Using Erasure Codes Efficiently for Storage in a Distributed System*

Marcos K. Aguilera
HP Laboratories
1501 Page Mill Road MS 1250
Palo Alto, CA, USA

Ramaprabhu Janakiraman
Dept. of Computer Science and Engineering
Washington University
Saint Louis, MO 63130, USA

Lihao Xu
Dept. of Computer Science and Engineering
Washington University
Saint Louis, MO 63130, USA

Abstract

Erasure codes provide space-optimal data redundancy to protect against data loss. A common use is to reliably store data in a distributed system, where erasure-coded data are kept in different nodes to tolerate node failures without losing data. In this paper, we propose a new approach to maintain ensure-encoded data in a distributed system. The approach allows the use of space efficient k-of-n erasure codes where n and k are large and the overhead $n-k$ is small. Concurrent updates and accesses to data are highly optimized: in common cases, they require no locks, no two-phase commits, and no logs of old versions of data. We evaluate our approach using an implementation and simulations for larger systems.

1. Introduction

Storage systems are quickly growing in size through the use of more and bigger disks, and through distribution over a network. With larger systems, the chance of component failure also increases, so techniques to protect data become more important. Single parity used in RAID systems no longer provides sufficient protection in all cases [1], and k-way replication is much too wasteful in storage space, even for small k. Therefore, new schemes are needed to protect data against multiple failures in a distributed storage system.

Erasure codes [2] have been used traditionally in communication systems, and more recently in storage systems as an alternative to replication (e.g., [3], [4], [5], [6]). Proper use of erasure codes provides greater space efficiency and fine tunable levels of protection, at the cost of greater complexity. An (n, k) MDS erasure code, or simply k-of-n code, encodes k blocks of data into $n > k$ blocks—which we call a *stripe*—such that any k blocks in the stripe can recover the original k blocks. By storing each block in a separate node, data are protected against the simultaneous failure of up to $n - k$ nodes.

A general challenge of distributed storage is to provide data consistency while allowing failures and concurrent access. At the same time, one would like to get reasonable performance, to scale with number of clients, and to allow expansion of storage capacity at low cost. These difficulties are well-recognized, understood, and reasonably addressed for replication-based storage. For erasure-coded storage, however, different schemes are still being proposed (e.g., [5], [6]), as researchers explore new ways to deal with the additional complexity created by erasure codes. Roughly speaking, this complexity is caused by an inherent coupling of data in erasure codes, as we explain deeper in the paper.

*This work is partially supported by NSF grants CCR-0208975, CNS-0322615, and IIS-0430224.

This paper proposes a new protocol and scheme to use erasure codes for distributed storage. Our scheme has the following features:

- *High concurrency:* It allows concurrent updates of blocks, including blocks coupled by the erasure code.
- *Consistency:* It ensures a strong type of consistency despite concurrent updates, and crashes of both storage nodes and clients.
- *Optimized for common cases:* It is highly optimized for the common cases where no failures occur; in such cases, it does not require use of expensive synchronization. A read requires only a round-trip to a storage node, and a write requires only a round-trip to each storage location that needs to be changed according to the erasure code; this is true even when there are concurrent operations.
- *Good performance with* highly-efficient *erasure codes:* The scheme performs well with k-of-n Reed-Solomon codes where k and n are large and $n - k$ is small—these are the codes with the best space efficiency for a given fault resiliency.
- *Online recovery:* When failures occur, recovery does not require to suspend read and write operations.
- *Small space overhead:* It keeps a small amount of overhead data at storage nodes—a goal consistent with using erasure codes to save space.
- *Thin servers:* It uses thin storage nodes that implement very simple functionality, thus following the principle of moving functionality to clients. This implies better scalability and lower costs to expand storage capacity when new servers are added.

To evaluate our approach, we built a prototype of a distributed and reliable storage service. The service comprises a set of storage nodes accessible to clients via a network, where clients read and write data using our protocols. We also use simulations to study the performance of larger systems.

Limitations of our approach

As a trade-off for its features, our scheme has four limitations:

- It is tailored for linear erasure codes, like Reed-Solomon codes, where redundant blocks are updated with commutative operations.
- It uses the redundancy of erasure codes solely for fault tolerance, not to improve read performance. For instance, our read performance is very similar to that of a system with no data redundancy. This is consistent with our motivation of supporting highly-efficient codes where $n - k \leq k$ (number of redundant blocks no greater than data blocks). Systems that use erasure code to improve read performance tend to have $n - k >> k$, and provide weak consistency or assume data are immutable.
- The write throughput of clients decreases as $n-k$ grows. However, this can be avoided if broadcasts are available.
- It can tolerate at most t_p client failures, where t_p is a chosen

failure threshold. If there are t_p+1 client failures and a storage crash, data may be lost. For example, consider the following scenario: (a) t_p+1 clients are simultaneously writing to the same stripe S, and (b) a network partition, such as caused by a switch failure, causes those t_p+1 clients to be permanently disconnected. This results in t_p+1 client partial writes that make the system vulnerable: a subsequent storage crash in this configuration cannot be tolerated. We mitigate this problem by using a monitoring mechanism that efficiently detects and fixes partial writes, to restore full recoverability and reduce the window of vulnerability. After any number of client failures, if this mechanism executes before a storage crash, data are safe again.

Related work and protocol comparison

The closest related work are the distributed protocols proposed by FAB [5] and Goodson et al [6]. FAB uses erasure codes in a distributed disk array built from low-cost commodity computers. The FAB protocol tolerates crash failures, ensures a strong form of consistency, and allows concurrent writes and reads. Concurrent writes to blocks in the same erasure code stripe return an exception. Servers have non-volatile memory and keep a log with old versions of data, which is periodically garbage collected.

Goodson et al also propose a protocol (GWGR) for distributed storage using erasure codes. The GWGR protocol tolerates Byzantine clients and storage nodes, ensures a strong form of consistency, and allows concurrent writes and reads. GWGR keeps a log with old versions of data for recovery, which is periodically garbage collected. GWGR only allows writes to modify the entire erasure code stripe at once; to modify individual blocks, it is necessary to read its stripe, and write it back. Doing so, however, has a performance cost and does not ensure consistency of concurrent updates to blocks in the same stripe.

Fig. 1 shows a performance comparison between our protocol (AJX-*), FAB and GWGR. Our protocol has at least as good latency, number of messages, and bandwidth. With FAB and GWGR, every write needs to contact all storage nodes in the erasure code stripe, and so these protocols perform poorly for random I/O, especially with *highly-efficient* erasure codes that have large k and n, and small $p=n-k$. These are the codes with best space efficiency for a given fault resiliency. For sequential I/O, all protocols allow pipelining of requests; with the optimizations of Section 6, we believe that our protocol is competitive, as shown by the results of Section 6. Thus, the advantages of our protocol over FAB and GWGR are (1) it supports well highly-efficient erasure codes, (2) it does not keep old versions of data at storage nodes (less space overhead), and (3) it allows for thin storage servers. An advantage of FAB and GWGR is that they can tolerate any number of client failures by using the log of old versions of data.* Our protocol keeps no log and tolerates only a chosen number of client crashes.

