eAID: Elastic Asynchronous Information Dispersal with Post-Dissemination Pruning

Rithwik Kerur
University of California, Santa Barbara, CA, USA

Divyakant Agrawal
University of California, Santa Barbara, CA, USA

Dahlia Malkhi
University of California, Santa Barbara, CA, USA

Michael K. Reiter
Duke University, Durham, NC, USA

Amit Wieder
University of Colorado, Boulder, CO, USA

Abstract

Spreading and storing erasure-coded data effectively in distributed systems is challenging in practical settings. Consider a system of N nodes designed to tolerate up to $F < N/2$ failures. The dissemination of erasure-coded information is typically designed to complete only after receiving messages from N nodes, thereby preparing for the worst-case, but rare, scenario of F failures. In steady state, the remaining F nodes may in fact be healthy, but their resources are not counted. This leads to over-provisioning of storage for encoded data.

This paper introduces **eAID**, a novel *elastic information dispersal* algorithm that addresses this conundrum through a two-stage approach.

First, the core protocol estimates the actual number f of faulty nodes, rather than assuming the worst-case bound F. Dissemination completes quickly when messages are received from $(N - f)$ nodes, and more gradually when fewer nodes respond. This flexibility enables the system to tune storage efficiency against latency. Second, after initial dissemination completes, eAID continues monitoring for additional responses. As responses arrive from up to N nodes, the system prunes the information stored at responding nodes accordingly.

A key technique enabling this seamless elasticity is an agile encoding scheme that varies the number of disseminated fragments while keeping both fragment size and the recovery threshold $(F + 1)$ fixed. Not only does this enable varying the number of disseminated fragments on the fly, it also allows nodes to discard encoded fragments autonomously. Crucially, this is achieved without maintaining complex metadata, without requiring nodes to reconstruct or re-encode information, and without global coordination for storage decisions.

We demonstrate the practicality of eAID by integrating it with a replicated key-value store and evaluating it in network environments with unpredictable latencies. The results show that eAID improves overall performance while significantly reducing long-term storage consumption.

2012 ACM Subject Classification Computer systems organization → Dependable and fault-tolerant systems and networks; Mathematics of computing → Coding theory

Keywords and phrases Fault Tolerance, Erasure Coding, Information Dispersal, Consensus

Digital Object Identifier 10.4230/LIPIcs.DISC.2026.16

1 Introduction

The challenge of reliably storing and disseminating information in distributed systems is often a trade-off between fault tolerance and resource efficiency. While replication provides high availability, it incurs significant storage and communication overheads. Information



© Rithwik Kerur, Divyakant Agrawal, Dahlia Malkhi, Michael K. Reiter, and Amit Wieder;
licensed under Creative Commons License CC-BY 4.0



dispersal algorithms, pioneered by Rabin's IDA [44], offer a compelling alternative by splitting data into fragments using erasure codes (EC), allowing for reconstruction from a subset of fragments. In theory, IDA provides optimal redundancy. In practice, however, deploying it within critical infrastructure presents significant operational challenges.

The primary difficulty lies not in the coding theory, as robust EC schemes are well-established [32, 37, 45], but in handling unpredictable network latencies. In *asynchronous* settings, IDA methods must complete disseminating encoded fragments to N nodes in face of up to $F < N/2$ unresponsive ones. The problem is that the system cannot distinguish between a slow node and a crashed node.

To address this challenge, prior IDA-based systems have proposed several adaptive dissemination strategies: falling back to full replication upon timer expiration, as in RS-Paxos [39] and CRaft [51]; re-sharing "endangered" fragments to responsive nodes, as in OceanStore [30] and HRaft [25]; and dynamically varying the encoding threshold, as in FlexRaft [52].

While these approaches improve dissemination under unstable network conditions, they suffer from two fundamental limitations. First, preserving optimal long-term storage cost is operationally burdensome. Temporary transmission delays during dissemination may cause the system to increase redundancy, leaving nodes with excess encoded data even after network conditions stabilize. Restoring storage efficiency then requires reconstructing and re-disseminating information, and typically relies on centralized coordination to determine storage decisions across nodes.

Second, these approaches complicate recovery. To support adaptive coding, they maintain per-entry metadata describing the specific coding configuration used for each entry, and recovery must therefore apply entry-specific reconstruction procedures and parameters.

In this paper, we introduce **eAID**, an elastic way to AID in Asynchronous Information Dispersal, addressing these shortcomings. The core idea in eAID is to employ an erasure code scheme that varies the **number** of disseminated fragments while keeping both fragment size and the recovery threshold $(F + 1)$ fixed. This technique fundamentally rethinks how erasure coding is managed in dynamic networks and enables two-prong, seamless elasticity:

1. The dissemination protocol estimates the actual number f of faulty nodes, rather than assuming the worst-case bound F. Dissemination in eAID tunes the number of disseminated fragments on the fly based on the estimate f. It completes quickly when messages are received from $(N - f)$ nodes, and more gradually when fewer nodes respond. This flexibility enables the system to tune storage efficiency against latency.
2. After initial dissemination completes, eAID continues monitoring for additional responses. As responses arrive from up to N nodes, the system prunes the information stored at responding nodes accordingly. Note that the eAID coding strategy allows nodes to discard encoded fragments autonomously. Crucially, this is achieved without maintaining complex metadata, without requiring nodes to reconstruct or re-encode information, and without global coordination for storage decisions. Thus, unlike prior works that focus exclusively on the dissemination phase, eAID is the first solution to incorporate *post-dissemination pruning*.

Concretely, this elasticity comes from a fixed $(k = (F + 1), m = (F + 1) \times (N - 1))$ code[1]. The leader draws from a large pool of distinct fragments and assigns more of them per node as its failure estimate grows, so that $F + 1$ fragments always survive among the

[1] k denotes the number of fragments required for reconstruction; m the count of extra fragments.

responsive nodes. When responses are slow, eAID can avoid a retransmission round by overestimating failures and sending a few extra fragments, then pruning the surplus as soon as acknowledgments arrive. This trades a little dissemination bandwidth, which is typically abundant, for lower latency, which is often critical, particularly for mission-critical data-center and cloud systems [13]. We detail these strategies in Section 4.

Because the recovery threshold is fixed, eAID can reconstruct every disseminated information item from any $F + 1$ fragments using the same procedure, regardless of how that item was disseminated or later pruned. Recovery is therefore uniform and needs none of the per-entry coding metadata that other schemes must need to track.

We integrate eAID into a key-value store and evaluate the resulting system, eAID-KV, under network settings with highly variable point-to-point latencies characteristic of shared cloud environments [18]. eAID matches or exceeds the low-latency dissemination of prior protocols with no metadata overhead and no re-encoding, while eAID-KV uniquely reclaims long-term storage as network health improves, incurring only a minimal and temporary storage increase during disruptions. By optimizing for the critical path first and storage second, the evaluation substantiates our motivation for creating a robust, and storage-efficient dissemination regime for a replicated key-value store.

2 Background

2.1 Information Dispersal

The information dispersal problem, originally formulated in Rabin's seminal IDA method [44], considers the following challenge: a leader has a message M of size B that it intends to disseminate among a set of N nodes, while providing both data availability and storage efficiency. Specifically, the dispersal must guarantee that despite up to $F < N/2$ nodes crashing, the message M can be successfully reconstructed from the information stored on nodes that haven't failed.

Rather than sending each of the N nodes a full replica of M, a key technique for storage efficiency is employing erasure codes. An erasure code is parameterized using two values, (k, m): it encodes a message M into $k + m$ code pieces referred to as *fragments*, such that any k fragments suffice to reconstruct M. In the encoded information dispersal approach, the leader encodes M via a (k, m) erasure code and disseminates fragment(s) to nodes. With a Reed-Solomon code [45], each fragment size is B/k, the total storage is $\frac{B}{k} \times (k + m)$, and the *storage blowup* over B is $\frac{(k+m)}{k}$. Other practical codes, such as Luby-Transform [32] or Fountain codes [7] have similar blowup characteristics.

In Rabin's IDA, the parameters (k, m) of the erasure code directly mirror the system parameters $N = 2F + 1$, with $k = F + 1, m = F$. With this parameterization, there are $k + m = 2F + 1 = N$ fragments, and a leader assigns exactly one fragment to each node. This assignment maintains data availability against up to F node failures with a factor of $(2F + 1)/(F + 1) \approx 2$ storage blowup.

2.2 Asynchronous Information Dispersal

Previous solutions deal with the key challenge of asynchrony in several ways. We first define the difference between a *responsive*, *unresponsive*, and *failed* node as follows: a responsive node is a node that responds to messages in a timely manner, i.e., before the leader's timeout expires. An unresponsive node is a node that is still alive, but due to bad network conditions does not respond before the timeout expires. Notably, an unresponsive node can later become

