SHARD: Scalable High-Availability RDMA Datastore

Technical Report

Aarul Dhawan Mihir Shah Hieu Nguyen
{adhaw2, mihirs2, hieunn2}@illinois.edu
University of Illinois Urbana-Champaign
CS 598: Storage Systems — Spring 2026

Contents

1 Introduction

Most storage systems today route every read and write through the server CPU. At small scale that's fine, but as clusters grow, the CPU becomes the bottleneck rather than the network or disk. Modern data center networks are fast enough — 25–100 Gbps is common — that the latency of going through a kernel network stack, a server thread, and back again is starting to dominate over the actual transmission time. RDMA (Remote Direct Memory Access) hardware offers a different model: a client can read or write directly into a remote server's memory over the network without the server CPU being involved at all. A handful of research systems from the last decade — FaRM [1], HERD [2], Pilaf [3] — showed that this can drop key-value latency by an order of magnitude over conventional TCP-based systems.

We built SHARD to explore what a practical distributed storage system looks like when you design around one-sided RDMA from the ground up. The data path uses one-sided RDMA for all bulk transfers. Objects are erasure-coded across multiple servers using Reed-Solomon coding, so the system tolerates node failures while using significantly less storage than full replication. A lightweight coordinator manages cluster membership, tracks which nodes are alive, and tells clients how to distribute their data. We ran the whole thing on CloudLab `xl170` nodes with Mellanox ConnectX NICs over 25 Gbps RoCE, using libfabric as the RDMA portability layer.

The project raised some genuinely hard design questions. RDMA's programming model is built around memory regions that are registered with the NIC at fixed addresses, but erasure coding requires splitting objects into shards and writing them to multiple servers simultaneously. Handling disk spill while keeping the same one-sided RDMA interface visible to clients required a staging buffer scheme that took a few iterations to get right. The coordinator's failure detection has to balance speed (you want to notice failures quickly) against false positives (declaring a slow server dead is worse than waiting a few extra seconds).

This report covers the full system in depth: the wire protocol, the memory management model, the erasure coding integration, failure detection, and what we learned from benchmarking it against AWS S3 Express One Zone.

2 Background and Motivation

2.1 RDMA Programming Model

With RDMA, the NIC handles memory operations directly without involving the remote CPU. Before any data can move, memory regions must be *registered* with the NIC, which pins the memory in physical address space and returns a remote key (`rkey`) that the remote side can use to authorize reads and writes. Once registration is done, a one-sided write looks like: post a work request to the local send queue with the local buffer address, the remote virtual address, and the remote key. The NIC takes care of the DMA, the network transfer, and the remote DMA into the target server's memory. The remote CPU never sees an interrupt. Completion is signaled by a Completion Queue Entry (CQE) on the initiating side.

We use one-sided RDMA writes for all PUT data transfers and one-sided reads for all GET data transfers. Control messages (slot requests, grants, commit acknowledgements) use two-sided RDMA *sends* (equivalently, TCP-like message passing), which do involve the remote CPU but are fine for the small messages involved. All of this goes through libfabric's `verbs` provider rather than directly through the InfiniBand verbs API, which means the same code can run on RoCE (RDMA over Converged Ethernet), InfiniBand, or AWS EFA by changing a single fabric provider string.

2.2 Erasure Coding

Reed-Solomon erasure coding is the same idea behind RAID-6 and the erasure codes used in production storage systems like Azure Storage [5] and Facebook's f4. The basic scheme takes an object, splits it into k data shards of equal size, and computes m parity shards using GF(256) arithmetic over a Vandermonde or Cauchy matrix. Any k of the $k + m$ shards are sufficient to reconstruct the original object.

The storage overhead is $1 + m/k$ compared to $f + 1$ for replication with f failures tolerated. With $k = 2$ and $m = 1$ (which handles one failure), you pay $1.5\times$ overhead versus $2\times$ for full replication. At $k = 4$, $m = 2$, you pay $1.5\times$ overhead and tolerate two failures, which would require $3\times$ full replication. The savings compound quickly as cluster sizes grow.

