Manifest-Driven Repair Contracts for LRC in Content-Addressed Storage

Yiwei Gan*, Zhijie Huang*‡, Xiao Zhang*, Shujie Han*, Chentao Wu†, Nannan Zhao*, and Yuantao Wen*
*Northwestern Polytechnical University, China
{ganyw, wenyuantao}@mail.nwpu.edu.cn; {jayzy.huang, zhangxiao, shujiehan, nannanzhao}@nwpu.edu.cn
†Shanghai Jiao Tong University, China
wuct@cs.sjtu.edu.cn
‡Corresponding author: Zhijie Huang

***Abstract*—Locally repairable codes (LRC) reduce repair bandwidth, but deployed LRC systems assume block-addressed storage: decoded bytes can be written back to a fixed location. Content-addressed storage (CAS) adds another requirement: repaired bytes must still verify against the published content identifier (CID), the Merkle proof scope, and the manifest epoch authorized to publish repair state. This paper presents manifest-driven repair contracts for LRC in CAS. An EC-Merkle-DAG records identity and proof scope at write time, and a two-lane protocol separates request-scoped degraded reads from durable repair commits. Foreground recovery may return CID-valid bytes after sub-object verification; background repair commits only after rechecking manifest epoch, publish conditions, and repair-state preconditions. We implement the design in IPFS Cluster and evaluate it on an Aliyun 37-node deployment across five regions. The results validate ordinary reads, degraded reads, and explicit durable repair; LRC preserves CID-correct degraded reads across controlled 0–7 shard loss at $1.6\times$ storage amplification, versus RF5's $5.0\times$, while making the throughput, repair-time, and CID-verification costs of CAS-preserving repair explicit.**

***Index Terms*—content-addressed storage, locally repairable codes, erasure coding, Merkle DAG, durable repair, IPFS**

I. Introduction

CAS organizes data by the cryptographic hash of its content rather than by network location. Systems such as IPFS [1], [2] and Filecoin [3] demonstrate that this model enables global deduplication, tamper detection, and location-independent retrieval across decentralized, wide-area node sets. As these networks grow beyond millions of peers and petabytes of stored data [2], storage efficiency becomes a critical operational concern: every stored object must remain durably retrievable under temporary unavailability and network partitions without the luxury of fixed-topology repair channels assumed by traditional distributed storage.

Full replication is simple to operate but stores n copies of every object. At scale this cost is prohibitive: Azure's deployment study showed that repair traffic alone accounts for the majority of cross-rack bandwidth [4], and Facebook's f4 saved over 50% of warm-storage capacity by moving from replication to erasure coding [5]. LRCs improve on standard Reed–Solomon by keeping the common repair case local—a single lost block can be rebuilt from its local group of g_l blocks rather than the full stripe of k blocks [4]–[8].

However, every deployed LRC system—Azure LRC [4], f4 [5], HACFS [7], HDFS-EC [9], Wide-LRC [10]—assumes block-addressed storage, where a repaired block is written to a fixed position and repair is complete. In CAS, object identity is a cryptographic hash of content. A repaired block must not only be decodable—it must hash back to the CID that the system advertises to clients. This requirement is absent from every block-addressed LRC deployment.

The straightforward solution—re-hashing the entire object after every repair—is correct but defeats the purpose of locality: a local repair that contacts only $g_l{=}5$ blocks still pays $O(N)$ verification cost to re-hash the full object. A system that skips verification avoids this cost but risks promoting stale or unauthorized candidates into durable metadata, as Table I illustrates.

A concrete failure. Consider a 64 MiB object stored with LRC(10,4,2). One shard is lost; local parity decodes the missing block. A naive system writes the decoded block and marks the object healthy—but between decode and write-back, the manifest may advance, a concurrent repair may commit a different candidate, or GC may reclaim parity inputs. Section II-C formalizes the three failure modes.

These failures expose two missing contracts in existing LRC designs for CAS. First, the *identity contract* must specify which sub-object proof scope a repaired role must satisfy. Second, the *commit contract* must distinguish a request-scoped CID-valid candidate from a durable metadata update. This paper presents a manifest-driven design that supplies these contracts without falling back to whole-object re-hashing. The EC-Merkle-DAG plane records the content-addressed repair skeleton at write time: the user-visible `MetaPin` points to a `ClusterDAG`, the `ClusterDAG` points to an `ECManifest`, and the manifest points to stripe segments that name every data, local-parity, and global-parity role by CID. A deterministic alignment rule binds the Merkle DAG frontier to the LRC stripe layout, so foreground and background repair replay the same proof boundary instead of re-segmenting the object. The metadata also records the ordered owner set selected at write time, so repair can replay the published shard-owner binding rather than choosing owners during repair.

The design separates three outcomes. A degraded read may produce *readable* bytes, but the result is accepted only after

the bytes hash back to the expected CID. The result is *CID-valid* when it matches the current EC-Merkle-DAG scope. It becomes *committed* only when the compact background repair intent still matches the current manifest epoch and the publish preconditions hold. This separation preserves LRC locality for request-scoped service while preventing stale manifests from becoming durable metadata.

We make three contributions:

1) **Repair contract.** Manifest epoch, Merkle proof scope, and compact repair intent jointly define the safety boundary for LRC repair in CAS. A deterministic EC-Merkle-DAG alignment rule co-locates repair scope, proof scope, and parity scope within each manifest group, reducing per-repair verification from whole-object $O(N)$ to sub-object $O(g_l{+}d)$.
2) **Two-lane repair protocol.** The foreground degraded-read lane may serve only request-scoped CID-valid bytes. The background durable-repair lane executes leased intent, GC protection, candidate verification, and commit-or-abort, and publishes only after rechecking manifest epoch, publish, and repair-intent preconditions.
3) **Scoped implementation and evaluation.** We implement the design in IPFS Cluster ($\sim$3,200 lines of Go) and evaluate it on an Aliyun 37-node deployment across five regions. The evaluation covers ordinary read/recovery behavior, explicit ECRecovery, CID-correct throughput, degraded-read latency, durable repair time, and reconstruction CPU cost.

II. Background and Motivation

A. Erasure Coding and Locally Repairable Codes

Erasure coding splits an object into k data blocks, encodes them into $k + m$ coded blocks (where m blocks are parity), and distributes the blocks across storage nodes. Any k out of $k{+}m$ blocks suffice to reconstruct the original data. Compared to n-way replication, erasure coding achieves the same fault tolerance at a fraction of the storage cost: a code with rate $k/(k + m)$ stores only $(k + m)/k$ times the original data, versus $n\times$ for replication.

Standard Reed–Solomon (RS) codes [11] achieve optimal storage efficiency but require contacting k nodes for every repair, even when only one block is missing. LRCs [4], [6] address this by organizing coded blocks into *local groups*, each with its own local parity. When a single block is lost within a group, it can be reconstructed from the group members alone—typically $g_l \ll k$ blocks—without contacting the full stripe.