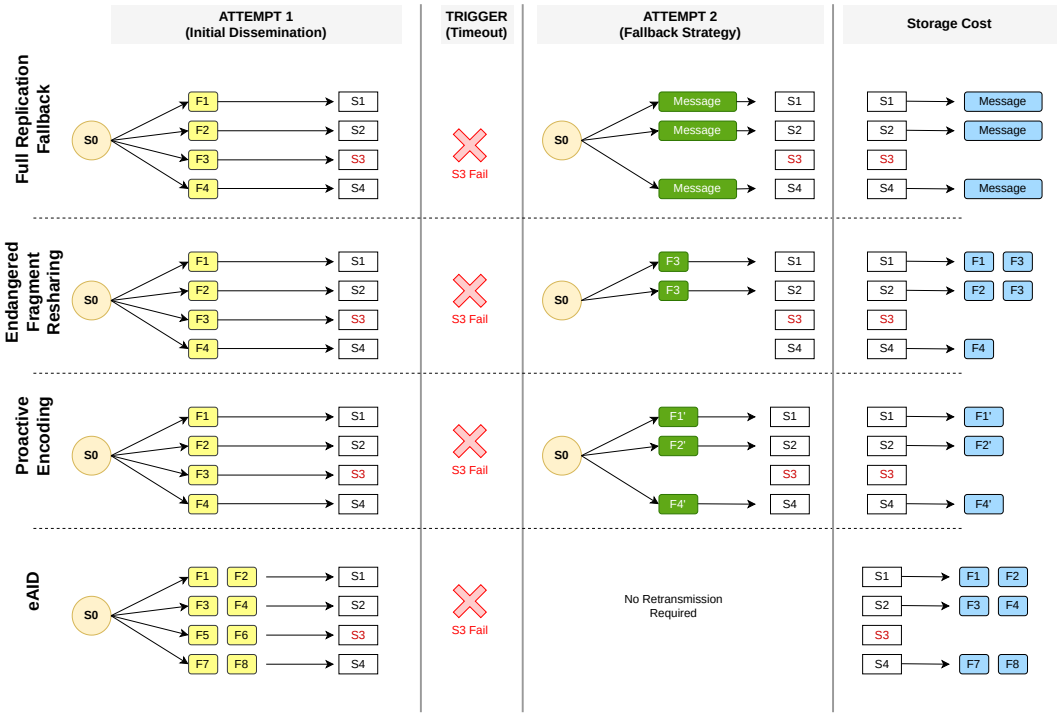


Figure 1 Scenario illustrating a single node failure (S3). In the Full Replication Fallback approach (top row), since the leader does not receive a response from S3, it reverts to full replication. In the Endangered Fragment Resharing approach (2nd row), the leader sends S3's fragment to $F = 2$ randomly picked nodes. In the Proactive Encoding approach (3rd row), the leader re-encodes information using a $(F + 1 - 1, F) = (2, 2)$ encoding scheme and disseminates new, larger fragments $\{F1', F2', F4'\}$ to the responsive nodes. In eAID, the leader initially disseminates a tunable number (2 here) of distinct fragments to each node and no retransmission is required.

responsive if network conditions improve. Lastly, a failed node is a node that has crashed and will not respond to any incoming messages. The difficulty in asynchrony is that a leader cannot distinguish between an unresponsive and failed node as from its perspective they both behave the same.

Full Replication Fallback. Figure 1 (top row) demonstrates an approach where a leader starts the replication of every message by broadcasting a single encoded fragment to each node. If there are any unresponsive nodes, i.e., if the leader doesn't hear from any node, the leader switches to full-copy replication. Note that in this case, a second round of retransmission is required and all nodes must store the entire message. Thus, with even a single unresponsive node, the latency doubles and the storage degenerates to full replication with $O(N)$ storage blowup. Some systems that implement this approach are CRaft [51] and RS-Paxos [39].

Endangered Fragment Resharing. Figure 1 (second row) illustrates an approach which starts the same way as Full Replication Fallback and addresses the abrupt performance degradation by reactively replenishing fragments across responsive nodes. Rather than defaulting immediately to full-copy replication, the leader replicates fragments assigned to unresponsive nodes on available nodes. More specifically, if the leader misses acknowledgments from f nodes, it sends copies of the fragments intended for the f missing nodes to F responsive

nodes. Consequently, some nodes hold f extra fragments. In many scenarios, resharing endangered fragments improves the storage required compared with falling back to full replication: rather than immediately degrading to full replication, it increases storage gradually, replicating only the fragments for which it misses acknowledgments. However, if many nodes are unresponsive, the responsive nodes must store all the fragments designated for them. For example, if $N/4$ of the nodes are unresponsive, this approach incurs $O(N)$ storage blowup. Several systems [25, 30, 4, 19, 3] adopt similar approaches.

Proactive Encoding. Another approach, which improves upon the two prior ones, proactively estimates the number of responsive nodes and adjusts the coding parameters based on this estimate. The leader adjusts the erasure code parameters and the assignment of fragments to nodes based on its current estimate. For example, if the leader estimates that one node may have failed, it adjusts the encoding scheme to $((F+1)-1, F)$. More generally, if the leader estimates that there are f failures, it uses a $(F+1-f, F)$ encoding scheme. Accordingly, each fragment size increases from $\frac{B}{(F+1)}$ to $\frac{B}{(F+1-f)}$, and only $(F+1-f)$ fragments are needed to recover the encoded Message. After re-encoding, the leader disperses the slightly larger fragment to responsive nodes. As shown in Figure 1 (third row), this approach may require a retransmission at the exact time a failure happens. Subsequent rounds do not require retransmissions as they make use of the existing coding parameters until the leader learns that the failure has been repaired. In general, this approach requires retransmission only if it receives fewer responses than it estimated using previous responses to dissemination. Although it lowers storage overhead, this approach ends up with a replicated log in which each entry may use a different coding scheme, complicating meta information maintenance and recovery. A Raft-based solution [52] adopts this approach.

In summary, all the above approaches incur a retransmission round if they encounter unresponsive nodes; they complicate the reconstruction process because different messages may use different encodings; and they do not address post-dissemination storage maintenance. In the rest of the paper, we detail how eAID overcomes these shortcomings and present an evaluation of its performance.

3 System Model

Asynchrony. We will consider the classical *asynchronous* model with $F < N/2$ benign failures. More specifically, the system consists of N nodes interconnected via point-to-point message passing, such that $F < N/2$ nodes can crash. We will assume static membership such that once the protocol has started, no nodes will join or leave the system, besides the nodes that fail. Once a node fails, it will not recover and start participating in the protocol again. Furthermore, if a node fails, we make no assumptions about the messages it sent prior to failing, and we make no assumptions about whether messages arrive if sent by a node that eventually crashes. We will assume that links between non-faulty nodes are reliable, and that messages sent between non-faulty nodes eventually arrive, after a finite but unbounded delay.

We formalize the information dispersal problem via the following API. $\textsc{Disperse}(M) \rightarrow \{\textsc{Done}\}$ is invoked at the leader to initiate dispersal of message M across the N nodes, returning Done when dispersal is complete. $\textsc{Prune}(id)$ is invoked at any non-failed node to potentially discard locally held fragments of the message identified by id.

A dispersal protocol is *correct* if it satisfies the following properties. **Dispersal Safety:** when $\textsc{Disperse}(M)$ returns Done, M remains reconstructable from the fragments held across nodes, despite up to F subsequent crash failures. **Pruning Safety:** after any invocation of

PRUNE(id), M remains reconstructable from the fragments held across nodes, despite up to F subsequent crash failures. **Dispersal Termination:** every invocation of DISPERSE(M) by a correct leader eventually returns DONE, provided the leader does not fail.

Performance measures. With these assumptions as our starting point, we shift our focus to the metrics we aim to optimize. Our primary goal is to enhance asynchronous performance by optimizing for two metrics: **storage cost** and **dispersal latency**.

We define **storage cost** as the aggregate size of storage used across all nodes to persist the fragments associated with a message. For a baseline, full replication – where a message of size B is sent to all N nodes – incurs a total storage cost of $O(N \cdot B)$. We also define **per-node storage** as the storage required on a responsive node in the system.

We define **dispersal latency** as the time between the leader initiating dissemination for a message and the moment the leader is certain that the message has been persisted on a sufficient number of nodes to guarantee recovery. We focus on this metric to isolate the performance of the information dispersal protocol, explicitly decoupling it from the baseline overhead of the replicated key-value store and client round-trip times.

4 eAID

4.1 Dissemination and Reconstruction

An eAID leader starts every dissemination attempt by encoding M into a **very large** number of fragments using a $(F+1, (F+1) \times (N-1))$ erasure coding scheme: creating $(F+1) \times N$ unique fragments such that any $F+1$ can reconstruct M. In practice, it need not encode using such a large code to begin with. It can generate only the fragments it needs for the initial dissemination and generate additional fragments as needed. We justify the choice of the erasure code in Appendix A. The leader employs an adaptive dissemination strategy, where it varies the number of fragments it sends to each node, from 1 to $F+1$. Algorithm 1 formalizes the leader side of this protocol, and Algorithm 2 specifies the corresponding storage node behavior. The eAID leader uses previous responses to estimate a responsive quorum. Based on this estimate, the leader determines how many fragments to assign to each node. For example, in steady state the leader may receive responses from all N nodes in the previous dissemination and send only one fragment to each node, similar to IDA. If the leader observes up to $N/4$ response omissions, it may increase the number of fragments per node by one, and send at least two fragments per node. More generally, if the leader estimates the responsive quorum to be $F+t$, it sends at least $\lceil \frac{(F+1)}{t} \rceil$ distinct fragments to each node (Algorithm 1, line 3 computing f_{per}). If the leader collects responses from $F+t$ nodes acknowledging that they received $\lceil \frac{(F+1)}{t} \rceil$ fragments each, there will always be $F+1$ fragments present in the system to reconstruct the original message M. If the leader does not collect $F+t$ acknowledgments, then it needs to retransmit additional fragments, as handled by line 16-20 in Algorithm 1. However, uniquely to eAID, the leader may prevent retransmission almost always if it initially sends each node a slightly higher number of fragments, say $\lceil \frac{(F+1)}{t'} \rceil$ where $t' < t$ (corresponding to the conservative margin $\delta = t - t'$ in Algorithm 1). As an example, in an 11-node setup, the leader can initially disseminate two fragments to each node. If one node is unresponsive, the leader does not have to retransmit, and is able to tolerate up to $N/4 \approx 3$ unresponsive nodes without retransmission. Notably if all nodes do respond, the leader can later inform nodes to prune a fragment. There are two advantages in sending a higher number of fragments initially: **1.** Lower the risk of retransmission (in the extreme case, zero risk when sending $F+1$ fragments per node). **2.** Faster completion of the dissemination

Algorithm 1 eAID IDA Dispersal (Leader).