Myriad [4] proposes erasure codes for *disaster tolerance*. Updates of redundant disks do not happen during a write, but are instead deferred and done in batches using two-phase commit. Zhang and Lian [7] also propose a general scheme to use erasure codes for distributed storage. However, this scheme does not handle concurrent updates; instead, it assumes some external mechanism, like a transaction for each operation. This appears to be expensive, but no performance data is provided. LH^*_{RS} [8] use erasure codes to implement an expandable and distributed data structure, where redundancy increases with the amount of data, to ensure a minimum availability. However, the paper does not address failures of clients or recoveries concurrent with client updates. In some systems, erasure cores are used for archival of immutable data (e.g., [3]). When data is immutable, there are few concurrency issues, and so much simpler protocols are needed than what we propose.

*In FAB, because client and storage nodes are colocated, this requires some fraction of the nodes to restart after failing.

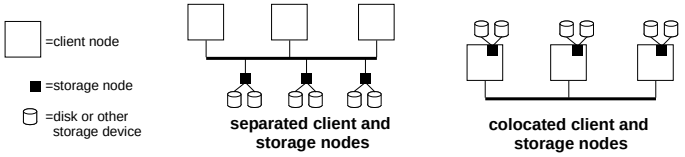


Fig. 2. System with client nodes and storage nodes that communicate via a network. Storage nodes may be thin or powerful devices; client nodes are computers running applications. We support both separated or colocated storage and client nodes.

The rest of this paper is organized as follows. In Section 2 we explain our assumptions and goals. Section 3 explains our design and protocols. We consider the protocol's failure resilience in Section 4. In Section 5 we validate our approach in two ways: An implementation is described in Section 5.1 and simulations of larger systems are described in Section 5.2. We give results in Section 6. All protocol correctness proofs are omitted due to space limitations; they can be found in [9].

2. Assumptions and Goals

We consider a distributed system where *client nodes* wish to store data at a set of *storage nodes* that are reachable through a fast network, like a local area network. A storage node may be a dedicated server with lots of memory, many processors, and a set of one or more disks or other storage connected to it. Or it could be a very thin passive device with a network interface, a storage interface, some memory, and some storage connected to it (and not much more beyond that). Storage devices have a standard fixed block size (e.g., 512 bytes) used as the minimum quantum of data transfer. A client node is a computer running applications that need to read and write data stored at the storage nodes. Client nodes have reasonable computational power and a network bandwidth that is not extremely limited. A client node may be colocated with a storage node, but the client node may need to access storage nodes not colocated with it. This might occur if a machine is powerful enough to host both applications and shared storage.

Each node has a network identity, like an IP address, used to communicate with other nodes. We assume that each client node can obtain the identities of the nodes providing the storage service. However, client nodes may not know about each other. As a result, two or more client nodes may issue storage operations concurrently. Most likely, those operations are on different locations most of the time or always (e.g., [5]). However, in some rare cases, two concurrent operations may try to access the same data. In those cases, the result should not be garbage.

Client and storage nodes are subject to fail-stop failures [10], which causes a node to halt its execution, and the node's halted state can be detected by other nodes if necessary. If a storage node fails, it may never recover, in which case the data that it stored is lost. We assume that failures are not extremely frequent, and if they occur, it is acceptable for the system performance to temporarily degrade a little.

Scheme	AJX-par	AJX-bcast	AJX-ser	FAB	GWGR	notes
min r/w granularity	1 block	1 block	1 block	1 block	k blocks	k determined by erasure code
read lat. (round trips)	1	1	1	1	1	
write lat. (round trips)	2	2	$p+1$	2	2	
# msgs for read	2	2	2	$2k$	$2n$	
# msgs for write	$2(p+1)$	$p+3$	$2(p+1)$	$4n$	$4n$	p much smaller than n
read bandwidth	B	B	B	B	nB	smaller is better
write bandwidth	$(p+2)B$	$3B$	$(p+2)B$	$(2n+1)B$	nB	(B = block size)

Fig. 1. Performance comparison in most likely (failure-free) executions using k-of-n erasure code ($p = n - k$). AJX-* are the protocols in this paper: -par uses parallel updates, -bcast uses broadcast (if available), and -ser uses serial updates (cf Section 4).

Our goal is to use erasure codes to provide redundancy to protect stored data against the above failures, while keeping space overhead small. Our goal is *not* to use redundancy to increase read throughput—this goal is often at odds with having small space overhead and supporting concurrent updates with strong consistency. We would like to support a wide variety of k-of-n erasure codes, especially highly efficient ones where n and k are large, but the overhead $n - k$ is small.

We also want to hide from applications the intrinsics of how erasure codes are used. Target applications include operating systems, databases, distributed file servers, or other higher-level services that require block storage. These applications access data through a block interface that support read-block and write-block operations. We prefer that the block size be one of the values commonly used; in fact, we prefer that all peculiarities of erasure codes be hidden from applications. However, these peculiarities may be known by some low-level module running at the client nodes below the application level.

3. Design and Algorithms

While designing our scheme, we chose to follow three well-known principles. *Shift functionality to clients:* Client nodes do active work, while storage nodes are thin, passive servers. This choice tends to provide better scalability, simplify crash recovery, and decrease the cost of adding more storage nodes to grow storage. *Optimize for common cases, simplify rare cases:* We avoid expensive mechanisms like locks or two-phase commits, in the common cases when there are no failures. In rare cases when failures do occur, we simplify the design using strong coordination via locks. *Hide intrinsics of mechanism being implemented:* The choice of erasure code does not affect the service interface provided to applications. For example, larger erasure-code stripes does not require applications to use larger block sizes.

3.1. Consistency

In the presence of concurrency and failures, we provide a reasonably strong consistency guarantee—the same as provided by *regular registers* [11] generalized to multiple writers [12]. Roughly speaking, it ensures that a read never returns a value that was never written, or a value that was overwritten by another write. If a write is concurrent with a read, the read may return the value of the write or the previously written value. If multiple writes are concurrent with a read, the read may return the value of any of the writes or the previously written value.

3.2. Organization

Our scheme is physically organized in two parts: (1) storage nodes are configured to serve simple requests from client nodes, and (2) client nodes orchestrate the storage nodes to store, retrieve, and recover data. Logically, the scheme has four components: (1) failure detection and node remap, (2) read and write algorithm, (3) recovery algorithm, and (4) garbage collection algorithm. We cover these components in later sections.

3.3. Brief primer on erasure codes

Roughly speaking, a k-of-n systematic maximum distance separable (MDS) erasure code [2] takes k *data blocks* and produces $n - k$ *redundant blocks*, such that any subset of k blocks (data or redundant or mixed) can reconstruct the k data blocks. We consider codes where the redundant blocks are linear combinations of the data blocks. We call *stripe* the combination of the k data blocks and $n - k$ redundant blocks.

For example, if a and b are data blocks, then we could produce two redundant blocks $a+b$ and $a-b$.[†] Given a stripe consisting of the four blocks $(a, b, a+b, a-b)$, any subset of two blocks can reconstruct a and b. For instance, given $a+b$ and b, we can obtain a by subtracting b from $a+b$. Therefore, we have a 2-of-4 erasure code, which can tolerate the loss of any 2 blocks in the stripe. Note how this is more powerful than 2-way replication with the same space overhead: if we simply replicate a and b, we get (a, b, a, b); if we later lose both replicas of a, we cannot reconstruct a.