We use the notation LRC(k, g, r) where k data blocks are protected by g global parity blocks (Reed–Solomon) and r local parity blocks (XOR within each local group), for a total stripe width of $k + g + r$. Each local group contains $g_l = k/r$ data blocks plus one local parity block.[1] For example, LRC(10,4,2) has $k{=}10$ data blocks, $g{=}4$ global parity blocks, and $r{=}2$ local parity blocks (two groups of five data plus one XOR parity each)—giving storage amplification of $(10 + 4 + 2)/10 = 1.6\times$, versus $5.0\times$ for 5-replica. Local repair contacts at most $g_l = 5$ data blocks plus one local parity, instead of the $k = 10$ blocks required by standard RS.

[1] Our notation follows the (k, g, r) convention of Huang et al. [4]. Gopalan et al. [6] use (n, k, r) with a different meaning for r (locality parameter). We use r to denote the number of local parity blocks throughout.

Azure Storage's deployment of LRC [4] demonstrated that repair traffic accounts for the majority of cross-rack bandwidth in large-scale storage clusters. Subsequent systems—f4 [5], HACFS [7], C2DN [8], Wide-LRC [10]—confirmed that locality-aware repair is essential for practical operation at scale.

B. Content-Addressed Storage and IPFS

In CAS, objects are identified by a cryptographic hash of their content, called a CID. Unlike block-addressed systems (HDFS, Azure Blob) where blocks are identified by position within a volume, CAS blocks are self-certifying: any holder can verify that a block matches its CID by re-hashing it.

IPFS [1] organizes objects as Merkle DAGs: a root CID references child blocks, each identified by its own CID, forming a hash-linked tree. IPFS Cluster adds a coordination layer: it manages object distribution, replication, and metadata across a set of IPFS nodes, providing a cluster-controlled storage service over the IPFS content-addressing model. The relevant property for this paper is that object identity is determined by content, not location. Moving a block to a different node does not change its CID; corrupting a block does.

C. Why CAS Complicates Coded Repair

In block-addressed systems, a repaired block is correct if it passes the coding equations—the system writes it to the designated position and repair is complete. In CAS, byte reconstruction is necessary but not sufficient, because CAS introduces a *multi-level identity chain* that block-addressed systems lack.

A CAS object has a root CID (the hash of the top-level Merkle node), which references child blocks each identified by their own shard CIDs. The root CID is the published identity that clients use to retrieve the object. The shard CIDs are internal: they bind each data block to a specific position in the Merkle DAG. A repaired block must satisfy *both levels*: it must reconstruct the correct bytes (shard-CID match) *and* those bytes must be consistent with the root-CID under the current manifest layout. This is not a generic concurrency problem—it is specific to content-addressing, because object identity is derived from content, not assigned by a coordinator.

A precise counterexample. Suppose an object with root CID R is stored with LRC(10,4,2). Shard 3 is lost from local group 1. Local parity l_1 is sufficient to reconstruct the bytes of shard 3. The decoded bytes pass the coding equations and match shard CID s_3. However, between the decode and the durable write-back, the operator re-encodes the object during manifest maintenance, advancing the manifest epoch. The new manifest assigns shard 3 to a different local group with different parity inputs. The old candidate's bytes are still

TABLE I
BYTE-CORRECT CAS REPAIR FAILURES.

Failure mode	What happens	Observable consequence
Stale parity	Manifest advances (e.g., object re-encoded with new layout) while repair uses old parity inputs. Decoded bytes pass old coding equations.	Reconstructed block hashes to wrong CID; client receives data under an identity the object no longer holds.
Epoch drift	Candidate is decoded correctly, but between decode and write-back another repair commits a different candidate for the same target.	Two conflicting durable entries for the same block; metadata diverges across peers.
GC race	Candidate depends on temporary parity artifacts. GC reclaims them before durable commit.	Committed metadata references parity objects that no longer exist; future repair of the same scope fails.

correct under the *old* manifest, but serving them under root CID R violates the current layout—the Merkle proof path from s_3 to R no longer traverses the expected intermediate nodes. A block-addressed system would simply overwrite the position; in CAS, the candidate is byte-correct but CID-invalid under the current epoch.

This counterexample illustrates the core CAS-specific requirement: a repaired block must not only decode correctly, it must be *provably consistent with the published root CID under the current manifest*. Two specific requirements follow:

1) **Identity preservation.** The repaired block must hash to the shard CID referenced by the object's Merkle DAG, and the Merkle proof path from that shard CID to the root CID must verify under the current manifest layout.
2) **Commit authorization.** Between reconstruction and durable write-back, the metadata environment may change: the manifest may advance to a new epoch, a concurrent repair may supersede the same target, or garbage collection may reclaim temporary artifacts. A system that promotes any candidate to durable state without rechecking these conditions risks polluting metadata with entries that are CID-invalid under the current epoch.

In block-addressed systems, verifying a repaired block costs $O(1)$ (a checksum comparison against a known position). In CAS, the naive alternative is whole-object re-hashing ($O(N)$), which negates the locality benefit of LRC. The design challenge is to achieve $O(g_l+d)$ per-repair verification—proportional to the local group size and Merkle depth, not the object size—while preserving CID correctness. Table I illustrates three failure modes when a CAS system applies byte-correct repair without these checks.

We call a reconstructed scope *CID-valid* if it hashes back to the expected object identity under the current manifest; it becomes *committed* only after durable metadata has advanced under verified publish conditions. This paper presents a design that closes these gaps without falling back to whole-object rehashing.

TABLE II
TARGET DEPLOYMENT ASSUMPTIONS.

Dimension	Assumption
Deployment	Operator-managed IPFS Cluster with a trusted control plane.
Faults	Crash, omission, corruption detection, manifest drift, and GC races.
Workload	Large immutable objects; degraded, partial, and seek-heavy reads.
Out of scope	Byzantine behavior, open-membership adversarial consensus, correlated-failure durability.

D. System Model

We target an operator-managed IPFS Cluster deployment where the control plane coordinates owner metadata, recovery metadata, and repair execution. The system relies on CID checks, Merkle-path validation, and manifest-derived repair metadata—not Byzantine consensus. Table II summarizes the assumptions.

The fault model covers crash faults plus corruption detection: peer or block unavailability, corrupted bytes that fail integrity checks, manifest or repair-state drift, and races between garbage collection and active repair. Compound failures are allowed as long as surviving data and parity remain sufficient for local or stripe-wide reconstruction. Byzantine behavior and malicious quorum manipulation are out of scope.

The target workload is immutable objects large enough that erasure coding is preferable to full replication, with degraded reads, partial reads, and seek-heavy access patterns where recovering a small scope should not require fetching or rehashing the full object.

This operating model imposes five requirements: (1) CID-valid service must be separate from committed state; (2) degraded reads must be side-effect-free with respect to durable metadata; (3) durable repair must recheck the manifest epoch and publish conditions before commit; (4) common-case repair should stay local whenever the code permits; (5) interrupted repair must resume from persisted state.

III. SYSTEM OVERVIEW

Figure 1 shows the end-to-end architecture. The same manifest-derived metadata guides write-time layout, foreground degraded reads, and background durable repair. What changes across paths is not the object identity or repair scope, but the authority each path has over durable state.