State: $responsiveQuorum \leftarrow N$; $A[\cdot]$: ack sets per id; $sent[id, j], held[id, j]$: # fragments sent to / acked by j; f_{per}: current per-node fragment threshold

procedure DISPERSE(M)
  $\mathcal{F} \leftarrow$ ENCODE(M); $t \leftarrow responsiveQuorum - F$; $\boldsymbol{t' \leftarrow \max(1, t - \delta)}$; $f_{\text{per}} \leftarrow \lceil (F+1)/t' \rceil$
  Partition $\mathcal{F}$ into $S_1, \ldots, S_N$ with $|S_j| = F + 1$; ▷ Each node gets a pool of F+1 unique fragments
  $A[M.id] \leftarrow \emptyset$
  for each node j **do**
    Send MESSAGE($M.id, S_j[1 \ldots f_{\text{per}}]$) to j; $sent[M.id, j] \leftarrow f_{\text{per}}$; $held[M.id, j] \leftarrow 0$
  end for
  while $|A[M.id]| < F + t'$ **do** ▷ Continuously loop and adjust the number of fragments sent
    Start timer τ
    while $|A[M.id]| < F + t'$ **and** timer not expired **do**
      Wait for ACK(id, j, q); **if** $q > held[M.id, j]$ **then** $held[M.id, j] \leftarrow q$
      if $held[M.id, j] \geq f_{\text{per}}$ **then** $A[M.id] \leftarrow A[M.id] \cup \{j\}$
    end while
    if $|A[M.id]| \geq F + t'$ **then break**
    end if
    ▷ timeout: recalculate bounds and send additional fragments
    $\boldsymbol{t'_{\text{new}} \leftarrow \max(1, |A[M.id]| - F)}$; $f_{\text{new}} \leftarrow \lceil (F+1)/t'_{\text{new}} \rceil$
    for each node j **with** $f_{\text{new}} > sent[M.id, j]$ **do**
      Send MESSAGE($M.id, S_j[sent[M.id, j]{+}1 \ldots f_{\text{new}}]$) to j; $sent[M.id, j] \leftarrow f_{\text{new}}$
    end for
    $t' \leftarrow t'_{\text{new}}$; $f_{\text{per}} \leftarrow f_{\text{new}}$; $A[M.id] \leftarrow \{j \mid held[M.id, j] \geq f_{\text{per}}\}$ ▷ Recompute Ack set under new threshold
  end while
  $responsiveQuorum \leftarrow |A[M.id]|$; **return** DONE
end procedure

procedure ONLATEACK(id, j, q) ▷ after DISPERSE returns
  if $q > held[id, j]$ **then**
    $held[id, j] \leftarrow q$; **if** $j \notin A[id]$ **and** $held[id, j] \geq f_{\text{per}}$ **then** $A[id] \leftarrow A[id] \cup \{j\}$
    Send ACKUPDATE($id, |A[id]|$) to all nodes ▷ Update nodes so they can prune
  end if
end procedure

phase, since a leader must wait for a lower number of responses. If a higher number of nodes **do** respond, e.g., $F + t$ nodes acknowledge that they store say $\lceil \frac{(F+1)}{t'} \rceil$ fragments each, then the extra fragments will be immediately pruned via the PRUNE procedure in Algorithm 2, which each follower invokes opportunistically as the leader's threshold values advance (see the pruning section below). Bandwidth is typically abundant, while dispersal latency may be critical; hence, the leader can trade off a slightly higher communication cost at the outset to avoid a latency penalty later.

Algorithm 2 also specifies RETRIEVE, which reconstructs M from the fragments held across the cluster. It may be invoked by a storage node or directly by an external client. The caller broadcasts a request and accumulates fragments until $F + 1$ have arrived, then decodes. Because fragments assigned to distinct nodes are disjoint, every reply contributes new fragments toward the threshold. This is the property that prior adaptive schemes forfeit: where they must apply entry-specific reconstruction parameters, and therefore must first learn the encoding used for that entry, eAID applies one parameter-free algorithm to every entry in the log.

Figure 1 (bottom) illustrates eAID in the single-failure scenario we have used before. We illustrate the case where the leader initially assigns two fragments to each node. Even though there is a failure, due to its conservative approach, eAID does not need to retransmit fragments.

4.2 Post Dissemination Pruning

eAID's primary departure from prior works is that every node can prune the extra fragments post-dissemination by itself. Pruning does not require coordination or re-coding of information and can be carried autonomously for every node.

1. **Asynchronous Collection:** After waiting for a certain number of responses in the initial dissemination round, the leader continues to collect responses for M in the background.
2. **Responsive-quorum Updates:** As the leader gathers more responses after initial dissemination completes, it updates its estimate of the number of nodes that have stored fragments of M and sends this information to nodes in the system.
3. **Dynamic Storage Optimization:** Updates about the responsive quorum permit nodes to dynamically discard their fragments. For instance, if a node learns that $\frac{3N}{4}$ are storing fragments of M, it may safely discard all but two of its local fragments. In general, if a node determines that $F + t$ nodes are storing fragments, it only needs to retain $\lceil \frac{(F+1)}{t} \rceil$ fragments. Since there are no overlaps between the fragments that nodes store, it does not matter which fragments the node drops.

Through a combination of these three approaches, eAID uniquely adapts to unstable network conditions by: **1.** Only varying the number of fragments assigned per node; **2.** Allowing different numbers of fragments to be stored by different nodes: so long as $F + 1$ fragments can be collected in the entire system, M will remain available; **3.** Not requiring re-encoding of M; **4.** Not impacting the recovery computation, which always takes $F + 1$ fragments and reconstructs M in the same manner, oblivious to the assignment of fragments to nodes **5.** Not requiring maintaining meta-information per message M; **6.** Almost always avoiding retransmission rounds if overprovisioning.

Beyond quantifiable storage and latency gains, a primary design objective is protocol simplicity. Existing solutions often compromise this goal, suffering from either storage inefficiency or significant metadata overhead. By always requiring exactly $F + 1$ fragments to reconstruct, regardless of how many were disseminated or later pruned, reconstruction is reduced to a simple collection of any $F + 1$ fragments, eliminating the need to track fragment placement or manage evolving encoding states.

4.3 Safety and Liveness

▶ **Lemma 1** (Dispersal Safety). *When* Disperse(M) *returns* Done, *M is reconstructable despite up to F failures.*

Proof. Let $A := A[M.id]$ at the moment Disperse exits its main loop. The exit condition (line 14) guarantees $|A| \geq F + t'$, and membership in A requires $held[M.id, j] \geq f_{\text{per}}$ for every $j \in A$ (lines 12 and 20). Since each S_j is a disjoint block of the partition $\mathcal{F} = S_1 \cup \cdots \cup S_N$, fragments held by distinct nodes in A are disjoint.

We first verify the invariant $f_{\text{per}} \cdot t' \geq F + 1$ at every point where the leader checks the exit condition. Initially (line 3), $f_{\text{per}} = \lceil (F+1)/t' \rceil$ with $t' = \max(1, t - \delta) \geq 1$, so $f_{\text{per}} \cdot t' \geq F + 1$ by definition of the ceiling. After a timeout the leader recomputes $t'_{\text{new}} = \max(1, |A| - F)$ and $f_{\text{new}} = \lceil (F + 1)/t'_{\text{new}} \rceil$ (line 16) and commits these to t' and f_{per} (line 20), preserving the invariant.

After up to F failures, at least $|A| - F \geq t'$ nodes from A survive. Each surviving node holds at least f_{per} fragments, and these fragments are pairwise disjoint across nodes. The total number of surviving fragments is therefore

$$S \;\geq\; (|A| - F) \cdot f_{\text{per}} \;\geq\; t' \cdot f_{\text{per}} \;\geq\; F + 1,$$

which suffices to reconstruct M. ◀

Algorithm 2 eAID: Storage, Autonomous Pruning, and Retrieval.

```
1: State: store[id]: fragments held for message id; Q[id]: highest known # nodes holding id

2: procedure OnMessage(leader, id, fragments)          ▷ from leader's Message in Alg. 1
3:     store[id] ← store[id] ∪ fragments
4:     Send Ack(id, self, |store[id]|) to leader       ▷ ack carries current fragment count
5: end procedure

6: procedure OnAckUpdate(id, q)                        ▷ from leader's AckUpdate in Alg. 1
7:     if q > Q[id] then
8:         Q[id] ← q; Prune(id)
9:     end if
10: end procedure

11: procedure Prune(id)
12:     q ← Q[id];  r ← ⌈(F + 1)/(q − F)⌉              ▷ required fragments per node
13:     while |store[id]| > r do
14:         Discard any one fragment from store[id]
15:     end while
16: end procedure

17: procedure Retrieve(id)                             ▷ invoked by any client or non-failed node
18:     C ← ∅
19:     Send FragRequest(id, self) to all N nodes
20:     while |C| < F + 1 do
21:         Wait for FragReply(id, j, frags)
22:         C ← C ∪ frags                              ▷ fragments are disjoint across nodes
23:     end while
24:     return Decode(any F + 1 fragments of C)        ▷ identical procedure for every id
25: end procedure

26: procedure OnFragRequest(id, j)                     ▷ at a storage node
27:     if store[id] ≠ ∅ then
28:         Send FragReply(id, self, store[id]) to j
29:     end if
30: end procedure
```

▶ **Lemma 2** (Pruning Safety). *If every node prunes its fragments according to the rule*

$$r(|Q|) \;=\; \left\lceil \frac{F+1}{|Q|-F} \right\rceil,$$

then M remains reconstructable after any pattern of up to F failures, regardless of how nodes apply the rule using their local views of $|Q|$.

Proof. Let Q_{hi} be the set of nodes corresponding to the highest acknowledgment count, $|Q_{hi}|$, known to any individual node in the system.

Every surviving node u that has pruned its fragments did so autonomously based on some local view $|Q_u| \le |Q_{hi}|$, retaining $r(|Q_u|)$ fragments (lines 12–13). Because the pruning function $r(x) = \lceil \frac{F+1}{x-F} \rceil$ is monotonically non-increasing as x increases, a lower local view of the quorum results in a node retaining more fragments. Therefore, each surviving node retains at least the minimum required by the highest view: $r(|Q_{hi}|) = \lceil \frac{F+1}{|Q_{hi}|-F} \rceil$ fragments.