We use Intel ISA-L (Intelligent Storage Acceleration Library) for the actual encoding and decoding. ISA-L uses AVX-512 intrinsics where available and is significantly faster than a naive GF(256) implementation, which matters at large object sizes where encoding can start to add measurable latency.

2.3 Why S3 Express One Zone as a Baseline

AWS S3 Express One Zone is Amazon's lowest-latency managed object store. It targets sub-10ms latency and runs within a single availability zone to avoid cross-AZ networking. Even so, every request goes through HTTP/TLS, request authentication (SigV4), and multiple layers of API infrastructure. That puts a hard floor of roughly 5–15ms on latency depending on object size. For latency-sensitive applications — databases writing logs, ML training jobs writing checkpoints, financial systems writing state — that floor matters.

S3 Express is also the right comparison point because it's what you'd actually deploy in production if you weren't building something custom. It has managed availability, geo-replication, IAM integration, and you pay per request with no infrastructure to maintain. Beating it on both latency *and* cost at sufficient throughput is the meaningful bar for a system like ours.

3 System Architecture

SHARD has three components: a coordinator, storage servers, and a client library. Figure 1 shows how they fit together.

The **coordinator** is a plain TCP server that manages cluster membership. Servers register their fabric addresses with it on startup. Clients query it to get the current list of alive servers and the k/m parameters to use. Clients can also open a persistent watch connection to receive push notifications when a server dies. The coordinator also pings each server periodically and declares it dead if enough pings fail.

Storage servers expose an RDMA-registered memory pool to clients. Each server manages two storage tiers: a primary in-memory RDMA pool and a secondary NVMe disk tier for overflow. A control thread handles slot allocation requests and commit messages over two-sided RDMA sends. All actual object data moves via one-sided RDMA writes and reads from the client, bypassing the server CPU entirely on the data path.

The **client library** is a C++ library (with Python bindings via pybind11) that handles erasure coding, drives all RDMA operations, and talks to the coordinator. For a PUT, the client encodes the object into shards, requests slots from all servers in parallel, writes shards via one-sided RDMA, and then sends commit messages. For a GET, the client asks servers for shard locations, reads k shards in parallel via one-sided RDMA reads, and decodes.

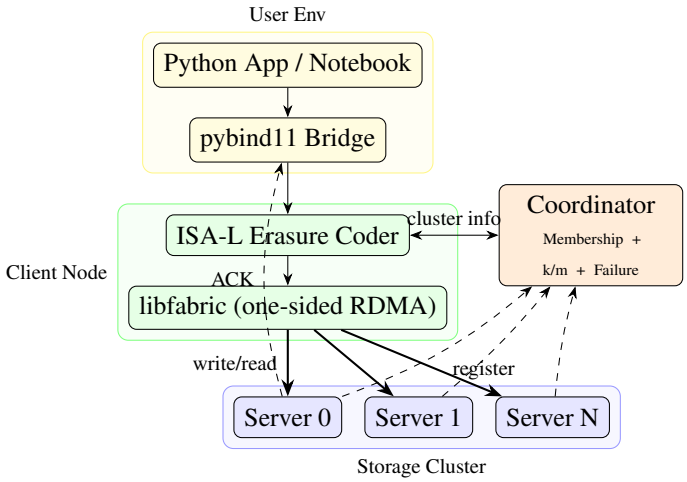

Figure 1: SHARD system architecture. Bulk shard data travels via one-sided RDMA (thick arrows); small control messages use two-sided sends (thin arrows); coordinator registration and watch connections use plain TCP (dashed arrows).

A key architectural decision was keeping the control plane and data plane completely separate. The coordinator and all slot negotiation happen over TCP and two-sided RDMA sends. All actual object data moves exclusively through one-sided RDMA. This means the coordinator is never in the hot path for data movement and server threads only wake up for control messages, not for every byte transferred.

4 Wire Protocol

4.1 Message Types

The control protocol uses eight message types, summarized in Table 1.

Table 1: Control protocol message types.