More technically if $b_1, \ldots, b_k$ are data blocks then, in a k-of-n code, each redundant block $b_{k+1}, \ldots, b_n$ is given by $b_j = \sum_{i=1}^{k} \alpha_{ji} b_i$ for $j = k+1, \ldots, n$, where α_{ji} are carefully chosen constants, and arithmetic is over some finite field, usually $\mathbf{GF}(2^h)$.

3.4. Challenges of erasure-coded distributed storage

There are two main reasons why known solutions for replicated storage cannot be used with erasure codes: (1) erasure codes couple together *different* blocks of data, while replicated storage only couples together replicas; and (2) divergence of erasure-coded data is harder to detect and correct than divergence of copies in replicated storage.

To illustrate the challenges, suppose that we use the 2-of-4 code of Section 3.3 to store blocks a and b: each of $(a, b, a+b, a-b)$ are each kept in a separate storage node. Now suppose client node c_1 wishes to change a to c, while another client node c_2 wishes to change b to d. Here, the updates are to different data, but because the erasure code couples together c and d, some care is needed with concurrency. The end result must be $(c, d, c+d, c-d)$, but how do we keep c_1 and c_2 from clashing? This is easy if we use locks: c_1 locks all four blocks, reads b, and then overwrites the blocks; and c_2 proceeds analogously. But locks are very expensive, and we want to avoid using them. Moreover, even with locks, if c_1 fails before completion then

[†]Technically, in this case $+$ and $-$ must be taken over a field with characteristic $\neq 2$.

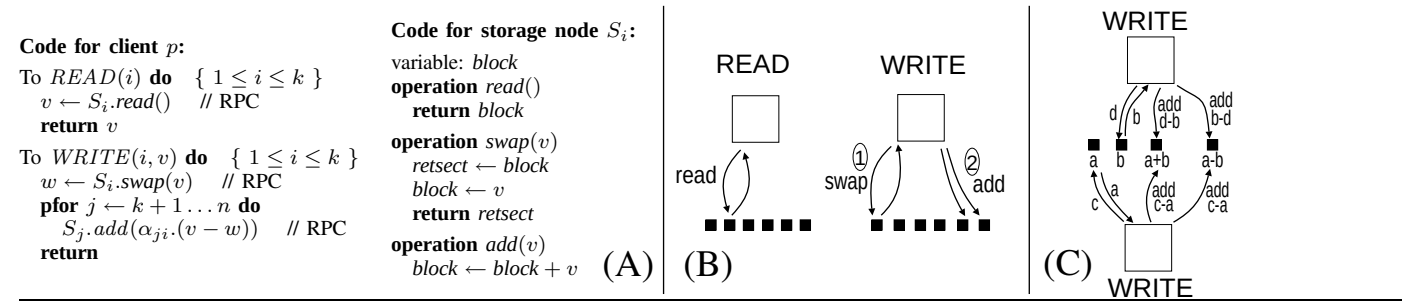


Fig. 3. Simplified algorithm: (A) code, (B) depiction, (C) example of concurrent writes preserving consistency of erasure code without any client coordination.

we can end up with $(c, b, c+b, a-b)$. Here, $a-b$ is an inconsistency. If two storage nodes fail, this inconsistency not only prevents recovery of correct data, but it may be undetectable. For example, if storage nodes 2 and 3 fail, we get $(c, -, -, a-b)$. This configuration is completely consistent with $c - a + b$ being previously stored in crashed node 2, and so it cannot be detected.

3.5. Failure detection and node remap

In our scheme, the failure of a storage node is detected when a client tries to access the node (we also allow to use periodic pings from some monitoring facility). This client then starts an expensive operation to reconstruct the lost data, which may need to be placed in a new storage node, if the failed one has not recovered (and it may never recovery). In those cases, we assume that a fresh replacement storage node is available, and there is some mechanism—like a directory service—to direct clients to this new node: clients simply access some logical node, which gets remapped on failures. The storage node has a flag indicating whether its data is valid, or just some uninitialized garbage.

3.6. Simple algorithm

Fig. 3 shows a simplified version of our algorithm that shows a core idea for the full algorithm. The simplified algorithm merely keeps data in n nodes consistent with a k-of-n erasure code, without tolerating failures. To read and write blocks $1, \ldots, k$ in a stripe, a client p communicates with storage nodes $S_1, \ldots, S_n$ via remote procedure calls. Storage nodes $S_1, \ldots, S_k$ keep the data blocks, while nodes $S_{k+1}, \ldots, S_n$ keep redundant blocks according to the erasure code. To *READ* block i ($i \le k$), p simply calls operation "read" on node S_i. To *WRITE* v to block i ($i \le k$), p swaps v into S_i, obtaining the old content w, and then adds $\alpha_{ji}.(v - w)$ to each redundant block in S_j for $j = k+1, \ldots, n$, where α_{ji} are the erasure code coefficients. The **pfor** is a parallel-for, whose iterations may be executed in parallel; after the **pfor**, the execution merges back.

What is interesting about this algorithm is that it keeps the erasure code consistent even if multiple clients write in parallel, regardless of how execution interleaves, even if both clients are trying to change the redundant blocks simultaneously. (This is not obvious; see Fig. 3 (C) for an example using the sample erasure code of Section 3.3.) And it does so without any synchronization via locks or two-phase commits.

3.7. Full algorithm: read and write

We now explain the full algorithm. Figure 4 shows the code for reading. In failure-free cases, it is very similar to the simple algorithm of Section 3.6. When a storage node fails and a new node is remapped (cf Section 3.5), the new node starts with *opmode* = INIT, indicating its data is initialized garbage. If a client tries to read from such a node, the read fails by returning $\perp$, and the client invokes the recovery procedure (Section 3.8) if the block is not locked (*lmode* $\in \{$UNL, EXP$\}$). If the block is locked, another client is already executing recovery.

Figure 5 gives the algorithm for writing data. When there are no failures and no clients *WRITE* to the same block simultaneously, the algorithm behaves like the simple algorithm of Section 3.6: To *WRITE*, client p first invokes *swap* (line 3), which returns *blk* $\neq \perp$. Then p invokes *add* on each redundant storage node (line 10). The *add*s all succeed, and so D is set to $\{i, k+1, \ldots, n\}$ (line 11), $Retry$ is set to $\emptyset$ (line 12), and T is set to $\emptyset$ (line 20), which causes p to finish the loops. The English comments in the figure provide a walk-through of the code, and to avoid repetition, we only explain here the higher level mechanisms. The basic idea to deal with storage node failures is for a client to invoke a recovery procedure, and later retry its *WRITE* or *READ* operation. More precisely, when storage node S_i fails, the remapped node (cf Section 3.5) starts out with *opmode* = INIT and *lmode* = UNL. When p invokes *swap* on S_i, *swap* fails, and p starts the recovery procedure. Recovery reads data from all storage nodes and uses the erasure code to reconstruct the lost data. Most of the complication in the write algorithm is to deal with concurrent online recovery by another client. We come back to these topics in Section 3.8.