After up to F worst-case failures, at least $|Q_{hi}| - F$ nodes from the set Q_{hi} will survive. Since the initial dispersal guarantees that fragments held by different nodes are strictly disjoint, the total number of surviving fragments, S, is bounded below by the number of surviving nodes multiplied by the minimum number of fragments each retained:

$$S \;\ge\; (|Q_{hi}| - F) \cdot \left\lceil \frac{F+1}{|Q_{hi}|-F} \right\rceil \;\ge\; F + 1.$$

Because $S \ge F + 1$, the threshold to decode the $(F + 1, (F + 1) \times (N - 1))$ Reed-Solomon scheme is always met. Thus, M remains reconstructable under pruning. ◀

▶ **Lemma 3** (IDA Termination). *Every invocation of* DISPERSE(M) *by a correct leader eventually returns* DONE, *provided the leader remains correct throughout.*

Proof. In round k, let $f_{\text{per}}^{(k)}$ be the per-node fragment count and $t'^{(k)}$ the wait threshold. If round k collects $F + t'^{(k)}$ acknowledgments, DISPERSE returns by Lemma 1. Otherwise, a timeout occurs, and the leader recalculates the wait threshold as $t'^{(k+1)} = \max(1, |A^{(k)}| - F)$ and the new fragment requirement $f_{\text{per}}^{(k+1)} = \lceil (F+1)/t'^{(k+1)} \rceil$ (line 16). The leader then broadcasts additional fragments to *all* nodes j with $f_{\text{new}} > sent[M.id, j]$ (line 18), rather than restricting retransmissions to any earlier subset.

Because retransmissions are broadcast globally and each node's highest acknowledged fragment count $held[M.id, j]$ is only ever increased (line 11), never discarded across rounds, the total number of fragments successfully held by correct nodes is monotonically non-decreasing in k. The wait threshold t' is bounded below by 1, which upper-bounds the per-node fragment count at $f_{\text{per}} = F + 1$.

Eventually $t' = 1$, in which the termination condition reduces to $|A[M.id]| \geq F + 1$ and each node is required to hold all $F + 1$ fragments of its assigned set S_j. By assumption, there are at least $F + 1$ responsive nodes. Each such node, upon eventually receiving its $F + 1$ fragments, replies with an acknowledgment $q = F + 1$, which the leader eventually delivers. Thus $|A[M.id]|$ eventually reaches $F + 1$ and the outer **while** loop exits (line 14), so DISPERSE returns DONE. ◀

5 Use Case: Encoded information dispersal in a key-value store

To demonstrate the benefit of the eAID approach, we build an application using it. We choose a replicated key-value store backed by log replication for two reasons: they are a demanding target where both dispersal latency and long-term storage matter, and they are the setting in which prior IDA-integrated protocols [25, 51, 52] have been evaluated, enabling direct comparison.

Replicated key-value stores are the storage substrate behind databases [11], distributed file systems [42], and control planes [5]: a replicated log establishes the authoritative order of operations across nodes, and a key-value store applied on top serves the live data. With IoT [35] and AI workloads [27] pushing enterprise storage demand into the zettabyte[2] range [46], the on-disk footprint of these stores has become an increasing cost. Bandwidth spent during short-term dissemination is transient, while storage is paid for indefinitely, on every replica. Key-value stores are therefore a setting where eAID's post-dissemination pruning translates most directly into long-term operational savings, making them a natural target for evaluating the protocol's storage benefits in practice.

Integrating eAID into this setting requires starting from a log-replication core that consistently sequences updates to key-value entries across replicas, and replacing its internal information-dispersal mechanism with eAID-based erasure-coded dissemination. To build this implementation, which we call eAID-KV, we borrow code from an existing replicated key-value store system [52] and modify its dissemination layer accordingly.

In Section 7, we evaluate eAID-KV against several prior protocols that integrate IDA with log replication in the same setting [52, 25, 51].

[2] a zettabyte is 10^9 terabytes

6 Implementation

We implemented eAID-KV in C++ building upon the existing FlexRaft implementation [1] which contains approximately 6,000 lines of code and modified it by adding roughly 1,000 lines of code. We choose FlexRaft as a starting point because it too implements a KV store backed by log replication. To facilitate evaluation, we constructed a distributed key-value store backed by RocksDB v7.3.1 [16]. Architecturally, each node hosts an eAID-KV alongside the RocksDB state machine, utilizing a background thread to asynchronously apply committed entries. We provide further details of the integration of eAID in Appendix B.

Our implementation utilizes a leader-driven dissemination model, where the leader broadcasts network response information to all followers as this is the model typically used in a key-value store. These metrics are derived from data gathered via periodic heartbeats and standard message exchanges in Raft. While we focus on this leader-centric approach, alternative mechanisms such as all-to-all peer communication could be employed to allow nodes to independently assess network state. Such decentralized state acquisition would enable nodes to perform local storage optimizations without relying on the leader's view, a direction we reserve for future work. We provide pseudocode for eAID's integration with Log Replication in Appendix B.1.

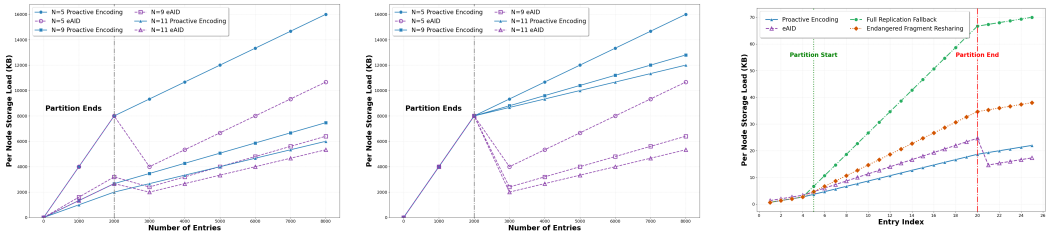
7 Evaluation

We evaluate eAID both as a standalone dispersal protocol (analytically) and as the engine of a key-value store, eAID-KV [28] (experimentally). Analytically, we study the dispersal protocol in isolation to separate its behavior from any application above it and to reproduce network-partition scenarios that are hard to stage in a live cluster. We compare eAID against three dispersal substrates: Full Replication Fallback, Endangered Fragment Resharing, and Proactive Encoding. Experimentally, we evaluate eAID-KV against FlexRaft, the only system that adopts the Proactive Encoding approach. Its codebase [1] provides a robust open-source implementation of this dispersal substrate integrated with a key-value store. For the experimental study, we build eAID-KV by modifying FlexRaft's dispersal layer to align with eAID, and use the unmodified FlexRaft as our baseline. This ensures any observed performance differences can be attributed to the dispersal protocol itself.

Through our evaluation we aim to demonstrate the following claims: **C1** eAID and eAID-KV maintain the same storage utilization as Proactive Encoding and FlexRaft, respectively, under normal-case network conditions; **C2** eAID and eAID-KV maintain better storage utilization than Proactive Encoding and FlexRaft, respectively, under network partitions; **C3** eAID maintains better dispersal latency than all previous approaches under unstable network conditions; **C4** eAID-KV maintains the same or better performance than FlexRaft in a distributed setup.

7.1 Experimental Setup

We conducted our experiments on a high-performance DigitalOcean Droplet equipped with 16 vCPUs and 64GB of RAM. We primarily focus on a cluster within a single physical machine to isolate the storage metrics from external network variance. Since our primary evaluation metric is storage utilization efficiency – which is a function of the consensus state rather than packet latency – colocating the nodes allows us to deterministically reproduce partition scenarios without the noise of non-deterministic network jitter. We also include performance evaluation of a distributed setup where we deploy five nodes with the same configuration in DigitalOcean's SFO3 region with a colocated client sending transactions. We include the results of this evaluation in Section 7.4



(a) Two partitioned nodes. **(b)** F partitioned nodes. **(c)** Zoom on partition; $N = 11$, $f = 2$.

Figure 2 Analytical evaluation of storage utilization under network partitions.

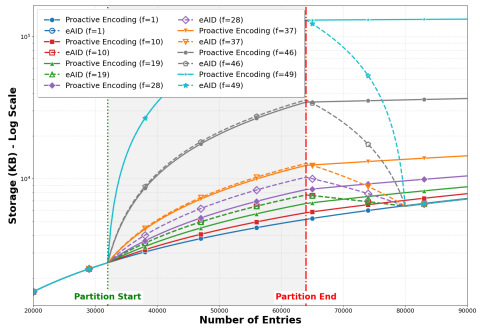
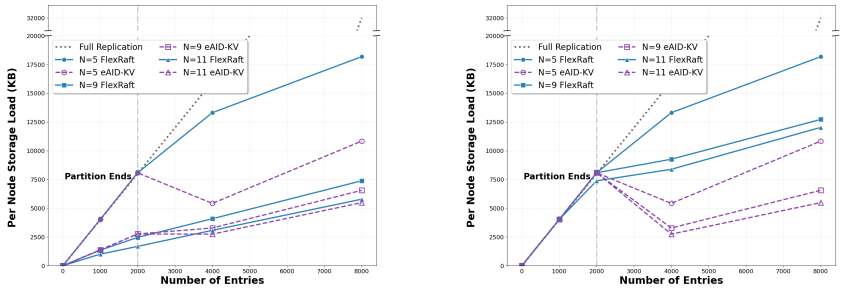


Figure 3 Storage cost for $N = 99$ with f partitioned nodes; the partition spans entries 32,000–64,000. Note the log-scale y-axis.