Type	Value	Purpose
`kPutRequest`	1	Client requests a slot to write a shard into
`kSlotGrant`	2	Server replies with remote address and RDMA key
`kPutCommit`	3	Client confirms RDMA write is complete
`kGetRequest`	4	Client asks for the location of a shard
`kSlotInfo`	5	Server replies with address, key, and on-disk flag
`kDelRequest`	6	Client requests deletion of a shard
`kAck`	7	Generic acknowledgement (with status code)
`kDiskRelease`	8	Client releases a staging slot after a disk read

4.2 Key Message Structures

PutRequestMsg (fixed header + variable-length key):

```
struct PutRequestMsg {
    MsgType  type;           // 1 byte
    uint8_t  _pad[3];        // 3 bytes alignment
    uint32_t key_len;        // length of key string
    uint32_t shard_len;      // size of this shard in bytes
    uint32_t object_size;    // original object size (for decode)
    uint8_t  shard_idx;      // which shard: 0 .. k+m-1
    uint8_t  k;              // data shard count
    uint8_t  m;              // parity shard count
    uint8_t  _pad2;
    // key bytes follow immediately in the same send buffer
};
```

SlotGrantMsg (server's reply to a PutRequest):

```
struct SlotGrantMsg {
    MsgType  type;           // 1 byte
    uint8_t  _pad[3];
    uint32_t slot_idx;       // server-assigned slot ID
    uint64_t remote_addr;    // RDMA target address
    uint64_t rkey;           // RDMA memory key
};
```

SlotInfoMsg (server's reply to a GetRequest):

```
struct SlotInfoMsg {
    MsgType    type;
    StatusCode status;         // kOk or kNotFound
    uint8_t    shard_idx;
    uint8_t    on_disk;        // 0 = RDMA pool, 1 = disk (staging)
    uint32_t   shard_len;
    uint32_t   object_size;
    uint64_t   remote_addr;
    uint64_t   rkey;
    uint32_t   staging_slot_idx;   // valid when on_disk = 1
    uint32_t   _pad2;
};
```

The design keeps control messages small. Even with a 256-byte max key, the largest control message is under 400 bytes, which fits in a single two-sided RDMA send. The pre-registered control send/receive buffers are 384 bytes each (`kCtrlBufSize = 128 + kMaxKeySize`), so no dynamic allocation is needed on the hot path.

4.3 Control Buffer Layout

Each connection (both client-side and server-side) maintains pre-registered buffers:

- `ctrl_send`: 384 bytes, registered for RDMA sends
- `ctrl_recv`: 384 bytes, registered for RDMA receives
- `data_buf`: 1 MB, registered for one-sided writes/reads

The data buffer is shared across all shards for a given server connection. Since operations are sequential per connection (the client waits for one operation to complete before starting the next on the same connection), this is safe without any synchronization.

5 Erasure Coding

5.1 Matrix Construction

The erasure coder uses a Cauchy matrix over GF(256), generated by `gf_gen_cauchy1_matrix()`

from ISA-L. For a (k, m) code:

- The encode matrix is $(k + m) \times k$
- The first k rows form the identity matrix (data shards pass through unchanged)
- The bottom m rows are the Cauchy coefficients for the parity shards

ISA-L precomputes a table of GF(256) lookup values from the encode matrix via `ec_init_tables()`, which trades a small upfront cost for very fast AVX-512 encoding.

5.2 Encoding

```
std::vector<std::vector<uint8_t>> Encode(
    const uint8_t *data, size_t data_len) {
    // Compute aligned shard size
    size_t frag = ((data_len + k - 1) / k + 63) / 64 * 64;

    // Create k+m zero-filled shards
    std::vector<std::vector<uint8_t>> shards(
        k + m, std::vector<uint8_t>(frag, 0));

    // Distribute data into k data shards
    for (int i = 0; i < k; i++) {
        size_t off = i * frag;
        size_t n = std::min(frag, data_len > off ? data_len - off : 0);
        memcpy(shards[i].data(), data + off, n);
    }

    // Compute m parity shards with ISA-L
    std::vector<uint8_t *> ptrs(k + m);
    for (int i = 0; i < k + m; i++) ptrs[i] = shards[i].data();
    ec_encode_data(frag, k, m, encode_gftbls.data(),
                   ptrs.data(), ptrs.data() + k);
    return shards;
}
```

The 64-byte alignment on shard sizes is required by ISA-L's SIMD routines and doubles as good cache alignment. All shards are zero-padded at the tail to reach the alignment boundary, and the original length is stored separately in each server's slot metadata so the decoder knows where to truncate.

