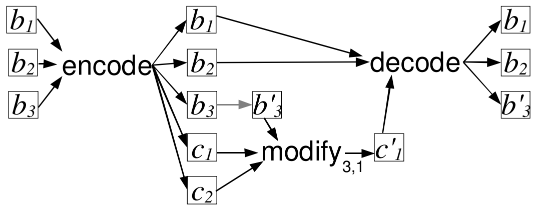

Figure 4: Use of the primitives for a 3-out-of-5 erasure coding scheme. Data blocks $b_1$ to $b_3$ form a stripe. The encode function generates two parity blocks $c_1$ and $c_2$. When $b_3$ is updated to become $b'_3$, we call $\mathsf{modify}_{3,1}(b_3, b'_3, c_1)$ to update $c_1$ to become $c'_1$.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

### 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 [7]). 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, $\text{LowTS} < t < \text{HighTS}$. 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 [8], 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. In [7], we prove the algorithm correct.

## 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 $p_1 \ldots p_m$ store the data blocks, and $p_{m+1} \ldots p_n$ store the parity blocks. It is straightforward to adapt the algorithm to more sophisticated data-layout schemes. In the following, we refer to $p_{m+1} \ldots p_n$ 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.

**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.

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

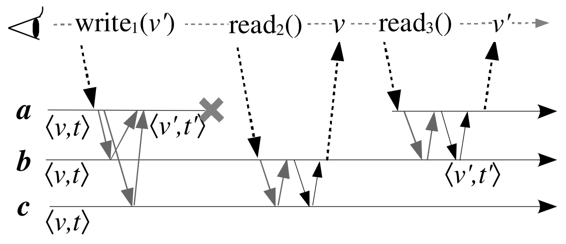

Figure 5: To ensure strict linearizability, read operations cannot simply pick, and possibly write-back, the value with the highest timestamp. In the example, the processes $a$, $b$ and $c$ implement a storage register; for simplicity, we use an erasure-coding scheme with a stripe size of 1 and where 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 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.

**4.1.2. 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: write$_1$ $\rightarrow$ read$_2$ $\rightarrow$ 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 of processes has no partial write; if so, it simply returns the current register value: the value received from the process containing 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. The write-back aborts any previous partial write operation.

## 4.2. Persistent data structures

Each process has persistent storage that survives crashes. In general, the 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.

```
 1: procedure read-stripe()
 2:   val ←fast-read-stripe()
 3:   if $val = \perp$ 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 $\perp$
12: procedure write-stripe(stripe)
13:   ts ←newTS()
14:   replies ←quorum([Order, ts])
15:   if status in any reply is false then return $\perp$
16:   else return store-stripe(stripe, ts)
17: procedure recover()
18:   ts ←newTS()
19:   s ←read-prev-stripe(ts)
20:   if $s \neq \perp$ and store-stripe($s$, $ts$) $=$ OK then
21:     return s
22:   else
23:     return $\perp$
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 $\perp$
30:     max ←the highest timestamp in replies
31:     blocks ←the blocks in replies with
            timestamp max
32:   until $|$ blocks $| \geq m$
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 $\perp$
```

Algorithm 1: Methods for accessing the entire stripe.

```
38: when receive [Read, $targets$] from coord
39:    $val\text{-}ts \leftarrow$ max-ts($log$)
40:    $status \leftarrow val\text{-}ts \geq ord\text{-}ts$
41:    $b \leftarrow \perp$
42:    if $status$ and $i \in targets$ then
43:       $b \leftarrow$ max-block($log$)
44:    reply [Read-R, $status$, $val\text{-}ts$, $b$] to coord

45: when receive [Order, $ts$] from coord
46:    $status \leftarrow (ts >$ max-ts($log$) and $ts \geq ord\text{-}ts)$
47:    if $status$ then $ord\text{-}ts \leftarrow ts$; store($ord\text{-}ts$)
48:    reply [Order-R, $status$] to coord

49: when receive [Order&Read, $j$, $max$, $ts$] from coord
50:    $status \leftarrow (ts >$ max-ts($log$) and $ts \geq ord\text{-}ts)$
51:    $lts \leftarrow$ LowTS; $b \leftarrow \perp$
52:    if $status$ then
53:       $ord\text{-}ts \leftarrow ts$; store($ord\text{-}ts$)
54:       if $j = i$ or $j =$ ALL then
55:          $[lts, b] \leftarrow$ max-below($log$, $max$)
56:    reply [Order&Read-R, $status$, $lts$, $b$] to coord

57: when receive [Write, $[b_1, \ldots, b_n]$, $ts$] from coord
58:    $status \leftarrow (ts >$ max-ts($log$) and $ts \geq ord\text{-}ts)$
59:    if $status$ then $log \leftarrow log \cup \{[ts, b_i]\}$; store($log$)
60:    reply [Write-R, $status$] to coord
```