7.2 Storage Utilization

We first present an analytical evaluation of eAID, and verify our results with an experimental evaluation of eAID-KV. Throughout the evaluation, the x-axis reports the cumulative number of log entries appended, which serves as our notion of logical time: each entry carries a fixed 4 kb payload, and partitions are specified by the entry index at which they begin and end. Figure 2a depicts storage utilization when two nodes are partitioned from the network for increasing cluster sizes. The two nodes are partitioned until 2000 entries and don't respond to any incoming messages. Once the partition ends the nodes start responding to messages. Under these conditions, eAID performs as well as or slightly worse than Proactive Encoding until the partition ends. Once the partition ends, eAID has better performance as it is able to adapt and reduce the higher storage cost required during the partition. Initially, upon detecting an unresponsive node, Proactive Encoding interprets this as a node crash and adjusts its encoding strategy accordingly. In contrast, eAID assumes the node is simply unresponsive (asynchrony). By assuming the node is unresponsive, eAID incurs a higher storage overhead as it waits for the node to become responsive. Initially, this is a slight disadvantage as all nodes in eAID are storing more information than required. However, once the node rejoins the network, nodes can prune the extra storage incurred and reduce storage down to the optimal amount.

Figure 2b presents a similar scenario where F nodes are initially partitioned and rejoin the cluster after 2000 entries. Initially for entries $0 - 2000$, eAID exhibits the same storage utilization as Proactive Encoding as both systems need to store the entire entry to maintain data availability. With only $F + 1$ responsive nodes remaining, eAID ensures data availability by storing $F + 1$ fragments per entry. However, the system's ability to recover storage



(a) Two partitioned nodes. **(b)** F partitioned nodes.

Figure 4 Experimental evaluation of storage utilization for eAID-KV and FlexRaft under a network partition that recovers at 2000 entries. We also show the storage utilization of Full Replication.

is demonstrated once the partition heals. When the leader confirms the F missing nodes have returned and are responsive, eAID performs a retrospective optimization. The storage utilization sharply drops (visible at the $2000 \rightarrow 3000$ entry mark) as the system reduces the redundancy of prior entries from $F + 1$ fragments down to one fragment per node.

Figure 3 extends the analysis to $N = 99$ subject to an increasing number of partitioned nodes. A partition occurs between 32,000 entries and 64,000 entries. We observe that the same pattern holds with eAID consistently recovering to optimal storage post-partition while Proactive Encoding does not. The distinction is that Proactive Encoding cannot retroactively re-optimize storage after a partition without reconstruction and re-encoding, while eAID leverages the distinction between crashed and unresponsive nodes to recover the temporary overhead it incurs without any re-encoding or reconstruction.

Figure 2c shows a zoomed in version of a partition start and end utilizing a configuration of $N = 11$ and $f = 2$. By zooming into the graph we can see the slightly higher storage cost of eAID at the beginning, but see the same benefits of its post commit. This confirms Claim **C2**: while eAID may temporarily use more storage to handle partitions, it uniquely recovers that space and maintains better storage utilization than Proactive Encoding once the cluster stabilizes.

We verify that these analytical storage predictions hold for the deployed eAID-KV system. Figure 4 reports the corresponding empirical measurements from eAID-KV deployed against FlexRaft, under the same two-node and F-node partition scenarios. The empirical curves track the analytical predictions for eAID, confirming that the dispersal behavior survives integration into the key-value store.

7.3 Dispersal Latency of eAID (Analytical)

Figure 5 illustrates the simulated latency required for each protocol to output Done on an entry under unstable network conditions. We define latency as the total time it takes from when a leader initiates dissemination, to when it outputs done, excluding leader-election round trips and assuming a steady state leader. To model network instability, we draw the nodes response time from a normal distribution centered at 0.8 ms with a standard deviation of 0.15 ms. We set the leader timeout to be 2 standard deviations above the mean. For Proactive Encoding, when the leader does not receive all N responses before the timeout, we repeatedly re-sample until the number of responses meets or exceeds the previous sample, reflecting the incremental re-encoding mechanism it employs. We evaluate eAID using a

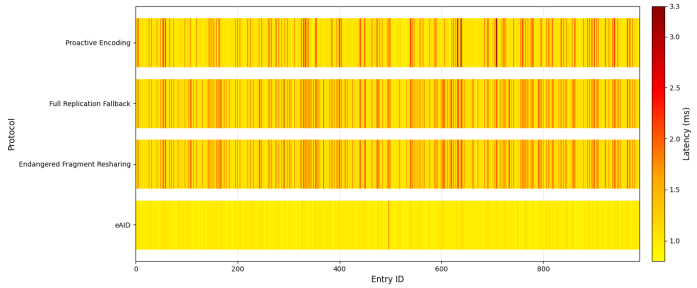


Figure 5 Analytical evaluation of network latency for Full Replication Fallback, Endangered Fragment Resharing, Proactive Encoding, and eAID during unstable network conditions where $N = 11$.

dissemination strategy where the leader initially disseminates a conservative two fragments per node. Under this dispersal strategy, the leader is able to tolerate up to $N/4$ missing responses without retransmission. We assume that entries are uncorrelated, and thus an entry at index 501, does not rely on the entry at index 500 to finish before it can complete.

The graph tracks performance across 1000 invocations of DISPERSE to the system, with the random latency distribution re-sampled for each entry. Under these circumstances, the baseline is ∼1 ms (yellow/orange) and everything above it (darker red) is retransmission overhead due to the leader not receiving enough responses and timing out. Note that due to the randomness employed, each entry's latency can vary slightly.

The Full Replication Fallback and Endangered Fragment Resharing approaches perform identically, requiring ∼2 ms for 22.5% of invocations and ∼1 ms for the remainder. Proactive Encoding performs slightly better, taking ∼2 ms for 14.6% of invocations, but its re-encoding approach occasionally triggers a third transmission round, pushing latency above 2 ms for 0.4% of invocations. By initially disseminating two fragments per node, eAID only requires ∼2 ms for less than 1% of invocations and ∼1 ms for the rest. This validates Claim **C3**: eAID outperforms the other approaches under unstable network conditions.

7.4 Key-Value Store over Log Replication (Experimental)

We deploy both eAID-KV and FlexRaft in a 5-node distributed setup and evaluate them on YCSB workloads [10] A, B, and C, which span the read/write spectrum. Figure 6 reports the average throughput over 5 runs as the number of concurrent clients increases from one to four.

Under the write-heavy YCSB-A workload (50% read / 50% write), eAID-KV consistently outperforms FlexRaft across all client counts, reaching ∼9.4 Mbps versus ∼8.5 Mbps at four clients. This gap stems from eAID-KV's overprovisioned dissemination: it sends extra fragments so it can commit with the fastest quorum and prunes the excess post-commit. Since FlexRaft only sends a single fragment, it must wait on every node to safely commit, which stalls reads as well as writes as reads cannot return values older than the latest committed write. As the workload shifts to read-dominant YCSB-B (95% read / 5% write), the gap narrows and both protocols converge near 40 Mbps at four clients. Under read-only YCSB-C, the two systems are indistinguishable, scaling from ∼17 Mbps with one client to ∼70 Mbps with four. This convergence is expected, as there are less writes to bound read progress.

We now evaluate the same setup with only sequential put operations. Since there are no reads to amplify the advantage eAID-KV offers, the performance is largely the same. Figure 7 decomposes end-to-end latency into its two components. The left panel shows end-to-end

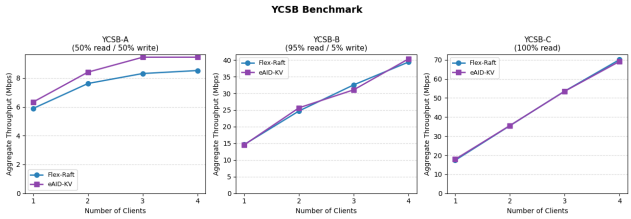


Figure 6 YCSB benchmark for FlexRaft vs. eAID-KV.

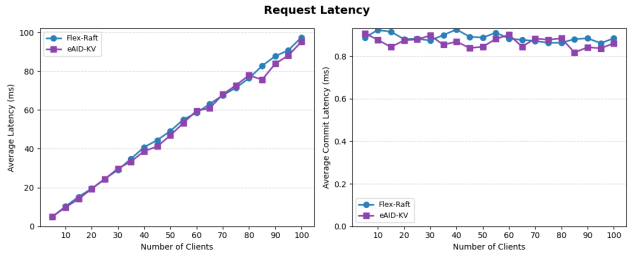


Figure 7 Distributed benchmark for FlexRaft vs. eAID-KV.

client latency, which is dominated by wide-area network round-trip time and steadily scales as we increase the number of clients. The right panel isolates the *commit latency* which represents the time spent in the consensus protocol. These results validate **C4** showing that the overhead of integrating our IDA approach with consensus does not affect performance in the steady state.

7.5 Summary

Our evaluation supports all four claims. eAID matches Proactive Encoding's storage utilization under normal network conditions (**C1**) and achieves lower storage post-partition through its autonomous pruning approach (**C2**). These results carry over to the deployed eAID-KV system, where the same trends hold experimentally as well. eAID also incurs optimal dispersal latency ∼99% of the time under unstable networks via its adaptive dissemination strategy (**C3**). Finally, eAID-KV matches FlexRaft's end-to-end performance in the steady state (**C4**), confirming that integrating eAID into a KV store imposes no observable overhead. A useful consequence of this design is that the leader can set aggressive commit timers without stalling on slow nodes: eAID-KV remains live, with storage utilization converging to the optimal state once slow nodes catch up.

8 Related Work

Replicated key-value stores rely on consensus protocols to maintain a consistent state across geographically dispersed nodes. Paxos [31] established the theoretical foundation for this field, powering systems such as Chubby [6] and Spanner [11]. To address the inherent performance bottlenecks of vanilla Paxos, numerous variants [2, 38, 9, 40, 33, 34] have been proposed. **Mencius** [33] utilizes a round-robin approach with rotating leaders to partition the log and balance network load. **EPaxos** [38] adopts a multi-leader design but can only provide a partial ordering (less strict than the total order that other alternatives provide), whereas **RingPaxos** [34] organizes nodes in a ring topology to exploit high-bandwidth multicast

and improve overall throughput. For wide-area deployments, **WPaxos** [2] employs multiple leaders to reduce geographic latency, and **DPaxos** [40] employs dynamic quorums that allow for small replication groups without requiring large, static leader election quorums. Additionally, **PigPaxos** [9] focuses on the communication layer, using in-network aggregation and piggybacking to decouple the leader's decision-making from message passing.