5.3 Decoding

Decoding is only needed when some data shards are missing. If all k data shards (indices 0 through $k-1$) are present, the decoded object is just their concatenation truncated to the original length. When some data shards are absent and parity shards must fill in:

1. Collect any k available shards (data or parity) and note which indices they came from.
2. Extract the rows of the encode matrix at those indices to form a $k \times k$ submatrix.
3. Invert the submatrix via `gf_invert_matrix()` (GF(256) Gaussian elimination).
4. Compute decode GF tables from the inverse matrix.
5. Apply `ec_encode_data()` to regenerate the k data shards.
6. Concatenate data shards and truncate to the original object length.

The key insight is that any k shards work as the input, regardless of whether they're data or parity shards, as long as their row indices from the encode matrix are linearly independent — which the Cauchy construction guarantees for any subset.

6 Memory Management

6.1 Free-List Allocator

The core allocator is a sorted free-list over a flat byte pool:

```
struct PoolAlloc {
    struct Block { uint64_t off; uint32_t len; };
    std::vector<Block> free_list;   // sorted by offset, coalesced
    uint64_t           pool_size;
    uint64_t           bytes_free;

    uint64_t alloc(uint32_t len);  // first-fit; UINT64_MAX on failure
    void     free(uint64_t off, uint32_t len);
};
```

Allocation uses first-fit: scan for the first block at least as large as the request, split it if larger, and return the offset. Deallocation inserts the freed block into the sorted list and coalesces with adjacent blocks to prevent fragmentation. This is the same allocator for both the RDMA slab pool and the NVMe disk pool, just operating over different address spaces.

6.2 Multi-Slab RDMA Pool

Registering all of a server's potential memory with the NIC upfront wastes physical pages that may never be used. Instead, servers grow the RDMA pool lazily in 512 MB slabs:

```
struct MultiSlabAlloc {
    struct Slab {
        Buffer    buf;       // 512 MB, physically allocated
        PoolAlloc palloc;    // free-list over this slab
    };
    std::vector<Slab> slabs; // up to kMaxSlabs = 8 (4 GB total)

    AllocResult alloc(uint32_t len);
    void        free(uint8_t slab_id, uint64_t off, uint32_t len);
};
```

When a PutRequest arrives and the current slabs are full, the allocator tries to add a new slab. If `slabs.size() < kMaxSlabs`, it allocates a fresh 512 MB buffer and appends it to the slab list. The new slab needs to be registered with every active client connection before it can be used for RDMA writes. This registration happens lazily: each connection thread maintains its own `slab_conns[]` array and registers any new slabs at the start of the next control message it handles.

This lazy approach means slab growth is almost invisible to the common case — you only pay the registration overhead once when the slab is first created, and only on the thread that happens to handle the first message after growth.

6.3 NVMe Disk Spill

When all 8 slabs (4 GB) are exhausted, shards spill to a local NVMe file. The disk pool uses the same `PoolAlloc` allocator, but over the file's byte range (up to 64 GB). Disk I/O uses `pread`/`pwrite` with the slot's byte offset.

The challenge with disk spill is that one-sided RDMA reads require a registered memory address — you can't RDMA-read directly from a file descriptor. This is solved with a staging buffer.