Concurrent writes to the same block. To guarantee recoverability, the algorithm ensures that if clients p and q *WRITE* to the *same* block, they apply *swap* and *add*s in the same order at all storage nodes. This ordering is ensured as follows: a *swap* operation returns to the caller p an identifier *otid* for the previous *WRITE*; p then piggybacks *otid* to the *add* operations on redundant blocks; upon receive an add, a storage node checks if it previously saw *otid* (*otid* $\in$ *recentlist*); if not, the storage node rejects the *add* and returns a special ORDER status code, which tells p to retry later. If the client executing the previous *WRITE* crashes, then p may retry many times without success. After a certain number of attempts, p starts the recovery procedure. Note that concurrent writes to the same block are very rare in most systems (e.g., [5]).

3.8. Recovery algorithm

The basic idea for data recovery is to read all blocks from the storage nodes, decode them using the erasure code, and write back the results. The main issues are the following:

- Blocks may all be inconsistent with each other, due to outstanding *WRITES* and failed storage nodes with random blocks. Mechanisms are needed to (A) know when a group of blocks are consistent, i.e., they yield correct data when used for reconstruction, and (B) ensure

Code for client p:

To $READ(i)$ **do** // $1 \leq i \leq k$
1 $\langle v, lmode \rangle \leftarrow S_i.read()$
2 **while** $v = \perp$ **do**
3 **if** *lmode* $\in \{\text{UNL}, \text{EXP}\}$
4 **then** *start_recovery*() // start recovery procedure
5 $\langle v, lmode \rangle \leftarrow S_i.read()$ // retry read
6 **return** v

Code for storage node S_i:

Global variables:
7 *block*, initially 0, after fail-remap *random* // block content
8 *opmode* $\in \{\text{NORM}, \text{RECONS}, \text{INIT}\}$, initially NORM, after fail-remap INIT
9 // NORM: valid data in block; INIT: invalid data; RECONS: limbo
10 *lmode* $\in \{\text{UNL}, \text{L0}, \text{L1}, \text{EXP}\}$, initially UNL, after fail-remap UNL
11 // UNL: block unlocked; L0, L1: partial or full lock; EXP: expired lock

operation *read*()
12 **if** *opmode* $\neq$ NORM or *lmode* $\neq$ UNL
13 **then return** $\langle block : \perp, lmode \rangle$
14 **else return** $\langle block : block, lmode \rangle$

Fig. 4. Full algorithm for reading data.

that some group with at least k blocks is or eventually becomes consistent, where k is the number required by the erasure code.

- If a client p crashes while executing recovery, recovery must be completable by another client.
- A *WRITE* concurrent with recovery may garble the redundant blocks after recovery completes.

We now explain how we address the above issues.

Recent list. To know if a group of blocks are consistent with each other, storage nodes keep a list with the identifiers of past *WRITES* that have modified data in the storage node (the list is periodically garbage collected; see Section 3.9). More precisely, when a client p starts a *WRITE*, it picks a unique identifier *tid* for the *WRITE*. The *tid* is piggybacked on *swap* and *add* requests and, when a storage node receives one such request, it stores the identifier in the node's *recentlist* variable. The recovery procedure reads the *recentlist* from nodes to determine which blocks have been updated consistently.

The basic recovery procedure. Recovery can be executed by any client p, and it has three phases. In phase (1), p acquires locks at each storage node. These nodes maintain the lock state in their local variable *lmode*: *lmode* = UNL allows *swap* and *add* operations, while *lmode* = L1 will reject them. Locks serve two purposes: (i) they "freeze" the data in storage nodes and (ii) they prevent different clients from concurrently executing recovery. To avoid deadlocks, locks are acquired in order, but other standard mechanisms can be used, like retrying after some exponential back-off.[‡]

In phase (2), p reads the contents and states of all storage nodes (line 7) and checks if there are $k + slack$ blocks consistent with each other, where k is the number of blocks needed by the erasure code, and *slack* is explained below. If there are not, p "weakens" the lock on the redundant storage nodes, by setting their *lmode* = L0: in this mode, a node allows *add*s to execute, but the node remains otherwise locked. The intuition here is that p wants outstanding *WRITES* to complete their *add*s so that blocks become consistent. With the proper bounds on failures, p will eventually find a large enough consistent set of blocks. Next p tries to change back the *lmode* of nodes to L1 (full lock mode) before further *add*s occur (line 19).[§] If p does not succeed, p restarts the search for consistent blocks. (Note that p will eventually succeed because *swap*s are blocked, so new *WRITES* will not issue *add*s.) Else p sets the nodes' *opmode* variable to RECONS (explained below).

[‡]Lines 4–6 in the algorithm are for storage nodes that fail while locked, losing their locked state.

[§]The reason is that additional *add*s may cause a *WRITE* to complete, and so the recovered contents must include the effects of such *WRITE*.

In phase (3), (a) p uses the found consistent blocks to reconstruct data through the erasure code, (b) p writes the recovered data to the storage nodes, (c) p changes nodes' *opmode* to NORM (normal mode), and (d) p unlocks the nodes.

Epochs. Roughly speaking, an *epoch* is the period between two recoveries. A *WRITE* whose *swap* executes in one epoch should not let its *add*s execute in later epochs because recovery already leaves all blocks consistent. Thus, (a) *swap*s return an epoch number, (b) recovery increments the epoch number, (c) p piggybacks the *swap*'s epoch into *add*s, and (d) storage nodes reject *add*s from previous epochs.

Crashes during recovery. If p crashes during recovery, nodes that are locked (with *lmode* $\in \{\text{L0}, \text{L1}\}$) will "expire" their locks setting *lmode* = EXP (line 34). If another client q sees a node in this lock mode, q starts recovery. If p crashed before ending phase (2), the data in storage nodes have not been changed, so re-recovery by q is safe. Else p has set the nodes' *opmode* to RECONS; when q sees that, it skips phase (2) and, in phase (3), q does exactly what p would have done (q gets the set of consistent blocks used by p by reading the nodes' variable *recons_set*). The *slack* variable mentioned before guarantees that q can still find k consistent blocks, despite further storage node failures.

3.9. Garbage collection algorithm

As explained above, storage nodes keep a list *recentlist* of the *tid*s of past writes. To garbage collect this list, we use a two-phase algorithm to handle client crashes. In phase 2, all *tid*s whose write has completed are moved from list *recentlist* to list *oldlist*. In phase 1, *tid*s from *oldlist* are discarded. If the client crashes, the *recentlist* and *oldlist* of different storage nodes may end up different. This is not a problem: when using these lists to determine if a set of blocks is consistent, the client knows that if *tid* is in some *oldlist* of any node, then the write has occurred at all nodes. See function *find_consistent* in Figure 6 for more details.

The mechanism to order *WRITES* to the same block needs to be adjusted to work with garbage collection, as follows. After p gets an ORDER status, rather than retrying the *add* immediately, p first checks if the *otid* has been garbage collected at the data storage node or any of the redundant storage node; if *otid* is no longer at one or more of these nodes, p knows that the *WRITE* for which it is waiting has completed, so p can ask the redundant storage node to *add* without checking for *otid*; else p retries the *add* after a while.