While the aforementioned works focus on structural optimizations, **RS-Paxos** [39] addresses the resource overhead of state machine replication. By integrating erasure coding directly into the Paxos protocol, RS-Paxos replaces traditional full-copy replication with coded fragments. This approach can reduce network and storage costs by over 50% and significantly improves write throughput by minimizing disk I/O. That said, RS-Paxos introduces a trade-off between fault tolerance and storage efficiency. To tolerate the maximum theoretical number of failures ($F < N/2$), it effectively reverts to full replication, employing erasure coding only when the system is configured to tolerate a strictly lower threshold. Finally, although early storage systems have utilized erasure codes to reduce storage [22], these systems do not integrate erasure coding with consensus.

Raft [41] is another widely adopted consensus algorithm designed for understandability, and it is functionally equivalent to Multi-Paxos. Recent research has introduced various optimizations to the original Raft protocol [8, 29, 49]. **ParallelRaft** [8] focuses on high concurrency and allows parallel replication of multiple entries as well as out of order execution for non-conflicting commands. To address hardware-level constraints, **HovercRaft** [29] leverages kernel-bypass techniques and in-network programmability to eliminate CPU and I/O bottlenecks. Finally, **RACS-SADL** [49] optimizes the performance by separating command dissemination from the core consensus logic.

Optimizing with Erasure Coding. Several recent protocols aim to reduce consensus/replication storage and communication cost via the use of erasure coding, including **CRaft** [51], **HRaft** [25], **FlexRaft** [52], and **Crossword** [21]. We discuss these works in detail in the Background section of the paper (Section 2). Briefly, all of them address only dissemination efficiency and do not support post-dissemination pruning (this has been mentioned as a future challenge [25]). They administer adjustments to the coding regime only by the leader; and most of them compound the recovery by varying coding regimes across different log entries. The exception is Crossword [21], a parallel independent work to eAID employing a $(F+1, (F+1) \times (N-1))$ erasure-code. However, Crossword aims to optimize bandwidth consumption, not storage, and additionally, like all previous works, has no post-dissemination adaptability. In contrast, eAID implements post-dissemination pruning, allows each node to locally and autonomously adjust the coding regime per log-entry, and applies a simple and uniform recovery algorithm across all log entries

Additionally, while we focus on crash fault tolerance in our protocol, there are numerous works that target Byzantine failures (e.g., [12, 14, 17, 36, 47, 26]).

9 Conclusion

We presented eAID, an elastic information dispersal algorithm that fixes the recovery threshold at $F + 1$ and varies only the number of fragments per node. eAID decouples dispersal completion from the final storage layout as it completes once enough responses arrive and prunes surplus fragments in the background, without coordination. eAID-KV, the result of integrating eAID into a replicated key-value store, matches or exceeds prior systems in the steady state while reclaiming storage after a partition. Two directions remain open: extending eAID to the Byzantine setting, and exploring further ways to reduce common-case latency.

References

1 ACS Storage Group. FlexRaft-Code. https://github.com/ACS-Storage-Group/FlexRaft-Code, 2024.

2 Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. Wpaxos: Wide area network flexible consensus. *IEEE Trans. Parallel Distrib. Syst.*, 31(1):211–223, January 2020. doi:10.1109/TPDS.2019.2929793.

3 Ross Anderson. The eternity service, December 2001.

4 Ranjita Bhagwan, Kiran Tati, Yu-Chung Cheng, Stefan Savage, and Geoffrey M. Voelker. Total recall: system support for automated availability management. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, page 25, USA, 2004. USENIX Association.

5 Marc Brooker, Tao Chen, and Fan Ping. Millions of tiny databases, 2020. URL: https://www.amazon.science/publications/millions-of-tiny-databases.

6 Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, USA, 2006. USENIX Association. URL: http://www.usenix.org/events/osdi06/tech/burrows.html.

7 J.W. Byers, M. Luby, and M. Mitzenmacher. A digital fountain approach to asynchronous reliable multicast. *IEEE Journal on Selected Areas in Communications*, 20(8):1528–1540, 2002. doi:10.1109/JSAC.2002.803996.

8 Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. Polarfs: an ultra-low latency and failure resilient distributed file system for shared storage cloud database. *Proc. VLDB Endow.*, 11(12):1849–1862, August 2018. doi:10.14778/3229863.3229872.

9 Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. Pigpaxos: Devouring the communication bottlenecks in distributed consensus. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, pages 235–247, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3448016.3452834.

10 Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*, pages 143–154. ACM, 2010. doi:10.1145/1807128.1807152.

11 James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Dale Woodford, Yasushi Saito, Christopher Taylor, Michal Szymaniak, and Ruth Wang. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

12 George Danezis, Giacomo Giuliari, Eleftherios Kokoris Kogias, Markus Legner, Jean-Pierre Smith, Alberto Sonnino, and Karl Wüst. Walrus: An efficient decentralized storage network, 2025. doi:10.48550/arXiv.2505.05370.

13 Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56:74–80, 2013. doi:10.1145/2408776.2408794.

14 Sisi Duan, Michael K. Reiter, and Haibin Zhang. Beat: Asynchronous BFT made practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 2028–2041, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3243734.3243812.

15 etcd. etcd documentation. https://etcd.io/docs/v2.3/admin_guide/, April 2021.

16 Facebook. RocksDB v7.3.1. https://github.com/facebook/rocksdb/tree/v7.3.1, 2024.

17 G.R. Goodson, J.J. Wylie, G.R. Ganger, and M.K. Reiter. Efficient Byzantine-tolerant erasure-coded storage. In *International Conference on Dependable Systems and Networks, 2004*, pages 135–144, 2004. doi:10.1109/DSN.2004.1311884.

18 Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, David A. Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on SIGCOMM*, pages 139–152. ACM, 2015. doi:10.1145/2785956.2787496.

19 Andreas Haeberlen and Alan Mislove. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *2nd Symposium on Networked Systems Design & Implementation (NSDI 05)*, Boston, MA, May 2005. USENIX Association. URL: https://www.usenix.org/conference/nsdi-05/glacier-highly-durable-decentralized-storage-despite-massive-correlated-failures.

20 T. Ho, M. Medard, R. Koetter, D.R. Karger, M. Effros, J. Shi, and B. Leong. A random linear network coding approach to multicast. *IEEE Transactions on Information Theory*, 52(10):4413–4430, 2006. doi:10.1109/TIT.2006.881746.

21 Guanzhou Hu, Yiwei Chen, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Crossword: Adaptive consensus for dynamic data-heavy workloads, 2025. doi:10.48550/arXiv.2509.07157.

22 Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in windows azure storage. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 15–26, Boston, MA, June 2012. USENIX Association. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/huang.

23 Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. TiDB: A Raft-based HTAP database. *Proceedings of the VLDB Endowment*, 13(12):3072–3084, 2020. doi:10.14778/3415478.3415535.

24 Intel. ISA-L: Intelligent storage acceleration library. https://github.com/intel/isa-l, 2024.

25 Y. Jia, G. Xu, C. W. Sung, S. Mostafa, and Y. Wu. HRaft: Adaptive erasure coded data maintenance for consensus in distributed networks. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1316–1326, 2022. doi:10.1109/IPDPS53621.2022.00130.

26 Ming Jin, Xiaojiao Chen, and Sian-Jheng Lin. Reducing the bandwidth of block propagation in bitcoin network with erasure coding. *IEEE Access*, 7:175606–175613, 2019. doi:10.1109/ACCESS.2019.2957496.

27 JLL. Growth of AI creates unprecedented demand for global data centers. https://www.jll.com/en-us/newsroom/growth-of-ai-creates-unprecedented-demand-for-global-data-centers, January 2024. Accessed: 2026-02-09.

28 Rithwik Kerur. eAID. https://github.com/RithwikKerur/eAID, 2026.

29 Marios Kogias and Edouard Bugnion. Hovercraft: achieving scalability and fault-tolerance for microsecond-scale datacenter services. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3342195.3387545.

30 John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: an architecture for global-scale persistent storage. *SIGPLAN Not.*, 35(11):190–201, November 2000. doi:10.1145/356989.357007.

31 Leslie Lamport. Paxos made simple, 2001. URL: https://api.semanticscholar.org/CorpusID:1936192.

32 Michael Luby. LT codes. In *The 43rd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 271–280, 2002. doi:10.1109/SFCS.2002.1181950.

33 Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 369–384, USA, 2008. USENIX Association. URL: http://www.usenix.org/events/osdi08/tech/full_papers/mao/mao.pdf.

34 Parisa Jalili Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. Ring paxos: A high-throughput atomic broadcast protocol. In *2010 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 527–536, 2010. doi:10.1109/DSN.2010.5544272.

35 Microsoft. IoT Signals report: IoT's promise will be unlocked by addressing skills shortage, complexity and security. URL: https://blogs.microsoft.com/blog/2019/07/30/iot-signals-report-iots-promise-will-be-unlocked-by-addressing-skills-shortage-complexity-and-security/, July 2019. Accessed: 2026-02-09.

36 Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 31–42, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2976749.2978399.

37 Michael Mitzenmacher. Digital fountains: A survey and look forward. In *Information Theory Workshop (ITW)*, pages 271–276, 2004. doi:10.1109/ITW.2004.1405313.

38 Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 358–372, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2517349.2517350.

39 Shuai Mu, Kang Chen, Yongwei Wu, and Weimin Zheng. When paxos meets erasure code: reduce network and storage cost in state machine replication. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '14, pages 61–72, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2600212.2600218.