6.4 Staging Buffer

The staging buffer is a pre-registered 64 MB region divided into 64 slots of 1 MB each:

```
Buffer             staging_buf;         // 64 MB, RDMA-registered
std::atomic<bool>  staging_in_use[64];  // per-slot occupancy
```

Slot allocation uses a lock-free compare-exchange scan:

```
int alloc_staging_slot(SharedState &shared) {
    for (;;) {
        for (int i = 0; i < kMaxStagingSlots; i++) {
            bool expected = false;
            if (shared.staging_in_use[i]
                    .compare_exchange_weak(expected, true))
                return i;
        }
        std::this_thread::yield();
    }
}
```

For a disk PUT, the server grants the client the staging slot's address. The client writes the shard there via RDMA. On commit, the server copies the shard from the staging slot to disk and releases the slot. For a disk GET, the server reads the shard from disk into a staging slot, gives the client the staging address, and waits for a `DiskRelease` message before freeing the slot.

From the client's perspective, disk-backed shards are completely transparent: the RDMA read or write targets a staging slot address instead of a slab address, but the operation looks identical. The only difference is the `on_disk` flag in `SlotInfoMsg`, which tells the client to send a `DiskRelease` message after it finishes reading.

6.5 Slot Metadata

Each server tracks up to 65,536 active shards in a flat array:

```
struct SlotEntry {
    bool     in_use      = false;
    uint8_t  slab_id     = 0;
    bool     on_disk     = false;
    uint32_t shard_len   = 0;
    uint32_t object_size = 0;  // for degraded decode on GET
    uint64_t pool_off    = 0;
};

SlotEntry slots[kMaxSlots];                          // 65536 entries
std::unordered_map<std::string, int> key_to_slot;  // shard_key -> slot
```

Shard keys are formed by appending the shard index byte to the user-supplied key string (e.g., `"mykey0002"` for shard 2 of key `"mykey"`). This lets a single server hold multiple shards of the same object without key collisions, which can happen when $k + m >$ cluster size.

7 PUT Protocol

A PUT proceeds in four phases, all of which are parallelized across the $k + m$ servers where possible.

Phase 1 — Encode. The client encodes the object into $k+m$ shards using ISA-L. This happens entirely on the client CPU and is proportional to object size. For a 64 KB object with $k = 2$, each shard is 32 KB.

Phase 2 — Control RTT (slot grant). The client sends a `PutRequest` to all $k + m$ servers in parallel and waits for all `SlotGrant` replies. Each server:

1. Checks if the key already exists and evicts the old slot if so (overwrite).
2. Allocates space from the RDMA slab pool or, if full, from the disk pool.
3. For disk allocations, allocates a staging slot.

4. Records a `PendingPut` entry with all the metadata.
5. Replies with the granted address and RDMA key.

The overwrite path is important: without it, repeated writes to the same key would leak slab memory. The server finds the old slot, frees its RDMA or disk allocation, and removes the key from `key_to_slot` before granting a new slot.

Phase 3 — RDMA Write. For each server, the client copies shard i into its local data buffer and issues a one-sided RDMA write targeting the granted address. All writes are issued in parallel and the client polls until all $k + m$ completion queue entries arrive. The server CPU sees none of this.

Phase 4 — Commit RTT. The client sends `PutCommit` to all servers. Each server finalizes its slot:

- For in-memory shards: marks the slot as committed and adds the key to `key_to_slot`.
- For disk shards: `pwrites` the shard from the staging buffer to disk, releases the staging slot, then commits.

The server replies with an `Ack`. The key is only visible to GET operations after all $k + m$ servers have committed, which provides a clean consistency boundary: a GET either sees all shards or none.

Figure 2 shows the message sequence for one server. The parallelism across all servers is the key to keeping latency low: the client doesn't wait for server 0 before starting with server 1.