Object lifecycle. At write time, the system splits the object into data blocks, produces local and global parity, and records the layout in cluster-managed metadata. The metadata chain runs from the user-visible entry (`MetaPin`) through the cluster object graph (`ClusterDAG`) to the stripe layout record (`ECManifest`) and stripe segment records. Each segment stores the role CIDs for data, local parity, and global

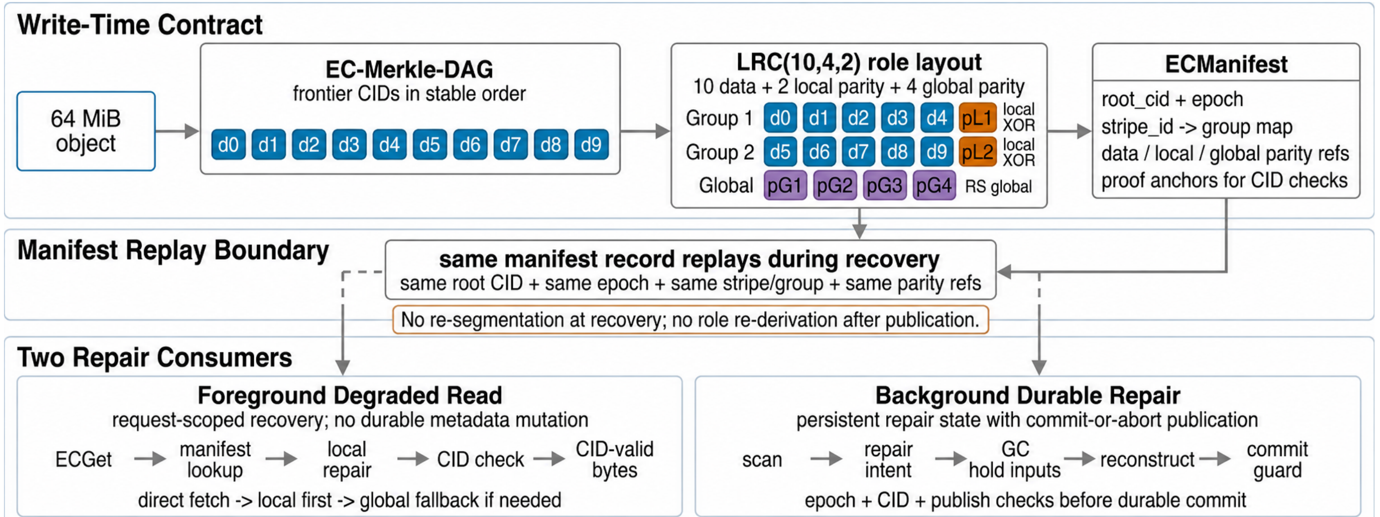


Fig. 1. LRC-CAS repair contract and authority boundaries.

parity in stripe order. On a degraded read, the read path follows this chain to locate the relevant stripe, group, and parity references, then attempts direct fetch, local repair, and stripe-wide fallback in order. A foreground recovery produces a CID-valid candidate but does not claim durable convergence; background repair commits or aborts conservatively after rechecking publish conditions.

Write-time owner binding. Write time records two bindings. The manifest captures which role CID belongs to which stripe and local group, and the ownership record captures which peers hold each shard in preference order. The allocator selects owners, streams shards to the ordered set, and commits the binding through cluster-managed metadata. Recovery replays this binding rather than selecting a new owner set.

IV. Repair Design

This section presents the repair design: a deterministic alignment rule that binds the object DAG to the coding layout at write time, three admissibility conditions that govern CID-correct reuse, and a two-lane protocol in which foreground reads restore service while background repair restores durable state.

A. Deterministic Alignment Rule

The design uses one deterministic rule to map the object Merkle DAG to the LRC stripe layout:

1) Choose the lowest DAG frontier whose payload fits the target stripe size.
2) Split wide frontiers by stable sibling order until each piece fits one local group.
3) Aggregate narrow frontiers under the same parent only when doing so preserves a stable proof boundary.
4) Bind each resulting scope to one stripe and one local group in the manifest.
5) Bind the enclosing shard to the ordered owner set recorded at publish time.
6) Store the parity references and root anchors needed to replay this decision later.

Given the same manifest and repair target, this procedure returns the same stripe/group assignment for both foreground and background repair. The rule is fixed at write time: recovery cannot invent new groupings, re-segment the object, or widen proof scope after publication. Figure 2 illustrates how the frontier, segment boundary, manifest group, and proof scope are recorded once and replayed by both repair lanes.

B. Three Admissibility Conditions

A reconstructed candidate may be reused as a CID-valid result only if:

1) **Scope match:** the manifest still names the stripe, group, and parity inputs used for reconstruction;
2) **Identity match:** the reconstructed scope verifies against the published CID under the intended proof boundary; and
3) **Epoch match:** the repair is evaluated against the same root, manifest, and repair target that authorized it.

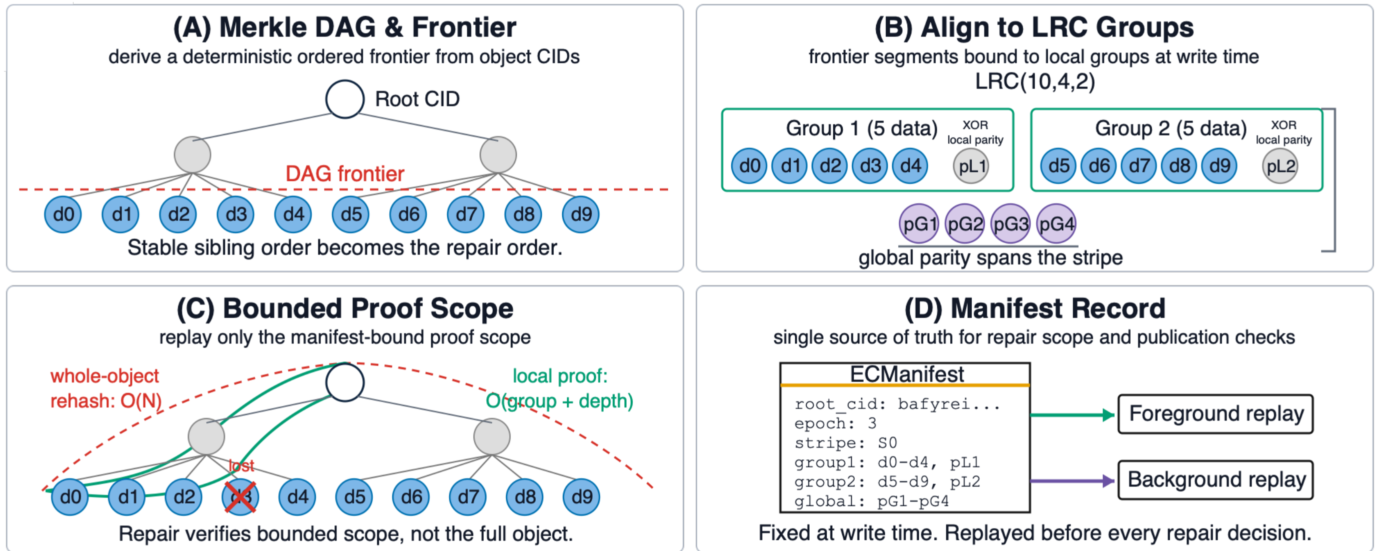
A candidate built from superseded parity fails condition (1). Corrupted bytes fail the Merkle hash check in condition (2). A candidate from an earlier manifest epoch fails condition (3) once the epoch advances. No candidate can pass all three without using the correct parity for the correct scope under current authorization—which is CID-correctness.