40 Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. Dpaxos: Managing data closer to users for low-latency and mobile applications. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1221–1236, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3183713.3196928.

41 Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, 2014. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

42 Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, et al. Facebook's tectonic filesystem: Efficiency from exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 217–231, 2021.

43 James S. Plank and Kevin M. Greenan. Jerasure: A library in C facilitating erasure coding for storage applications, version 2.0. https://jerasure.org/, 2014. Technical Report UT-EECS-14-721, University of Tennessee.

44 Michael O. Rabin. The information dispersal algorithm and its applications. In *Sequences: Combinatorics, Compression, Security, and Transmission*, pages 406–419. Springer New York, 1990.

45 Irving S. Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

46 David Reinsel, John Gantz, and John Rydning. The digitization of the world from edge to core. Technical report, International Data Corporation (IDC), 2018. Sponsored by Seagate Technology. URL: https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf.

47 Victor Shoup. Sing a song of simplex. Cryptology ePrint Archive, Paper 2023/1916, 2023. doi:10.4230/LIPIcs.DISC.2024.37.

48 Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. CockroachDB: The resilient geo-distributed SQL database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*, pages 1493–1509. ACM, 2020. doi:10.1145/3318464.3386134.

49 Pasindu Tennage, Antoine Desjardins, and Lefteris Kokoris-Kogias. Racs-sadl: Robust and understandable randomized consensus in the cloud, 2025. arXiv:2404.04183.

50 Myna Vajha, Vinayak Ramkumar, Bhagyashree Puranik, Ganesh Kini, Elita Lobo, Birenjith Sasidharan, P. Vijay Kumar, Alexandar Barg, Min Ye, Srinivasan Narayanamurthy, Syed Hussain, and Siddhartha Nandi. Clay codes: Moulding MDS codes to yield an MSR code. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 139–154, Oakland, CA, February 2018. USENIX Association. URL: https://www.usenix.org/conference/fast18/presentation/vajha.

51 Zizhong Wang, Tongliang Li, Haixia Wang, Airan Shao, Yunren Bai, Shangming Cai, Zihan Xu, and Dongsheng Wang. CRaft: An erasure-coding-supported version of Raft for reducing storage cost and network cost. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (USENIX FAST 20)*, pages 297–308, 2020. URL: https://www.usenix.org/conference/fast20/presentation/wang-zizhong.

52 Mi Zhang, Qihan Kang, and Patrick P. C. Lee. FlexRaft: Exploiting flexible erasure coding for minimum-cost consensus and fast recovery. *IEEE Transactions on Parallel and Distributed Systems*, 2024.

A The Erasure Code

The most common code used in Information Dispersal is Reed-Solomon codes [45]. However, a natural question is whether information dispersal could benefit from other codes such as LT codes [32], Fountain codes [7], Locally Recoverable Codes (LRC) [22], Minimum Storage Regenerating (MSR) codes [50] or Random Linear Network Codes (RLNC) [20].

MSR codes and LRC codes are primarily optimized for node repair. Since our approach is not concerned with repairing a node, we do not consider these classes of codes. Both LT and Fountain codes are classes of rateless codes that allow the encoder to generate an unlimited number of fragments. We deliberately chose a fixed-parameter Reed-Solomon scheme for four reasons. First, Reed-Solomon is Maximum Distance Separable (MDS): *any* $F+1$ fragments suffice to reconstruct M, deterministically. LT and Fountain codes are only near-MDS and typically require $(1+\epsilon)(F+1)$ fragments to decode with high probability; for the small dimensions typical in Raft deployments (e.g., $F+1=3$ for $N=5$), this overhead is both significant and probabilistic, which would weaken the safety argument in Lemma 2 that relies on exactly $F+1$ fragments always being sufficient. RLNC achieves MDS behavior in expectation but only when the random coefficient matrix is full-rank, which holds with high probability over large fields but is not guaranteed. Second, Reed-Solomon fragments carry only a small integer index as metadata, whereas RLNC fragments must carry a length-$(F+1)$ coefficient vector and LT/Fountain fragments must carry a degree and neighbor list. Avoiding per-fragment metadata is central to eAID's goal of simplifying per-log-entry meta-information. Third, autonomous pruning is sound under Reed-Solomon precisely because fragments are interchangeable in count: a node can locally discard fragments knowing that any $F+1$ surviving fragments cluster-wide will decode. With LT or Fountain codes, the decodability of the surviving set depends on the specific degree distribution of which fragments were

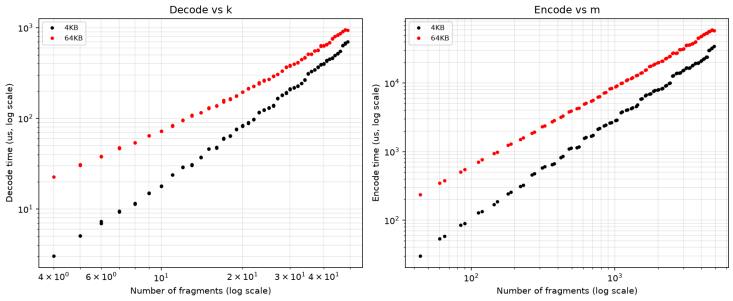


Figure 8 Reed–Solomon encode and decode cost for 4KB and 64KB payloads. Decode scales with the recovery threshold $k = F+1$; encode scales with the fragment pool size $m = (F+1) \times (N-1)$.

retained, which would require coordination to reason about safely. Fourth, although rateless codes offer the advantage of generating fragments on demand, in practice this benefit is modest in our setting: eAID's adaptive leader only generates fragments per node based on its current responsive quorum estimate. We elaborate on how many fragments the eAID leader generates in Section 4.1. Beyond these design-level considerations, Reed-Solomon also offers a practical implementation advantage: SIMD optimized libraries [24] (which we use in our implementation) deliver encoding and decoding throughput that, to the best of our knowledge, is not matched by any production-grade rateless-code library.

Notwithstanding, eAID can work with any of these erasure codes. However due to the practicality of the optimized library, we choose Reed-Solomon codes.

A.1 Coding Overhead

eAID draws each entry's fragments from a $(k = F+1, m = (F+1) \times (N-1))$ code, so the fragment pool grows quadratically in N while the recovery threshold stays fixed. We use a SIMD-optimized GF(2^{16}) Reed–Solomon library [43]. Figure 8 reports the resulting encode and decode costs for 4KB and 64KB payloads. Decode time depends only on k and remains under 1 ms even at $k = 50$ (i.e., $N = 99$). Critically, it is independent of m, and therefore of how many fragments were disseminated or subsequently pruned. Encode time scales with m and thus with the total coded bytes produced, reaching tens of milliseconds at the largest configurations. This cost is paid once per entry, at the leader, off the follower critical path, and is negligible at production cluster sizes: at $N = 5$ and $N = 11$, m is 12 and 60 respectively, placing encode time in the tens of microseconds.

B Integration with a Replicated Key-Value Store via Log Replication

In Section 5 we identified replicated key-value stores backed by log replication as the deployment setting for eAID. This appendix details that integration. We use Raft [41] as the log-replication substrate because it is the standard choice for production key-value stores [15, 48, 23] and provides a concrete, widely understood target. We assume familiarity with the Raft Consensus Protocol and describe how eAID replaces Raft's full-payload AppendEntries with encoded fragment dissemination (Appendix B.1), and finally show that this replacement preserves Raft's safety and liveness guarantees end-to-end, making eAID-KV a correct replicated key-value store.

B.1 Integration with eAID

The integration with Raft is structurally similar to the standalone IDA, with one key difference exploited from Raft's sequential log. Because Raft guarantees that a follower acknowledging log entry at index i must also have acknowledged all entries at indices $i' < i$, the leader does not need to broadcast a per-entry ack count. Instead, it maintains two monotonically advancing thresholds, T_1 and T_2: the highest index acknowledged by at least $\frac{3N}{4}$ followers, and the highest index acknowledged by all N followers, respectively. These thresholds are piggybacked on subsequent AppendEntries and heartbeat messages, allowing followers to determine the required fragment count for any prior entry from T_1 and T_2 alone (Algorithm 4). This compresses the cluster-wide ack-count broadcast of the IDA layer (Algorithm 2) into two integers, eliminating per-entry metadata while preserving the pruning guarantees of Lemma 2.

Algorithm 3 eAID-KV: Leader Adaptive Dissemination.

State: $currentTerm, commitIndex$ (Raft); $respEst \leftarrow N$; $A[m]$: ack sets; $sent[m,j], held[m,j]$: # frags sent / acked; $f_{\text{per}}[m]$: committed per-node threshold for entry m; T_1, T_2: highest indices acked by $\lceil 3N/4 \rceil, N$ followers
Tiers: $\mathcal{T} = [(N,1), (\lceil 3N/4 \rceil, 2), (F{+}1, F{+}1)]$ ▷ Tier 1, Tier 2, Tier 3 in list order
Tier(q) returns $(w, f_{\text{per}}) \in \mathcal{T}$ with the largest $w \le q$