3.10. Monitoring mechanism to trigger recovery

If client crashes while writing, or a storage node crashes, the system is in a fragile state that tolerates one less failure than before. It is

Code for client p:

```
Global variable: seq, initially 0                                   // sequence number for unique transaction id (tid)
To WRITE(i, v) do
1  repeat
2    ntid ← ⟨seq, i, p⟩; seq ← seq + 1                              // obtain unique id
3    ⟨blk, epoch, otid, lmode⟩ ← S_i.swap(v, ntid)                  // swap new value into data block
4    while blk = ⊥ do                                               // error, data unavailable
5      if lmode ∈ {UNL, EXP} then start_recovery()                  // nobody running recovery, so we do it
6      ⟨blk, epoch, otid, lmode⟩ ← S_i.swap(v, ntid)                // try swap again
7    T ← {k + 1, ..., n}                                            // node where we want to apply add operation
8    D ← {i}                                                        // nodes done with update
9    while T ≠ ∅ and D ≠ ∅ do                                        // while there are nodes to update, and done nodes are still up
10     for each j ∈ T do r[j] ← S_j.add(α_ji.(v−blk), ntid, otid, epoch)   // perform add at nodes in T
11     D ← D ∪ {j ∈ T : r[j].status = OK}                           // successful nodes
12     Retry ← {j ∈ T : r[j].status = ORDER or r[j].lmode ∉ {UNL, L0}}// nodes to retry due to ordering or lock problems
13     if ∃j ∈ T : r[j].lmode = EXP or                              // if some node has expired lock or
         (r[j].opmode ≠ NORM and r[j].lmode = UNL) or               //   it is not in normal mode and unlocked or
         (r[j].status = ORDER and tired of looping)                 //   it has returned ORDER for too long
14     then start_recovery()                                        // then start recovery
15     if ∃j ∈ T : r[j].status = ORDER then                         // some node complained about ordering
16       pfor each j ∈ D do
17         s[j] ← S_j.checktid(ntid, otid)                          // check if otid has been garbage collected
18       if ∃j ∈ D : s[j] = GC then otid ← ⊥                       // yes, no need to check ordering any more
19       D ← D − {j ∈ D : s[j] = INIT}                              // remove crashed nodes from successful list
20     T ← Retry
21   for each j ∈ D do gc[j] ← gc[j] ∪ {ntid}                       // for garbage collection
22 until D = {i, k + 1, ..., n}                                     // repeat until all blocks have been updated
```

Code for storage node S_i:

```
Global variables:
23  epoch ∈ N, initially 0, after fail-remap 0                      // epoch number
24  recentlist ∈ set of ⟨tid, time⟩, initially ∅, after fail-remap ∅ // recent write list
25  oldlist ∈ set of ⟨tid, time⟩, initially ∅, after fail-remap ∅    // old write list
26  time, initially 0, after fail-remap 0                           // local time, auto incremented at some rate

operation swap(v, ntid)
27  if opmode ≠ NORM or lmode ≠ UNL                                 // if not normal opmode or locked
28  then return ⟨block : ⊥, epoch, ⊥, lmode⟩                        // return error
29  retblk ← block                                                  // do swap
30  block ← v
31  if recentlist = ∅ then otid ← ⊥                                 // no previous write
32  else otid ← tid in recentlist with largest time                 // find tid of previous write
33  recentlist ← recentlist ∪ {⟨ntid, time⟩}                        // record tid of this write
34  return ⟨block : retblk, epoch, otid, lmode⟩

function tids(tidtime_list)
35  return tid of entries in tidtime_list                           // return tids in a list

operation add(v, ntid, otid, e)
36  if opmode ≠ NORM or lmode ∉ {UNL, L0} or e < epoch              // if not normal opmode or locked or old epoch
37  then return ⟨status : ⊥, opmode, lmode⟩                         // return error
38  if otid ≠ ⊥ and otid ∉ tids(recentlist ∪ oldlist)               // if previous write did not occur yet
39  then return ⟨status : ORDER, opmode, lmode⟩                     // tell client
40  block ← block + v                                               // perform add
41  recentlist ← recentlist ∪ {⟨ntid, time⟩}                        // record tid of this add
42  return ⟨status : OK, opmode, lmode⟩

operation checktid(ntid, otid)
43  if ntid ∉ tids(recentlist) then return INIT                     // only occurs if node crashes
44  else if otid ∉ tids(recentlist) then return GC                  // previous write not yet performed
45  else return NOCHANGE                                            // all is fine
```

Fig. 5. Algorithm for writing data.

thus desirable to restore the system's resiliency by starting recovery. Clients do so upon stumbling on a problem, but that only happens if they try to read or write. Thus, it might be useful to have a monitoring mechanism executed periodically by some client to probe the system for failures, and trigger recovery if necessary. This can be done very efficiently: for each storage node S_i, the client simply checks if (1) S_i's *recentlist* has some old *tid*, indicating a started but unfinished write, or (2) S_i's *opmode* is INIT, indicating initialization after recovery. In those cases, the client starts the recovery algorithm, which restores full recoverability of the system. This mechanism even works if the threshold t_p of client failures was exceeded, as long as no storage nodes have crashed.

```
Code for client p:

Global variables: data[i] for i = 1, ..., n

procedure start_recovery()
1   if recover not yet running locally then fork recover()                    // if/then executed atomically

procedure recover()
2   for j ← 1 ... n do                                                        // phase 1 starts: try to lock all blocks
3     r[j] ← S_j.trylock(L1)
4     if r[j].status = ⊥ then                                                 // somebody else locked
5       pfor each ℓ ≤ j such that r[ℓ].status = OK do S_ℓ.setlock(r[ℓ].oldlmode)   // release lock
6       return
    // phase 2 starts: now we are running solo
7   for j ← 1 ... n do data[j] ← S_j.get_state()                              // read state from all storage nodes
8   if for some h, data[h].opmode = RECONS then                               // another client previously crashed during recovery?
9     cset ← data[h].recons_set − {j : data[j].opmode = INIT}                 // yes, pick up their recovery
10  else
11    cset ← find_consistent()                                                // find consistent set of blocks
12    slack ← t_d − |{j : data[j].opmode = INIT}|                             // t_d is the max number of storage node failures (cf Section 4)
13    while |cset| < k + slack do                                             // while consistent set not large enough
14      pfor j ← k + 1 ... n do S_j.setlock(L0)                               // partially release locks to allow add operations
15      while |cset| < k + slack do                                           // while consistent set not large enough
16        pfor j ← k + 1 ... n do data[j] ← S_j.get_state()                   // get new state
17        cset ← find_consistent()                                            // find consistent blocks
18        slack ← t_d − |{j : data[j].opmode = INIT}|
19      pfor j ← k + 1 ... n do list[j] ← S_j.getrecent(L1)                   // try to lock blocks before new adds occur
20      cset ← cset − {j : list[j] ≠ data[j].recentlist}
    // phase 3 starts: now nodes have lmode = L1 and opmode = RECONS, and data[j].block has data for all nodes j ∈ cset
21  blocks ← erasure_decode(data[∗].block, cset)                              // decode blocks to retrieve data
22  pfor j ← 1 ... n do epoch[j] ← S_j.reconstruct(cset, blocks[j])           // write recovered data
23  pfor j ← 1 ... n do S_j.finalize(max_a{epoch[a]} + 1)                     // bump epoch, release locks, change to normal opmode

function find_consistent()                                                    // finds a set of blocks consistent with erasure code
24  return a maximal set S such that
      (1) ∀i ∈ S : data[i].opmode = NORM,                                     // only non-crashed blocks
      (2) ∀ redundant blocks r, s ∈ S : f̂_S(r) = f̂_S(s), and
      (3) ∀ redundant blocks r ∈ S, ∀ data blocks j ∈ S : Ĥ_S(r, j) = f̂_S(j)
    where Ĥ_S(i, j) = {x ∈ f̂_S(i) such that x = ⟨∗, j, ∗⟩ },                  // tids in f̂_S(i) originated by j
          f̂_S(i) := tids(data[i].recentlist) − Ĝ_S,                         // tids in S_i's recentlist minus Ĝ_S
          Ĝ_S := ∪_{i∈S} tids(data[i].oldlist),                             // tids in some oldlist
          tids(list) is the set of tids of items in list.

Code for storage node S_i:

Global variables: lid, initially ⊥, after fail-remap ⊥                        // identity of client locking block
                  recons_set ∈ set of integers                                // saved set of consistent blocks for recovery
operation trylock(lm)                                                         // try to lock if not locked yet
25  if lmode ∈ {L0, L1} then return ⟨status : ⊥, lmode⟩                       // already locked
26  oldlmode ← lmode; ⟨lmode, lid⟩ ← ⟨lm, caller⟩; return ⟨status : OK, oldlmode⟩   // update lock mode and return old mode
operation setlock(lm)   ⟨lmode, lid⟩ ← ⟨lm, caller⟩                           // set lock mode
operation get_state()                                                         // get node's state for recovery
27  if opmode = NORM then blk ← block else blk ← ⊥                            // if opmode ≠ NORM then block has garbage
28  return ⟨opmode, recons_set, oldlist, recentlist, blk⟩                     // return state for recovery
operation getrecent(lm)   ⟨lmode, lid⟩ ← ⟨lm, caller⟩; return recentlist     // change lock mode and return recentlist
operation reconstruct(set, blk)                                               // recover block
29  ⟨opmode, recons_set⟩ ← ⟨RECONS, set⟩                                      // remember set of blocks used for reconstruction
30  block ← blk; return epoch                                                 // write block
operation finalize(ep)                                                        // finish recovery
31  epoch ← ep; ⟨recentlist, oldlist⟩ ← ⟨∅, ∅⟩                               // advance epoch and clean lists of tids
32  if opmode = RECONS then opmode ← NORM                                     // back to normal mode
33  lmode ← UNL                                                               // and unlock storage node
upon failure of lid when lmode ∈ {L0, L1} do                                  // lid is the client locking block
34  lmode ← EXP                                                               // expire lock
```