Durable commit adds a fourth requirement: the background lane must recheck these conditions under the *current* epoch before metadata can advance.

C. Foreground Degraded Read

When a degraded read arrives, the system resolves the manifest, locates the relevant stripe and local group, and follows a local-first policy:

1) try direct block fetch from the recorded owner;
2) try peer fetch from available peers;



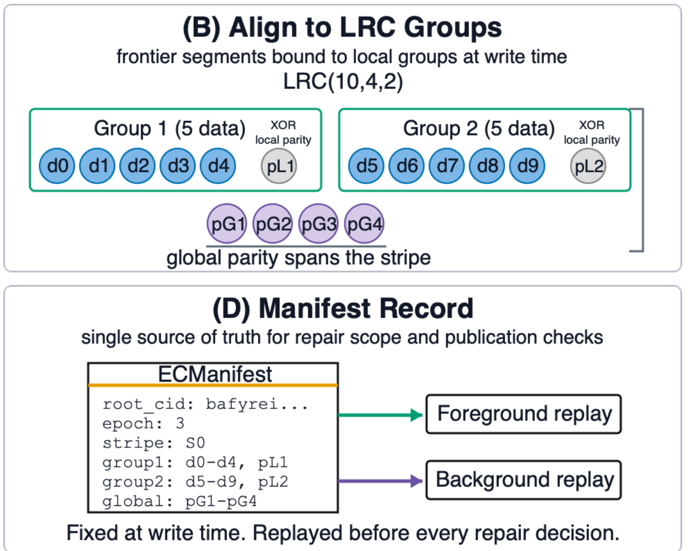


Fig. 2. DAG-to-LRC alignment and proof scope.

TABLE III
PROTOTYPE ENFORCEMENT OF REPAIR INVARIANTS.

Invariant	Runtime carrier	Checked in
Repair scope fixed at write time	manifest stripe/group layout	write finalize, both repair paths
Owner order fixed at write time	shard-owner binding	write finalize, recovery scheduling
CID-correct reuse	object root + proof scope	degraded-read validation, publish checks
Durable commit authorization	repair state + manifest epoch	background publish or abort

3) reconstruct within the local group (fan-in $\leq g_l$);
4) escalate to stripe-wide reconstruction only when local repair fails.

Foreground recovery stops once it produces a CID-valid result. It may cache validated blocks for later reads, but it does not advance durable metadata.

D. Background Durable Repair

Background repair uses the same reconstruction primitive but applies stronger commit checks:

1) discover a degraded object and persist repair intent;
2) protect inputs and candidate artifacts from GC;
3) reconstruct with the same manifest-guided primitive;
4) recheck current epoch and publish conditions;
5) commit or abort conservatively.

Foreground success does not imply background success. A candidate sufficient to serve a foreground request may fail the epoch or publish conditions required for durable commit. The background lane commits by writing updated repair state into the cluster metadata chain via consensus, so all peers observe the same outcome. If any check fails, the lane aborts, requeues the object, and leaves the CID unchanged.

The protocol aborts in three situations: validation failure, epoch/target change, or interrupted repair. Restart resumes from the last durable checkpoint under the current epoch. Overlapping repairs are handled conservatively: only the background lane owns publish-visible state, and abort takes precedence over speculative publish.

E. Worked Example: Single-Shard Loss

Consider a 64 MiB object stored with LRC(10,4,2): 10 data blocks of 6.4 MiB each in two local groups of five, plus one XOR local parity block per group and four RS global parity blocks (16 blocks total).

Suppose block 3 (in group 1, which holds data blocks 0–4 and local parity l_1) becomes unavailable.

Foreground: The degraded-read path identifies group 1, fetches blocks $\{0, 1, 2, 4\}$ and local parity l_1, reconstructs block 3 via XOR, and verifies against the root CID via the Merkle proof path. The candidate is *CID-valid* and served to the client; durable metadata does not change.

Background: The same reconstruction primitive runs intentionally, but the lane has different authority: it persists a repair record, holds parity against GC, and rechecks the manifest epoch before committing. If the epoch has advanced, the lane aborts and requeues. The candidate becomes *committed* only after the commit guard discharges.

F. Applicability Boundary

Local repair reads at most $g_l = 5$ surviving group members (the remaining data blocks plus local parity) out of 16 total, versus $k = 10$ for standard RS—a 50% fan-in reduction. This benefit dominates for objects large enough that multi-shard network cost exceeds local verification work. Captured prototype metadata is small in absolute terms: ECManifest CBOR blocks are about 0.1–0.3 KiB, and the pin-level JSON records in the Aliyun artifacts are about 0.7 KiB. Below $\sim$1 MiB, however, this fixed metadata plus 16-role stripe bookkeeping is

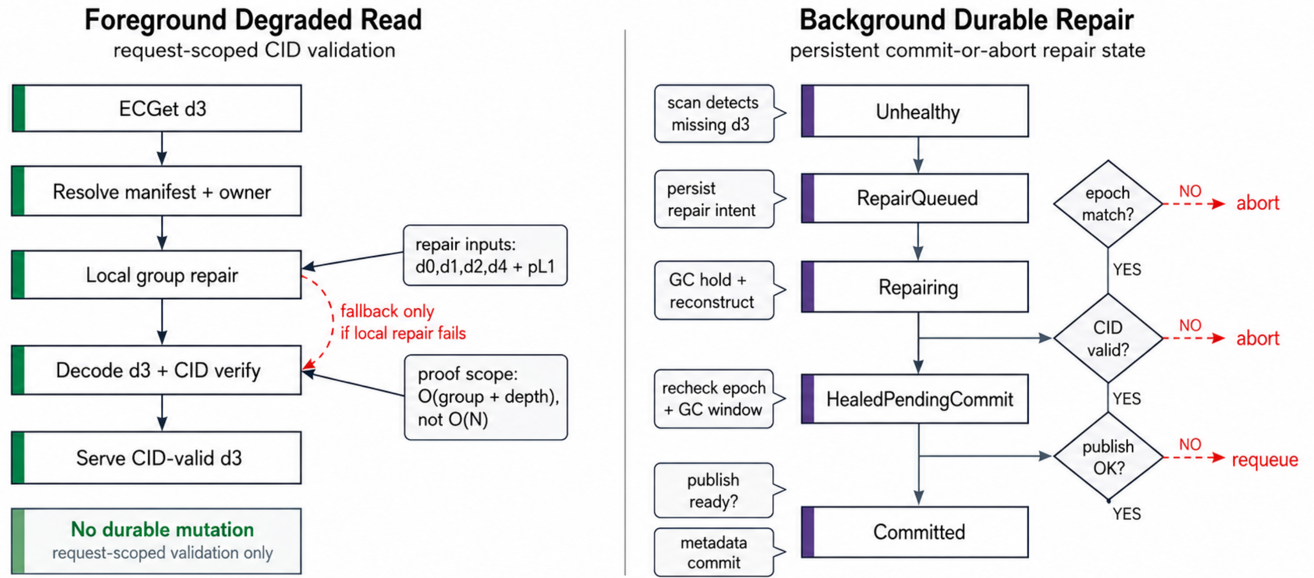


Fig. 3. Two-lane recovery and durable-state authority.