Algorithm 2: Register handlers for process $p_i$

```
61: procedure read-block($j$)
62:    $replies \leftarrow$ quorum([Read, $\{j\}$])
63:    if status is all true and $p_j$ replied
          and $val\text{-}ts$ in all replies is the same then
64:       return the block in $p_j$'s reply
65:    $s \leftarrow$ recover()
66:    if $s \neq \perp$ then
67:       return $s[j]$
68:    else
69:       return $\perp$

70: procedure write-block($j$, $b$)
71:    $ts \leftarrow$ 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:    $replies \leftarrow$ quorum([Order&Read, $j$, HighTS, $ts$])
76:    if status contains false or $p_j$ did not reply then
77:       return $\perp$
78:    $b_j \leftarrow$ the block in $p_j$'s reply
79:    $ts_j \leftarrow$ the timestamp in $p_j$'s reply
80:    $replies \leftarrow$ quorum([Modify, $j$, $b_j$, $b$, $ts_j$, $ts$])
81:    if status is all true then return OK
82:    else return $\perp$

83: procedure slow-write-block($j$, $b$, $ts$)
84:    $data \leftarrow$ read-prev-stripe($ts$)
85:    if $data = \perp$ then return $\perp$
86:    $data[j] \leftarrow b$
87:    return store-stripe($data$, $ts$)

88: when receive [Modify, $j$, $b_j$, $b$, $ts_j$, $ts$] from coord
89:    $status \leftarrow (ts_j =$ max-ts($log$) and $ts \geq ord\text{-}ts)$
90:    if $status$ then
91:       if $i = j$ then
92:          $b_i \leftarrow b$
93:       else if $i > m$ then
94:          $b_i \leftarrow$ modify$_{j,i}$($b_j$, $b$, max-block($log$))
95:       else
96:          $b_i \leftarrow \perp$
97:       $log \leftarrow log \cup \{[ts, b_i]\}$; store($log$)
98:    reply [Modify-R, $status$] to coord
```

Algorithm 3: Block methods and handlers for $p_i$

### 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 $j$th 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 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 sev-

|  | Our algorithm |  |  |  |  |  |  | lynch-shvartsman1997 |  |
|---|---|---|---|---|---|---|---|---|---|
|  | Stripe access |  |  | Block access |  |  |  |  |  |
|  | read/F | write | read/S | read/F | write/F | read/S | write/S | read | write |
| latency | $2\delta$ | $4\delta$ | $6\delta$ | $2\delta$ | $4\delta$ | $6\delta$ | $8\delta$ | $4\delta$ | $4\delta$ |
| # 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. $\delta$ 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.

eral 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 [15] also provides erasure-coded storage in a decentralized manner using a combination of a quorum system and log-based store. The algorithm in [15] 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 [15] 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 [15] 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 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, December 2003.

[2] K. Amiri, G. A. Gibson, and R. Golding. Highly concurrent shared storage. In *20th Int. Conf. on Dist. Comp. Sys. (ICDCS)*, Taipei, Taiwan, April 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 (JACM)*, 42(1):124–142, 1995.

[5] J. Elson, L. Girod, and D. Estrin. Fine-grained network time synchronization using reference broadcasts. In *5th Symp. on Op. Sys. Design and Impl. (OSDI)*, pages 147–163, Boston, MA, USA, December 2002.

[6] S. Frolund, A. Merchant, Y. Saito, S. Spence, and A. Veitch. FAB: Enterprise Storage Systems on a Shoestring. In *8th Workshop on Hot Topics in Operating Systems (HOTOS-VIII)*, Kauai, HI, USA, May 2003.

[7] S. Frolund, A. Merchant, Y. Saito, S. Spence, and A. Veitch. A Decentralized Algorithm for Erasure-Coded Virtual Disks. Technical Report HPL-2004-46, HP Labs, April 2004.

[8] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. on Prog. Lang. and Sys. (TOPLAS)*, 12(3):463–492, July 1990.

[9] N. A. Lynch and A. A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *27th Int. Symp. on Fault-Tolerant Computing (FTCS)*, pages 272–281, Seattle, WA, USA, June 1997.

[10] N. A. Lynch and A. A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *16th Int. Conf. on Dist. Computing (DISC)*, pages 173–190, Toulouse, France, October 2002.

[11] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Int. Conf. on Management of Data (SIGMOD)*, pages 109–16, Chicago, IL, USA, 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] S. Reah, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowics. Pond: the OceanStore prototype. In *USENIX Conf. on File and Storage Technologies (FAST)*, San Francisco, CA, March 2003.

[14] J. Satran, K. Meth, et al. RFC3720: Internet small computer systems interface (iSCSI). http://www.faqs.org/rfcs-/rfc3720.html, 2004.

[15] J. J. Wylie, G. R. Goodson, G. R. Ganger, and M. K. Reiter. Efficient byzantine-tolerant erasure-coded storage. In *Int. Conf. on Dependable Systems and Networks (DSN)*, Frorence, Italy, June 2004.