procedure AppendEntry(M, index m)
  $\mathcal{F} \leftarrow$ Encode(M), tagged $(currentTerm, m)$; $(w, f_{\text{per}}) \leftarrow$ Tier$(\max(F{+}1, respEst - \delta))$
  Partition $\mathcal{F}$ into $S_1, \ldots, S_N$, $|S_j| = F{+}1$; $A[m] \leftarrow \{leaderId\}$
  Locally store $S_{leaderId}[1{:}f_{\text{per}}]$; $held[m, leaderId] \leftarrow f_{\text{per}}$
  for each follower j **do**
    Send AppendEntries$(currentTerm, leaderId, m{-}1, \log[m{-}1].term,$
    $\langle (M, S_j[1{:}f_{\text{per}}], currentTerm, m) \rangle, commitIndex, T_1, T_2)$ to j
    $sent[m,j] \leftarrow f_{\text{per}}$; $held[m,j] \leftarrow 0$
  repeat
    Start timer τ
    while $|A[m]| < w$ **and** timer not expired **do**
      Wait for AppendEntriesResponse$(j, term_j, success, held_j)$
      if $term_j > currentTerm$ **then** step down; **return**
      if $success$ **and** $held_j > held[m,j]$ **then** $held[m,j] \leftarrow held_j$
      if $held[m,j] \ge f_{\text{per}}$ **then** $A[m] \leftarrow A[m] \cup \{j\}$; UpdateThresholds$(m)$
    if $|A[m]| \ge w$ **then break**
    $(w, f_{\text{new}}) \leftarrow$ Tier$(\max(F{+}1, |A[m]|))$ ▷ timeout: step down a tier
    for each follower j **with** $f_{\text{new}} > sent[m,j]$ **do**
      Send AppendEntries$(currentTerm, leaderId, m{-}1, \log[m{-}1].term,$
      $\langle (M, S_j[sent[m,j]{+}1{:}f_{\text{new}}], currentTerm, m) \rangle, commitIndex, T_1, T_2)$ to j
      $sent[m,j] \leftarrow f_{\text{new}}$
    $f_{\text{per}} \leftarrow f_{\text{new}}$; $A[m] \leftarrow \{j \mid held[m,j] \ge f_{\text{per}}\}$
  until false
  $f_{\text{per}}[m] \leftarrow f_{\text{per}}$ ▷ freeze per-entry threshold for late-ack accounting
  $respEst \leftarrow |A[m]|$; Commit$(m)$
end procedure

procedure OnLateAppendEntriesResponse$(j, m, held_j)$
  if $held_j > held[m,j]$ **then**
    $held[m,j] \leftarrow held_j$
    if $j \notin A[m]$ **and** $held_j \ge f_{\text{per}}[m]$ **then** ▷ same threshold as the main loop (f_{per} for entry m)
      $A[m] \leftarrow A[m] \cup \{j\}$; UpdateThresholds$(m)$
end procedure

procedure UpdateThresholds(m)
  if $|A[m]| \ge \lceil 3N/4 \rceil$ **and** $m > T_1$ **then** $T_1 \leftarrow m$
  if $|A[m]| = N$ **and** $m > T_2$ **then** $T_2 \leftarrow m$
end procedure

We now describe how this integration manifests in three concrete modifications to Raft.

B.1.1 LogEntry Modifications

We extend the `LogEntry` structure to include two new fields: **Threshold 1** and **Threshold 2**. Both are tuples in the form $(Term, Index)$. **Threshold 1** represents the highest $(Term, Index)$ for which the leader has received responses from $3N/4$ nodes. Similarly, **Threshold 2** represents the highest $(Term, Index)$ for which the leader has received responses from all N nodes.

B.1.2 Log Replication and Commitment

In eAID-KV, log replication follows the standard Raft structure but utilizes erasure coding as described in Section 4.

B.1.3 Follower Processing and Storage Pruning

Upon receiving an `AppendEntries` message, the follower node initially persists all associated fragments. To optimize storage usage, it subsequently prunes fragments based on the global resilience indicated by the leader's **Threshold** values. The pruning strategy maintains a safety invariant: the cluster must always retain sufficient fragments to reconstruct a log entry despite $F < N/2$ failures. As shown in Section 4.3, it is safe for a follower to retain two fragments for all entries indexed until **Threshold 1**, and a single fragment for all entries indexed until **Threshold 2**.

B.2 Safety and Liveness

We prove Safety and Liveness under the **Partial Synchrony Model** with $F < N/2$ benign failures. The model functions similarly to the asynchronous model, except there is an unknown Global Stabilization Time (GST), after which all message transmissions have a known bound.

▶ **Theorem 4** (Safety of Raft with eAID). *Replacing Raft's full-replication* AppendEntries *payload with eAID's encoded fragments preserves all five Raft safety invariants.*

Proof. eAID modifies only the *payload* of AppendEntries (full log entries become Reed-Solomon fragments) and the *commit threshold check*. It does not alter term progression, voting rules, log indexing, or the leader-election protocol. We address each invariant.

Election safety and *leader append-only* depend solely on term progression and leader-local log management, neither of which is modified by eAID. They hold by Raft's original argument.

Log matching requires that two logs containing entries with the same (term, index) agree on all prior entries. eAID preserves the $(term, index, prevLogIndex, prevLogTerm)$ tuple in AppendEntries unchanged (Algorithm 3); the consistency check that followers perform on $prevLogIndex/prevLogTerm$ operates on the index/term metadata, not on the fragment payload. A follower that accepts a fragment for entry m has verified the prefix at $m-1$ matches the leader's, exactly as in vanilla Raft.

Leader completeness requires that any committed entry appears in the log of every subsequent leader. Raft enforces this via the voting rule: a candidate must have a log at least as up-to-date as a majority of voters. In eAID, an entry is committed once the leader collects $F+t' \geq F+1$ acks (i.e., a majority). By Lemma 1, at the commit point M is reconstructable from the cluster's surviving fragments. By Lemma 2, this reconstructability is preserved

Algorithm 4 eAID-KV: Follower.

State: log[]: local log; each entry stores fragments, term, index
currentTerm, *commitIndex*: standard Raft state
myT_1, myT_2: locally known pruning thresholds

procedure OnAppendEntries(*leaderId*, *term*, *prevLogIndex*, *prevLogTerm*, *entries*, *leaderCommit*, T_1^{ldr}, T_2^{ldr})
  if *term* < *currentTerm* **then**
    Send AppendEntriesResponse(*self*, *currentTerm*, *false*, 0) to leaderId; **return** ▷ (1) stale leader
  if log[*prevLogIndex*] does not exist **or** log[*prevLogIndex*].*term* ≠ *prevLogTerm* **then**
    Send AppendEntriesResponse(*self*, *currentTerm*, *false*, 0) to leaderId; **return** ▷ (2) log mismatch
  for each entry $e \in entries$ **do**
    if log[*e.index*] exists **and** log[*e.index*].*term* ≠ *e.term* **then**
      Truncate log from *e.index* onward ▷ (3) conflicting entry
    if any fragment in *e* is not tagged (*e.term*, *e.index*) **then**
      Send AppendEntriesResponse(*self*, *currentTerm*, *false*, 0) to leaderId; **return**
    Persist *e*'s fragments at log[*e.index*] ▷ (4) append
  if *leaderCommit* > *commitIndex* **then**
    commitIndex ← min(*leaderCommit*, index of last new entry) ▷ (5)
  $myT_1 \leftarrow \max(myT_1, T_1^{ldr})$; $myT_2 \leftarrow \max(myT_2, T_2^{ldr})$
  Prune ▷ opportunistic, runs in background
  for each entry $e \in entries$ **do**
    Send AppendEntriesResponse(*self*, *currentTerm*, *true*, |log[*e.index*].*fragments*|) to leaderId
end procedure

procedure Prune
  for each entry m in log **do**
    if $m \leq myT_2$ **then**
      Retain 1 fragment; discard the rest ▷ Tier 1: all N nodes hold a fragment
    else if $m \leq myT_1$ **then**
      Retain 2 fragments; discard the rest ▷ Tier 2: $\geq \lceil 3N/4 \rceil$ nodes hold fragments
    else
      Retain F+1 fragments ▷ Tier 3: bare quorum
end procedure

procedure Reconstruct(entry index m)
  Collect any F+1 distinct fragments for m from the cluster, all tagged (log[m].*term*, m)
  Decode M via Reed-Solomon interpolation ▷ same procedure for every entry
  return M
end procedure

through arbitrary pruning. Since a future leader is elected from a majority overlap with the committing majority, and at least one node in this overlap holds at least one fragment of M, the new leader can collect $F + 1$ fragments from the cluster (via Reconstruct, Algorithm 4) and reconstruct M. The leader then includes M in its log before serving new appends, satisfying leader completeness.

State machine safety follows from leader completeness and log matching: every node that applies entry m does so against an identical reconstructed M, since reconstruction is deterministic from any $F + 1$ fragments (a property of MDS codes). ◀

▶ **Theorem 5** (Liveness of Raft with eAID). *Under partial synchrony with a stable leader after GST, every client command submitted to the leader is eventually committed.*

Proof. Raft's liveness rests on three operational guarantees from its replication layer: (a) the leader can issue AppendEntries to followers and receive responses in bounded time after GST; (b) AppendEntries to a majority of correct followers eventually succeeds; (c) the leader can determine when an entry is replicated on a majority. We show eAID preserves all three.

(a) *Bounded response time.* eAID's AppendEntries carries fragments of size $B/(F+1)$ rather than the full entry of size B, and message delivery bounds after GST are preserved.

(b) *Eventual success.* The wait loop inside AppendEntry (Algorithm 3) is the Raft-integrated counterpart to Disperse's wait loop, and the same termination argument applies. By Lemma 3, after GST the loop collects $F + t' \geq F + 1$ acks within bounded time. The retransmission mechanism converges (Lemma 3) and does not introduce unbounded waiting.

(c) *Quorum certification.* eAID commits an entry when $|A[m]| \geq F+t'$, where $F+t' \geq F+1$ is precisely Raft's majority threshold. The leader treats Dispersed (i.e., the return of AppendEntry) as the commit-eligibility signal, preserving Raft's majority-quorum invariant.

The remainder of Raft's liveness argument – election timeouts, leader stability after GST, and progress through the log – is unchanged by eAID, since eAID modifies only how a single log entry's payload is replicated, not the Raft control flow (elections, term progression, commit-index advancement). Therefore Raft's liveness proof carries through with AppendEntries replaced by eAID's encoded variant.

Pruning (Lemma 2) and asynchronous late-ack collection (OnLateAppendEntriesResponse, Algorithm 3) occur after a log entry is committed and do not gate progress on subsequent entries; they affect only the storage footprint of already-committed entries and therefore cannot impact liveness. ◀