TABLE IV
PROTOTYPE METADATA CARRIERS.

Carrier	Role
`MetaPin`	User-visible object entry and stable root identity.
`ClusterDAG`	Cluster-managed object graph for enumeration and repair targeting.
`ECManifest`	Stripe, group, parity references replayed by both repair paths.
Shard-owner binding	Ordered owner record for fetch and repair scheduling.
Repair state	Persisted intent, health state, and publish progress.

poorly amortized, so the write-time policy routes such objects to replication instead. Verification dominates reconstruction CPU cost (Section VI-F), making proof caching a natural future optimization target.

V. Implementation

The design is implemented in IPFS Cluster by extending the existing write, read, and repair paths rather than introducing a separate storage stack.

Scale. The implementation adds ∼3,200 lines of Go: the encoding and recovery module (∼1,800 lines covering erasure coding, manifest construction, Merkle-tree alignment, and degraded-read recovery) and the repair loop and state modules (∼1,200 lines covering background scheduling, durable state persistence, and GC isolation). The consensus, API, and storage engine layers are untouched.

Metadata chain. The metadata chain runs from the user-visible entry (`MetaPin`) through the cluster object graph (`ClusterDAG`) to the stripe layout record (`ECManifest`), with parity stored in packed superblocks. Table IV summarizes the carriers. Recovery joins the shard-owner binding with the manifest's block-to-stripe index to reconstruct a full block-to-owner view from write-time metadata.

Durable state and GC isolation. Durable repair progress is kept in cluster-managed metadata, not process-local memory. The state records finer-grained checkpoints than the paper's two design-level outcomes (CID-valid, committed), enabling persistence, resume from the last checkpoint after restart, and conservative abort. GC isolation pins metadata and parity objects at write time; background repair temporarily protects candidate artifacts under a 30 s repair lease during active repair. Commit and abort clear the lease deadline, while an expired lease can be reacquired by a later repair attempt.

Engineering extensions. The prototype uses a serialized encode-and-stream path at write time, and the write measurements include that implementation cost. The manifest-driven pipeline admits parallel role encoding and proof caching as direct engineering extensions, but this paper does not evaluate those optimizations.

VI. Evaluation

The WAN evaluation uses one platform: an Aliyun 37-node deployment across five regions. It addresses five questions, with one result surface for each data category. (1) Does the current prototype preserve CID correctness for ordinary reads and controlled degraded reads? (2) What write/read throughput and storage-amplification tradeoff do RF1, RF5, and LRC show on the Aliyun 37-node deployment? (3) What degraded-read latency does LRC show under controlled shard

loss? (4) What wall-clock time does explicit durable repair require? (5) How much reconstruction CPU is spent on decoding versus CID verification? The measurements use n=5 where repeated runs are reported and are treated as a cost envelope rather than as a statistical ranking of nearby latency values. They combine WAN transfer, prototype scheduling, and CID verification costs; they are not isolated coding microbenchmarks or controlled comparisons against other EC implementations. Concurrency scaling and head-to-head RS, LRC, and alternative IPFS code-family comparisons under identical CID-verification contracts remain future work. The CPU split answers a narrower cost-attribution question inside the repair path; it does not substitute for the WAN wall-clock results.

A. Experimental Setup

Aliyun 37-node deployment. The evaluation uses 37 ECS instances across five Alibaba Cloud regions: Hangzhou (8), Heyuan (8), Chengdu (8), Zhangjiakou (8), and Hohhot (5). Each ECS instance runs one IPFS daemon and one cluster daemon (Ansible-orchestrated Docker Compose, direct public-IP communication). Inter-region RTT ranges from 3.7 ms (Chengdu–Zhangjiakou) to 63.7 ms (Hangzhou–Heyuan), with a typical cross-region latency of 30–65 ms. Heyuan and Hohhot share a low-latency path (~0.4 ms) and are effectively co-located for network purposes. Storage uses Aliyun local-path EBS. The LRC configuration is LRC(10 / 4 / 2). WAN throughput, degraded-read latency, and durable repair measurements are reported for this deployment. The reconstruction CPU split is reported as a repair-path cost attribution associated with the same implementation and evaluation surface. Throughput, degraded-read, and durable-repair rows report mean and standard deviation across five runs. The evaluation studies one LRC design point; comparing RS, LRC, and alternative IPFS code families under the same CID-verification contracts remains future work. The claim boundary is the Aliyun 37-node surface: WAN throughput, degraded-read latency, durable repair time, and 0–7 shard-loss CID correctness.

B. WAN Correctness Under Controlled Shard Loss

The Aliyun 37-node characterization verifies that the prototype recovers data and preserves CID correctness across all loss levels within the LRC(10 / 4 / 2) stripe tolerance. Using the full degraded-read path on a 64 MiB object with 0 through 7 simultaneous shard losses, losses of 1–2 trigger local repair within the group of 5; losses of 3–6 trigger stripe-wide reconstruction; loss of 7 uses the full global parity set. In every case, the recovered object hash matched the expected digest and the CID verification check passed, confirming end-to-end correctness of the manifest-guided repair and CID verification pipeline under real WAN conditions. Table V summarizes the recovery path and CID outcome under shard loss; latency is reported only in Figure 5. The next performance summaries report the performance envelope: WAN throughput (Figure 4), degraded-read latency (Figure 5), and durable repair time (Figure 6).

TABLE V. SHARD-LOSS CORRECTNESS (ALIYUN 37-NODE, 64 MIB).

Shards lost	Recovery path	CID valid
0	Full fetch (no repair)	✓
1–2	Local repair	✓
3–6	Stripe-wide fallback	✓
7	Global repair	✓

C. WAN Write and Read Throughput

Figure 4 reports Aliyun 37-node WAN throughput for RF1, RF5, and LRC (mean $\pm\sigma$, n=5). Across 64–1024 MiB, LRC writes at 2.5–2.7 MiB/s and reads at 2.5–3.7 MiB/s; its write path is within 1.0–1.3× of RF5 while using 1.6× rather than 5.0× storage. These rows include WAN transfer, stream scheduling, serialized write-side encoding, and CID verification, so they characterize the current CID-preserving prototype rather than isolated LRC equations. As secondary cost context, write throughput per unit storage is roughly 0.4 MiB/s for RF5 and 1.6–1.7 MiB/s for LRC; the primary comparison remains raw throughput plus storage amplification. We do not infer size-by-size monotonicity from individual read points: for example, the RF1 256 MiB read row is treated as a WAN/read-path measurement artifact within this cost envelope rather than as a scaling claim.