Fig. 6. Algorithm for recovery.

3.11. Optimizations for sequential I/O and throughput

To optimize sequential I/O, consecutive blocks are mapped to different storage nodes and different stripes, and the redundant blocks rotate with each stripe, thus avoiding bottlenecks. In this way, clients can pipeline sequential I/O and get great bandwidth. As sequential writes occur, a redundant block R of a storage node is updated multiple times. When using disks to store data, the storage node can postpone writing R to disk until after the node knows that the sequential writes will no longer affect R. This can be determined when the node sees a write for large enough logical block C. For

Code for client p:

```
task collect_garbage
1   repeat periodically while not executing WRITE or READ
2     pfor j ← 1...n do
3       repeat r[j] ← S_j.gc_old(old[j])
4       until r[j] = OK
5     pfor j ← 1...n do
6       repeat r[j] ← S_j.gc_recent(gc[j])
7       until r[j] = OK
8       old[j] ← gc[j]; gc[j] ← ∅
```

Code for storage node S_i:

```
operation gc_old(list)
    if opmode ≠ NORM or lmode ≠ UNL then return ⊥
    remove entries in oldlist with tid in list
    return OK

operation gc_recent(list)
    if opmode ≠ NORM or lmode ≠ UNL then return ⊥
    for each t ∈ list do
      if exists entry in recentlist with tid t then
        move entry from recentlist to oldlist
    return OK
```

Fig. 7. Algorithm for garbage collection.

extra performance, R can be laid out on disk so that it is close to C.

Another optimization when writing is to use broadcast to send *add* to update the redundant blocks, thus saving client bandwidth. For this to work, the storage nodes, not the client, must do the multiplication by α_{ji} in line 10 of Fig. 5; clients simply broadcast the new content subtracted by old content—the same data for all storage nodes.

4. Correctness and Maximum Number of Failures

For correctness, we assume that $k \geq 2$ (more than one storage node), and $n - k \leq k$ (redundant blocks do not outnumber data blocks). Let t_p and t_d be the maximum number of client and storage node failures.

Theorem 1: The algorithms of Section 3 are correct if $t_d \leq d_{\text{SERIAL}} = \lceil \frac{n-k}{t_p+1} - \frac{t_p}{2} \rceil$.

The algorithm in Figure 5 updates to redundant blocks *in series* (**for** loop in lines 10–11). For better performance, we can parallelize the update by replacing **for** with a parallel-for (**pfor**). Then, a common-case *WRITE* takes only one *swap* and one batch of parallel *add*s. The tradeoff is reduced fault resiliency, as stated below:

Theorem 2: With parallel adds, the algorithms of Section 3 are correct if $t_d \leq d_{\text{PARALLEL}} = \lceil \frac{n-k}{2^{t_p}} - \frac{t_p}{2} \rceil$.

Corollary 1: To tolerate t_p client failures and t_d storage node failures, we need δ redundant storage nodes where:

$\delta = 1 + (t_p + 1)(t_d + t_p/2 - 1)$ (original algorithm), or $\delta = 1 + 2^{t_p}(t_d + t_p/2 - 1)$ (parallel adds).

The latency ρ for common *WRITES* is $\rho = 1 + \delta$ (original algorithm) or $\rho = 2$ (parallel adds).

Due to space limitations, proofs are omitted. They are given in [9].

A hybrid scheme. By corollary 1, the parallel scheme has smaller latency for common *WRITES* but much lower tolerance for client or storage node failures. As a compromise, we can define a *hybrid parallel-serial* scheme, where we partition the set of redundant storage nodes into s groups $G_1, \ldots, G_s$, of size at most $r = \lceil \frac{n-k}{s} \rceil$, where *add*s within a group are in parallel, but groups are updated in series. That is, we replace the **for** in line 10 in Fig. 5 with

```
for h ← 1...s do
  pfor each j ∈ G_h ∩ M do
      r[j] ← S_j.add(α_ji.(v−blk), ntid, otid, epoch)
```

Theorem 3: With parallel-serial updates, the algorithms of Section 3 are correct if $t_d \leq d_{\text{SERIAL}}$ and $r \leq d_{\text{SERIAL}}$.

For the parallel-serial scheme to tolerate t_p client failures and t_d storage node failures, we need the same $\delta = 1 + (t_p + 1)(t_d + t_p/2 - 1)$ storage nodes as in the serial update case, but the latency ρ for common *WRITES* is $\rho = 1 + \lceil \delta / d_{\text{SERIAL}} \rceil$, potentially much lower for small values of t_p (when $t_p = 0$, $d_{\text{SERIAL}} = \delta$ and $\rho = 2$).