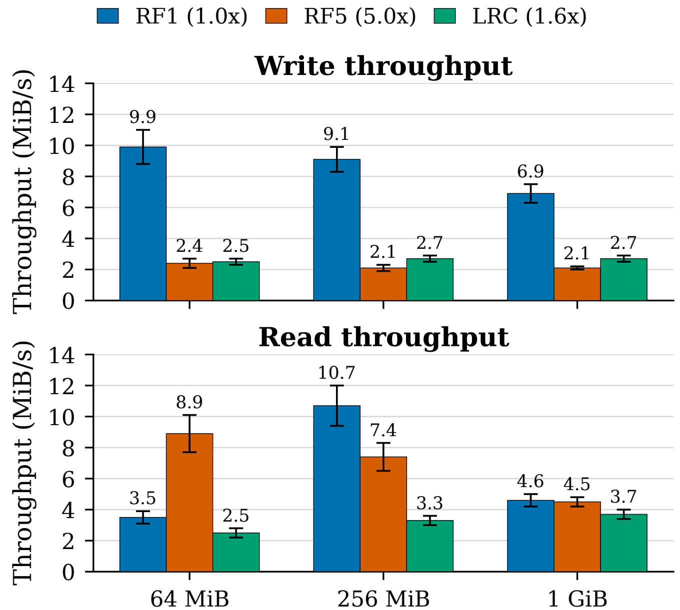


Fig. 4. WAN throughput and storage amplification (Aliyun 37-node, n=5).

Context from the IPFS-EC study. Gan et al. [12] reported RS(14,10) write throughput of ~7.6 MiB/s on the same Aliyun 37-node cluster at 1.4× storage amplification. Their published tables report aggregate add/recovery averages rather than per-size CID-preserving repair rows, so we use those results only as related context rather than a head-to-head baseline.

D. WAN Degraded-Read Latency

Figure 5 reports degraded-read elapsed time under controlled shard loss (mean $\pm\sigma$, n=5). All data points in Figure 5 go through the same manifest-guided degraded-read code path (manifest lookup, per-shard owner fetch, and CID verification).

The loss = 0 point is therefore not a lightweight direct read—it fetches all k=10 data shards (64 MiB total) through the full pipeline. Local repair at loss = 1–2 fetches at most g_l=5 surviving group members plus one local parity (∼38 MiB of WAN transfer), so those rows benefit from reduced fetch volume rather than a lighter-weight code path. The curve should be read as a coarse WAN recovery envelope shaped by fan-in, owner placement, and n=5 run-to-run variance, not as a point-by-point latency ranking. All 64 MiB loss levels complete within 25 s, and every 256 MiB loss level from 2–7 completes within 60 s; the 256 MiB loss = 1 outlier is discussed in Section VI-G.

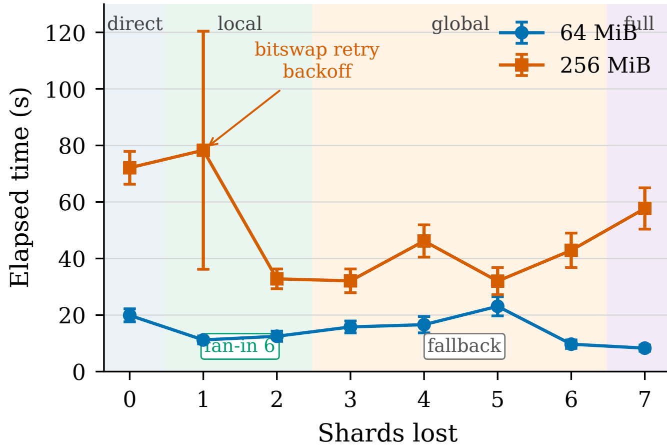


Fig. 5. WAN degraded-read latency (Aliyun 37-node, n=5).

E. Background Durable Repair: WAN Completion Time

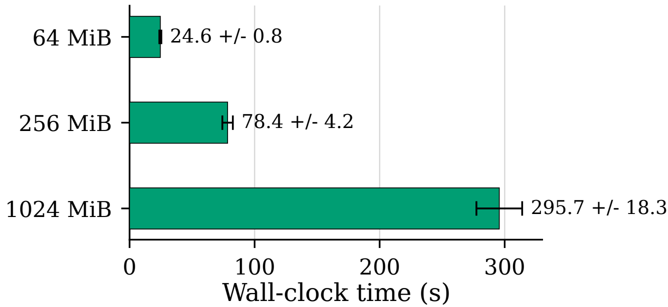


Fig. 6. Durable repair time (Aliyun 37-node, n=5).

Figure 6 reports durable-repair completion time in the Aliyun 37-node characterization across three object sizes (n=5 runs each, 1 shard lost). For 64 MiB objects, the durable-repair sequence completes in 24.6 ± 0.8 s (mean $\pm\sigma$), with all runs passing CID verification. The durable-repair lane reports a compact intent status surface (`Pending`, `Leased`, `CandidateReady`, `Committed`, or `Aborted`) while preserving the same commit-or-abort boundary. Larger objects scale approximately linearly: 256 MiB completes in 78.4±4.2 s and 1024 MiB in 295.7±18.3 s. The dominant cost is cross-region shard fetch over 30–65 ms RTT links; the durable commit boundary (metadata advancement, GC protection, commit-or-abort logic) contributes negligible overhead.

F. Reconstruction CPU Cost

Figure 7 reports a repair-path CPU-cost measurement associated with the Aliyun evaluation, using five repeats. The result isolates CPU work after bytes are available: LRC decoding accounts for about 7% of total reconstruction time, while CID verification accounts for about 92%. The five-repeat means are 1.59 ms/op for decoding, 22.16 ms/op for CID verification, and 23.96 ms/op total after bytes are fetched. This is a cost-attribution measurement for the repair path, not a replacement for the WAN wall-clock measurements, where cross-region fetch and scheduling also matter. It explains why CAS repair cannot be evaluated only as a coding operation: the correctness check against the CID proof scope is the dominant per-block CPU cost.

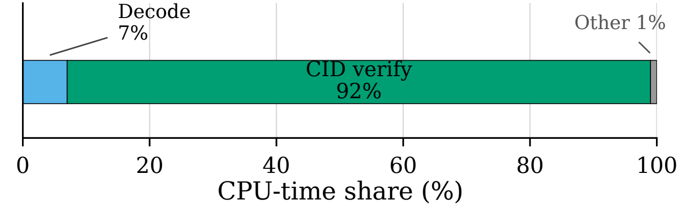


Fig. 7. Repair-path CPU attribution after fetch (n=5).

G. Discussion

Prototype and statistical boundary. The 37-node rows use five runs per configuration, so we report means and standard deviations but do not use them for hypothesis tests or fine-grained rankings. The prototype also leaves important implementation headroom: the write path does not fully parallelize role encoding and streaming, and the Aliyun artifacts identify write-tail behavior as an engineering bottleneck rather than a fundamental LRC limitation. The evaluation therefore treats throughput results as measured prototype cost envelopes under CID-preserving repair semantics. The protocol evidence is empirical; a formal model or proof of the repair state machine remains future work.

Read-path interpretation. Figure 4 shows that RF5 read throughput exceeds LRC read throughput by 1.2–3.6× across the reported object sizes (8.9 vs. 2.5 MiB/s for 64 MiB, 7.4 vs. 3.3 MiB/s for 256 MiB, and 4.5 vs. 3.7 MiB/s for 1 GiB). The two panels characterize different bottlenecks: writes include ingestion, coding/streaming, and metadata publication, whereas reads mainly retrieve already-published blocks. Thus, per-size read measurements are more exposed to owner placement, request scheduling, and WAN run-to-run variability. We therefore treat the RF1 256 MiB point as a read-path/WAN sample within the reported cost envelope, not as a general size-scaling trend. This gap captures the cost profile of a CID-preserving LRC read path in CAS: the read path performs manifest lookup, targeted owner fetch, and Merkle-proof verification on every repaired scope. The CPU split in Figure 7 shows why this verification is visible in the prototype. The EC-Merkle-DAG design bounds this identity-preserving work to $O(g_l+d)$ per repair scope rather than $O(N)$ per object. For workloads where storage cost dominates (warm and cold

tiers), the $3.1\times$ reduction in storage amplification (from $5.0\times$ to $1.6\times$) outweighs the read throughput gap.

Read-path and WAN-variance interpretation. All points in Figure 5 exercise the same manifest-guided degraded-read code path (Section VI-D), so the loss = 0 row is not a lighter-weight "direct read"—it fetches all k=10 data shards (64 MiB total) through the same per-shard WAN fetch and CID-verify pipeline. Local repair at loss = 1–2 fetches at most g_l=5 surviving group members plus one local parity (∼38 MiB), reducing WAN transfer volume and explaining why those rows are faster than the loss = 0 full-fetch baseline. This is a fan-in effect, not a physical claim that losing shards improves the storage medium. The non-monotonic loss-level shape should likewise be read as a WAN cost envelope affected by owner placement, cross-region scheduling, and n=5 run-to-run variance, not as a pointwise ranking of coding difficulty. The 256 MiB loss = 1 case (78.3 ± 42.1 s) is treated as an owner-discovery/backoff outlier because the remaining 256 MiB recovery rows fall in a narrower 32.0–57.7 s range.

Applicability boundary. The size cutoff in Section IV-F is an implementation policy, not a claim about sub-MiB LRC efficiency; the reported WAN results target immutable objects large enough to amortize fixed repair metadata and stripe bookkeeping.

Proof-caching boundary. CID verification is necessary for CAS correctness: a candidate role cannot be committed merely because it decodes. WAN wall-clock includes cross-region fetch, manifest lookup, and verification. Because CAS repair repeatedly verifies Merkle proof scopes, proof caching is a natural future optimization; this paper quantifies the verification cost but claims no measured speedup from proof caching. The CPU-only upper bound is the measured 92% verification share in Figure 7; end-to-end WAN speedup would be lower because cross-region fetch and scheduling remain.

Engineering headroom. The current prototype exposes three candidate engineering extensions: parallel role encoding, coarser owner-batched reads, and proof caching; the paper does not claim these optimizations have been evaluated. Similarly, concurrency scaling and controlled degraded-read comparisons with other systems under the same CID-preserving semantics remain outside this evaluation.

VII. Related Work

Table VI summarizes the key distinctions among repair-relevant systems along five dimensions.

Repair-efficient storage systems. Azure LRC [4], [6] established that local parity reduces repair fan-in and cross-rack bandwidth. f4 [5], HACFS [7], C2DN [8], and Wide-LRC [10] confirmed that code geometry, data distribution, and workload must be co-designed. Recent work on degraded-read optimization (DRBoost [13], LESS [14], NCBlob [19], Hitchhiker [20]) further reduces I/O and network cost. HDFS natively supports RS erasure coding since Hadoop 3.0 [9]. All these systems assume block-addressed storage and do not face CID verification during repair.

TABLE VI
Repair-relevant systems (Pub.=durable publication; P=partial, I=implicit).

System	Substrate	CID	Pub.	WAN	Model
Azure LRC [4]	Local par.	×	×	✓	Oper.
f4/HACFS [5], [7]	RS/layer	×	×	✓	Oper.
HDFS-EC [9]	RS	×	×	×	Oper.
Wide-LRC [10]	Local par.	×	×	✓	Oper.
DRBoost [13]	MSR	×	×	×	Oper.
LESS [14]	RS	×	×	×	Oper.
IPFS-EC variants [12], [15], [16]	RS/custom	P	×	P	Dec.
IPFS-AE [17]	AE	P	×	P	Dec.
Filecoin [3]	PoRep	✓	I	✓	Dec.
Glacier [18]	Repl.	×	I	×	Dec.
This paper	LRC	✓	✓	✓	Cluster

Content-addressed and decentralized storage. IPFS [1], [2] establishes content-addressing and Merkle DAG structure. The IPFS-EC row groups Gan et al.'s decentralized EC optimization [12], Shin et al.'s RS-based IPFS storage mechanism [15], and Liang and Yang's IPFS reliability report [16]; each adds coding for reliability or space savings but stops short of a durable commit boundary. Estrada-Galiñanes et al. [17] build IPFS dApp storage components around alpha entanglement codes and decentralized monitoring/repair. That work makes the complementary case that IPFS applications can avoid simple pinning by choosing a code family with stronger entanglement structure and community-level maintenance. Our design makes a different comparison point explicit: for an RS/LRC-style layout in a cluster-controlled CAS, the missing mechanism is not only a code choice but a repair contract that says when a reconstructed role is CID-valid, epoch-current, protected from GC races, and safe to publish. The evaluation therefore compares against replication for storage and bandwidth cost, while the related-work comparison separates codec families from the publication semantics needed for durable CAS repair. Filecoin [3] uses proof-of-replication to verify storage, but its incentive-layer proofs serve a different purpose (storage market verification) than our cluster-internal repair authorization. Glacier [18] addresses correlated failure in decentralized replication. Our work targets a narrower operating point: cluster-controlled deployment where coded repair, object identity, and durable publication share the same control plane.

Proof-oriented integrity. PDP [21], PoR [22], and data-availability proofs [23], [24] show that coding and proof structure must be designed together. Survey work on network-coded storage [25]–[27] makes the same point. Our work is not a remote-audit protocol; it asks how much proof structure is needed for cluster-internal repair to remain CID-correct.