Resetting the number of failures. After recovery completes, if no additional processes or storage nodes fail during the recovery then the system is in a "clean" state, where it can tolerate *additional* t_p process crashes and t_d storage node failures.

5. Validation

For validation, we have implemented our protocol and instantiated a small system with 8 hosts, where we varied the role of a host per experiment between client and storage node. We also used simulation to study the behavior of larger systems.

5.1. Implementation

We implemented our protocol in C using RPC in user mode running over TCP. Storage and clients nodes are multi-threaded. The number of threads at the server limit the number of RPC calls that are served simultaneously; at the client, it limits the number of outstanding calls. We implemented Reed-Solomon codes using hand optimized code for field arithmetic.

We instantiated our implementation in a system with 8 nodes for varying number of clients and nodes, and various levels of redundancy. Nodes were 2.4GHz-2.8GHz Pentium 3 or 4 machines with 256MB-1024MB of memory and a low-end gigabit ethernet card (no jumbo-frame support). Inter-node latency is 50 μs as reported by ping, and inter-node network bandwidth is 500Mbits/s as reported by Netperf.

To separate disk performance from our results, we used RAM memory as the storage medium for data in all experiments. Our results for latency and throughput correspond to a system with disks in cases where data is cached or the I/O is sequential with prefetching.

5.2. Simulation

We study larger systems through simulation. In the simulation, nodes have limited bandwidth and computing power, and the network also has limited bandwidth. Each client has multiple threads, one for each outstanding RPC call; there is a processor to serve all threads. In each thread, each phase of the protocol allocates the processor and the node's network adapter for some time for an RPC call, thus causing latency and consuming node bandwidth. Once an RPC message is placed on the network, the message incurs latency and consumes network bandwidth. When an RPC call arrives at the storage nodes, it allocates the receiving node's network adapter for some time, incurring extra latency and consuming node bandwidth. To serve an RPC call, the storage node incurs some variable latency that depends on the RPC call. Returning from an RPC is similar to issuing an RPC. The simulator reports per-node and aggregate throughput for reads and writes.

We tuned our simulator using the real system to determine values for the delays to encode and decode blocks for the erasure code,

latencies for various operations on the storage node, network latency, and bandwidth of each node.

6. Results

Our goal is to answer the following questions: (1) Are k-of-n Reed-Solomon erasure codes fast enough for storage, including with large k and n? (2) What are the latency and throughput numbers for our system, and how do they vary with n, k, and the number of clients? (3) What is the impact of failures on the system? (4) How complicated are storage nodes, and how much memory does our protocol use?

6.1. Erasure code choice and performance

Fig. 8(a) shows the k-of-n Reed-Solomon codes that we chose for real (non-simulated) runs with 4-7 storage nodes, together with their failure resiliency and computation time. Failure resiliency indicates the maximum tolerated number of client and storage node failures, e.g., "1c1s, 0c2s" means it tolerates either 1 client crash and 1 storage crash or 0 client and 2 storage crashes. For computation time, Delta is the time for finite-field subtraction followed by multiplication of a 1KB block (at the client node), and Add is the time for finite-field addition of a 1KB block (at the storage node). Full encode and full decode are the time to encode and decode a full stripe, used in recovery. All times are very small, as we wrote carefully optimized erasure code functions that runs 10-20 times faster than textbook implementations.

Fig. 8(b) shows the computation time for erasure codes with larger n and k used in our simulations for a 1KB block. The full-encoding and -decoding times are close, so the graph only shows encoding. Times for Delta and Add are combined. With large k, the full de/encoding time becomes significant, but in the common executions our scheme only uses Deltas and Add computations, whose times remain approximately constant even for large k.

Fig. 8(c) shows how many client and storage node crashes we can tolerate with the k-of-n erasure codes used in our simulations; it depends only on $n - k$, not on n or k individually.

6.2. Throughput

Fig. 9(a) shows aggregate write throughput as we vary the number of outstanding requests of size 1KB each, with 2 clients. Note that (1) the curves start to flatten after 64 simultaneous requests per client, and (2) increasing the number k of data storage nodes does not improve performance much. This is because the client network bandwidth saturates. The numbers for read throughput (not shown) are typically 4-5 times higher than write throughput, but are otherwise similar.

Fig. 9(b) shows how write aggregate throughput increases with number of clients; read throughput is similar and thus omitted. The graph does not have all combinations of erasure code and clients because we are limited by 8 nodes. The slope of the curves decreases after 3 clients as the storage nodes' bandwidth starts to saturate. For k-of-n erasure codes with larger k, the sloper is higher because there is more aggregate storage node bandwidth.

Fig. 9(c) shows how write throughput decreases with the redundancy of the erasure code ($n-k$). The decrease happens because more redundancy means more data for a client to send, which consumes its bandwidth faster. The decrease is gentler when k is larger—which is consistent with our goal to support high-efficient erasure codes with large n and k, and small $n - k$.

Fig. 9(d) shows an experiment where two clients are reading and writing random blocks using a 3-of-5 erasure code. After 28 minutes, one of the storage nodes crashes, causing throughput to drop to 1/3 for both clients. As clients access unavailable blocks, they recover those blocks and throughput gradually increases until all blocks are recovered. In another experiment (not depicted), three clients are recovering the blocks of a crashed storage node sequentially. The aggregate recovery throughput is around 17 MB/s, and latency is around 22ms for a request with 16 blocks.

6.3. Latency

Computation, including finite-field arithmetic for the erasure codes, contributed to less than 5% of the latency for writes or reads; 95% of latency is due to communication delays, including network delays, and TCP and RPC overheads. The total latency for a 4-block write was less than 3ms for a 3-of-5 code. Note that the storage medium is memory, and so there is no disk latency. Had we been using disks with a latency of 10ms, it would dominate.

6.4. Protocol complexity

The implementation has around 5,500 lines of C code: 1,200 for the erasure code library, 2,000 for clients, 2,000 for storage nodes, and 250 for common thread control. The storage node's code consists of independent remote procedures invoked by the client, which we consider simple because there is little code interdependency.

6.5. Space overhead at storage nodes

The memory used by our protocol at the storage nodes is 10 bytes per block—a 1% overhead for 1KB blocks. We could reduce this to 6 bytes per block; by increasing the block size to 16KB, it would result in a 0.04% overhead. Thus, the space overhead beyond the erasure code redundancy is very small.

6.6. Results from simulation

We used simulation to study throughput for systems with more hosts than we have. We checked accuracy by simulating our real system, and found an error of at most 20%.

We considered many combinations of erasure codes with $n = 4 \ldots 32$ and $k = 2 \ldots 16$, and had $1 \ldots 64$ clients executing operations simultaneously. Figs. 10(a) and 10(b) show aggregate write and read throughput respectively as the number of clients varies. For write throughput, the slope of the curve decreases with higher redundancy $n-k$, and the maximum decreases as n decreases and $n-k$ decreases, as also shown in Fig. 10(c). For reads, the throughput does not depend on k, only on n, because reads do not involve the redundant nodes.

Fig. 10(d) shows (on a log-scale) the write performance of a modified protocol that uses broadcast optimization to update each redundant block, as described in Section 3.11. With this optimization, the throughput of 1 client running alone does not decrease as $n - k$ increases. With 64 clients running simultaneously, the aggregate throughput decreases with $n - k$ as the storage nodes' bandwidth saturates.

6.7. Evaluation summary

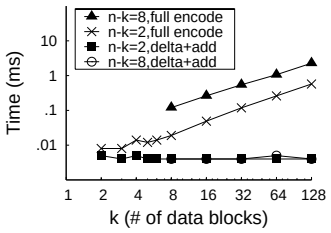
Our approach performs fairly well and offers many performance and resiliency modes. Failures disrupt a running system, but not to an extreme. Without broadcasts, a client's bandwidth becomes a bottleneck for write throughput if $n - k$ is large. Storage nodes are simple and keep little control data.

7. Conclusion

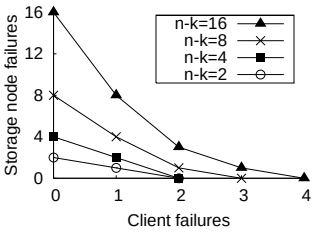
Erasure codes are powerful alternatives to replication for storage, as they provide better space efficiency and finer control over the redundancy level. However, they create complications due to complexity and cohesion of data, especially with concurrent updates and failures. Here, we propose a new protocol to address these complications. The protocol has features to make it broadly applicable, and its efficiency

Erasure Code	Failure Resiliency (c=client, s=storage)	Delta +Add (μs)	Full Encode (μs)	Full Decode (μs)
2-of-4	1c1s, 0c2s	5	8	8
3-of-5	1c1s, 0c2s	4	8	8
3-of-6	1c1s, 0c3s	4	11	12
4-of-6	1c1s, 0c2s	5	14	15
4-of-7	1c1s, 0c3s	4	15	15
5-of-7	1c1s, 0c2s	4	12	13

(a) Codes used in real runs

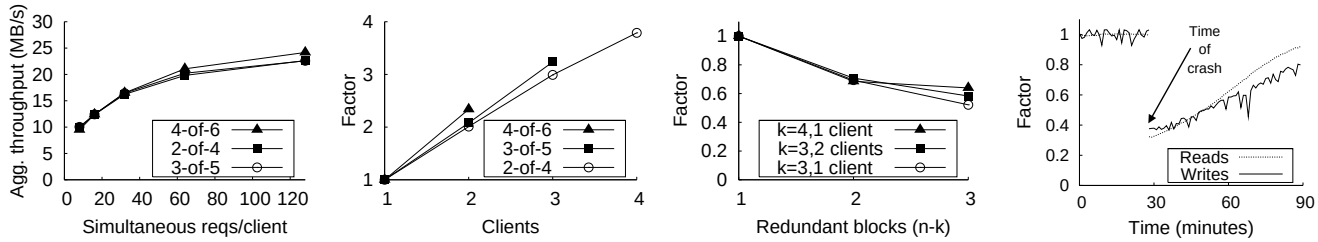


(b) Codes used in simulations



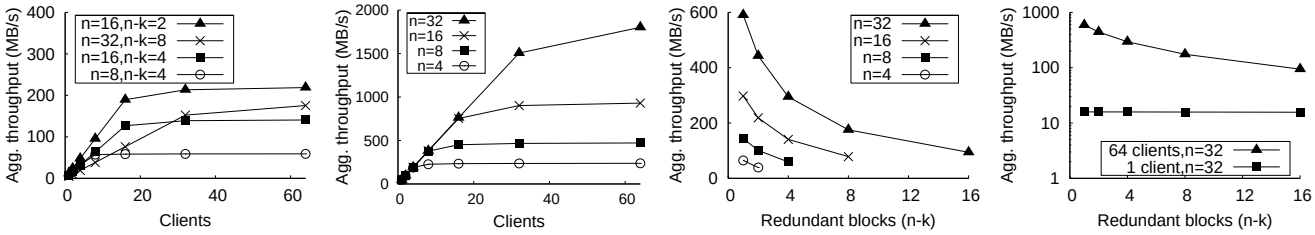
(c) Failure resiliency in simulations

Fig. 8. Performance and Fault-resilience of erasure codes.



(a) Write TP vs. #reqs (b) Write TP vs. #clients (c) Write TP vs. redundancy (d) TP when storage node crashes

Fig. 9. Results from real runs.



(a) Write TP vs. #clients (b) Read TP vs. #clients (c) Write TP vs. redundancy (d) Write TP using broadcast

Fig. 10. Results from simulations.

is reasonable as demonstrated by experiments. Our protocol allows the use of highly-efficient erasure codes, i.e., codes with large n and k, and small $n - k$. We envision a system that uses our protocol to build an industrial-strength distributed disk array with cheap adapters to connect disks to a network, powerful machines to serve as the array nodes, and highly-efficient erasure codes to tolerate multiple disk and array node crashes. External parties send requests for logical blocks to the array nodes; array nodes act as "clients" in our protocol, while the cheap adapters act as "storage nodes".

Acknowledgements. We are grateful to our shepherd Yair Amir, and to Mark Lillibridge, Janet Wiener and John Wilkes for suggestions that improved the paper. We thank Cheng Huang and Sinchan Mitra for providing computing resources.

References

[1] P. Corbett et. al., "Row-diagonal parity for double disk failure correction," in *Proceedings of FAST*, 2004.

[2] V. Pless, *Introduction to the Theory of Error-Correcting Codes*, Wiley-Interscience, 1998.

[3] J. Kubiatowicz et al., "Oceanstore: An architecture for global-scale persistent storage," in *Proceedings of ASPLOS*, 2000.

[4] F. Chang et al., "Myriad: Cost-effective Disaster Tolerance," in *Proceedings of FAST*, 2002.

[5] S. Frolund et. al., "A decentralized algorithm for erasure-coded virtual disks," in *Proceedings of DSN*, 2004.

[6] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter, "Efficient byzantine-tolerant erasure-coded storage," in *Proceedings of DSN*, 2004.

[7] Z. Zhang and Q. Lian, "Reperasure: Replication protocol using erasure-code in peer-to-peer storage," in *Proceedings of SRDS*, 2002.

[8] W. Litwin and T. Schwarz, "LH* RS : A high-availability scalable distributed data structure using reed solomon codes," in *Proceedings of SIGMOD*, 2000.

[9] M. K. Aguilera, R. Janakiraman, and L. Xu, "Efficient fault-tolerant distributed storage using erasure codes," Tech. Rep., Washington University in St. Louis, Feb 2004, Available at http://www.nisl.wustl.edu/Papers/Tech/aguilera04efficient.pdf.

[10] F. Schneider, "Byzantine generals in actions: implementing fail-stop processors," *ACM Transactions on Computer Systems*, vol. 2, no. 2, pp. 145–154, May 1984.

[11] L. Lamport, "On interprocess communication," *Distributed computing*, vol. 1, no. 1, pp. 77–101, 1986.

[12] C. Shao, E. Pierce, and Jennifer L. Welch, "Multi-writer consistency conditions for shared memory objects," in *Proceedings of ICDCS*, October 2003, pp. 106–120.