Positioning. Azure LRC shows why locality matters but operates without content-addressing. IPFS-EC shares the CAS setting but lacks a durable commit boundary. PDP/PoR design coding and proofs together but address remote audit. This paper combines local-first repair, CID-correct reuse, and an explicit durable commit guard in one protocol.

VIII. Conclusion

This paper presents manifest-driven repair contracts for LRC in content-addressed storage. The EC-Merkle-DAG binds the object DAG, manifest, stripe segments, and CID proof scope at write time, reducing per-repair verification from whole-object $O(N)$ to sub-object $O(g_l+d)$. Foreground reads restore service via CID-valid candidates without mutating durable metadata; the background lane commits only after rechecking epoch and publish preconditions. The Aliyun 37-node evaluation validates read/recovery behavior and explicit ECRecovery. It also shows CID-correct recovery across controlled 0–7 shard loss and reports WAN throughput, degraded-read latency, and durable repair time at $1.6\times$ LRC storage amplification versus RF5's $5.0\times$.

These results support the central claim that LRC can retain locality in content-addressed storage without giving up published object identity or durable repair semantics. The remaining boundary is explicit: the paper does not claim Byzantine safety, adversarial open membership, correlated-failure durability, or production-wide self-healing.

Acknowledgment

This work was partially supported by the National Natural Science Foundation of China (Grant Nos. 62272394, U25B2022, 62502391, and 62202382). We would like to thank the anonymous reviewers and the paper shepherd, Theodore M. Wong, for their insightful comments and constructive suggestions, which have greatly improved the quality of this manuscript.

References

[1] Benet J. IPFS – Content Addressed, Versioned, P2P File System[J/OL]. arXiv:1407.3561, 2014.

[2] Trautwein D, Raman A, Tyson G, et al. Design and Evaluation of IPFS: A Storage Layer for the Decentralized Web[C]//SIGCOMM 2022. New York: ACM, 2022.

[3] Protocol Labs. Filecoin: A Decentralized Storage Network[R/OL]. 2017.

[4] Huang C, Simitci H, Xu Y, et al. Erasure Coding in Windows Azure Storage[C]//USENIX ATC 2012. Berkeley: USENIX Association, 2012.

[5] Muralidhar A, Lloyd W, Roy S, et al. f4: Facebook's Warm BLOB Storage System[C]//11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). Berkeley: USENIX Association, 2014: 383–398.

[6] Gopalan P, Huang C, Simitci H, Yekhanin S. On the Locality of Codeword Symbols[J]. IEEE Transactions on Information Theory, 2012, 58(11): 6925–6934.

[7] Xu T, Zhou Y, Jiang P, et al. HACFS: A Hierarchical Architecture Combining Erasure Codes and Replication for Fault-Tolerant File Systems[C]//14th USENIX Conference on File and Storage Technologies (FAST 16). Berkeley: USENIX Association, 2016: 43–58.

[8] Alimadadi M, Bai Y, Chen S, et al. C2DN: How to Harness Erasure Coding at the Edge for Efficient Content Delivery[C]//18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21). Berkeley: USENIX Association, 2021: 1159–1173.

[9] Apache Software Foundation. HDFS Erasure Coding[EB/OL]. `hadoop.apache.org`, 2018.

[10] Kadekodi S, Silas S, Clausen D, et al. Practical Design Considerations for Wide Locally Recoverable Codes (LRCs)[C]//21st USENIX Conference on File and Storage Technologies (FAST 23). Berkeley: USENIX Association, 2023: 1–16.

[11] Reed I S, Solomon G. Polynomial Codes Over Certain Finite Fields[J]. Journal of the Society for Industrial and Applied Mathematics, 1960, 8(2): 300–304.

[12] Gan Y, Huang Z, Shi Y, et al. Erasure Coding Based Optimization in Decentralized Distributed Storage Systems[C]//2024 International Conference on Networking, Architecture and Storage (NAS 2024). IEEE, 2024: 96–103.

[13] Niu X, Zhang G, Li Z, et al. DRBoost: Boosting Degraded Read Performance in MSR-Coded Storage Clusters[C]//24th USENIX Conference on File and Storage Technologies (FAST 26). Berkeley: USENIX Association, 2026: 613–629.

[14] Cheng K, Li G, Li X, et al. LESS is More for I/O-Efficient Repairs in Erasure-Coded Storage[C]//24th USENIX Conference on File and Storage Technologies (FAST 26). Berkeley: USENIX Association, 2026: 631–641.

[15] Shin H, Lee M, Kim S. Space and Cost-Efficient Reed-Solomon Code Based Distributed Storage Mechanism for IPFS[C]//14th International Conference on Information and Communication Technology Convergence (ICTC 2023). Jeju Island: IEEE, 2023: 1165–1169.

[16] Liang Q, Yang Y. Making the InterPlanetary File System (IPFS) More Reliable[R]. EPFL, 2023.

[17] Estrada-Galiñanes V, ElRouby A, Theytaz L M-A. Efficient Data Management for IPFS dApps[J/OL]. arXiv preprint arXiv:2404.16210, 2024.

[18] Haeberlen A, Mislove A, Druschel P. Glacier: Highly Durable, Decentralized Storage Despite Massive Correlated Failures[C]//2nd Symposium on Networked Systems Design and Implementation (NSDI 05). Berkeley: USENIX Association, 2005.

[19] Gan C, Hu Y, Zhao L, et al. Revisiting Network Coding for Warm Blob Storage[C]//23rd USENIX Conference on File and Storage Technologies (FAST 25). Berkeley: USENIX Association, 2025: 139–154.

[20] Rashmi K V, Shah N B, Gu D, et al. A Hitchhiker's Guide to Fast and Efficient Data Reconstruction in Erasure-coded Data Centers[C]//SIGCOMM 2014. New York: ACM, 2014.

[21] Ateniese G, Burns R, Curtmola R, et al. Provable Data Possession at Untrusted Stores[C]//ACM CCS 2007. New York: ACM, 2007.

[22] Juels A, Kaliski B S. PORs: Proofs of Retrievability for Large Files[C]//ACM CCS 2007. New York: ACM, 2007.

[23] Al-Bassam M, Sonnino A, Buterin V. Fraud and Data Availability Proofs: Maximising Light Client Security and Scaling Blockchains with Dishonest Majorities[J/OL]. arXiv:1809.09044, 2018.

[24] Al-Bassam M, Sonnino A. Fooling Data Availability Sampling in High-Throughput Blockchains[J/OL]. IACR ePrint 2023/1079, 2023.

[25] Dimakis A G, Prabhakaran V, Ramchandran K. Decentralized Erasure Codes for Distributed Networked Storage[J]. IEEE Transactions on Information Theory, 2006, 52(6): 2809–2816.

[26] Dimakis A G, Godfrey P B, Wu Y, et al. Network Coding for Distributed Storage Systems[J]. IEEE Transactions on Information Theory, 2010, 56(9): 4539–4551.

[27] Dimakis A G, Ramchandran K, Wu Y, et al. A Survey on Network Codes for Distributed Storage[J]. Proceedings of the IEEE, 2011, 99(3): 476–489.