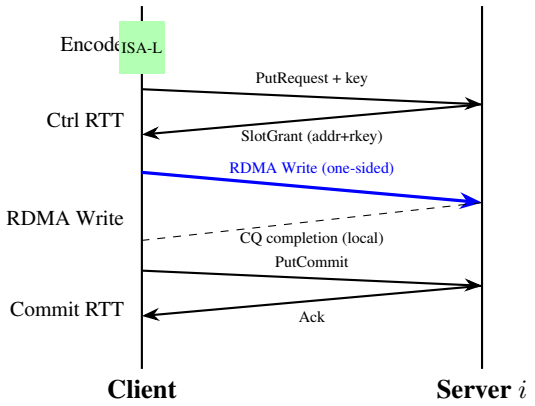

Figure 2: PUT message sequence for one server. Phases 2 and 4 are issued to all $k+m$ servers in parallel; phase 3 RDMA writes are also parallel.

8 GET Protocol

A GET also proceeds in phases, but with more flexibility because the client only needs k of the $k + m$ shards.

Phase 1 — Control RTT. The client sends `GetRequest` to all alive servers (consulting the `dead[]` bitmap from the watcher thread). Dead servers are skipped entirely. Each alive server looks up the shard key and replies with `SlotInfoMsg`:

- If `status = kNotFound`: the shard isn't on this server (shouldn't happen in normal operation, but handled).
- If `on_disk = 0`: replies with the RDMA pool address directly.
- If `on_disk = 1`: reads the shard from disk into a staging slot, then replies with the staging slot's address.

Phase 2 — RDMA Read. The client issues one-sided RDMA reads for the first k available shards. For disk-backed shards, the read targets the staging buffer on the server. After all k reads complete, the client sends `DiskRelease` messages for any disk-backed shards to free their staging slots.

Phase 3 — Decode. If all k data shards (indices 0 through $k - 1$) are present, the decoded result is just concatenation truncated to the original object length. If some data shards are missing (because servers are dead or returned `kNotFound`), the client falls back

to the ISA-L decoder with whichever k shards it has.

The dead server detection is what enables the degraded read path without coordinator involvement. The client's watcher thread updates `dead[i]` when a `DEAD` notification arrives. The GET code checks this bitmap before deciding which servers to contact and which shard indices to pass to the decoder.

9 DELETE Protocol

DELETE is the simplest operation. The client sends `DelRequest` to all $k+m$ servers in parallel. Each server:

1. Looks up the shard key in `key_to_slot`.
2. If found: frees the RDMA or disk allocation and removes the key.
3. Replies with `Ack(kOk)` or `Ack(kNotFound)`.

The client waits for all acks. There's a subtle issue with concurrent writes and deletes, which we discuss in Future Work: if a write and delete race, the current protocol doesn't provide any atomicity guarantee beyond what the commit phase gives you for writes. In practice our workloads don't overlap these, but a production system would need a version number or epoch in the slot metadata.

10 Coordinator and Failure Detection

10.1 Protocol

The coordinator speaks a simple line-based text protocol over TCP. The four commands are:

`REGISTER <hex_addr>` Server startup. The coordinator assigns a registration index, stores the server's fabric address, and spawns a ping thread. Returns `OK <idx>`.

`LIST` Client discovery. Returns the current alive server list with k/m parameters, or `WAIT n/2` if fewer than 2 servers are registered.

`WATCH` Client watch connection. The coordinator keeps this connection open and pushes `DEAD <idx>` lines when servers die.

`STATUS` Diagnostic. Returns per-server state including key counts and RDMA pool utilization.

10.2 k/m Computation

The coordinator computes k and m from the current alive server count n:

$$m = \lfloor n/3 \rfloor, \quad k = n - m$$

This keeps parity overhead at roughly 33% regardless of cluster size. For a 2-node cluster, you get $(k = 1, m = 1)$ which can survive one failure but doesn't really scale. At 3 nodes you get $(2, 1)$, at 6 nodes $(4, 2)$, and so on. The coordinator re-computes k/m any time a server joins or is declared dead, but doesn't actively tell existing clients to reconfigure — new clients get the updated parameters when they `LIST`, and existing clients keep using the k/m they were initialized with. This is fine for a research system where cluster topology is relatively static.

10.3 Failure Detection

The coordinator pings each registered server every 2 seconds. If a server doesn't reply within 1 second, the ping is counted as missed. After 3 consecutive misses (roughly 6 seconds of silence), the server is declared dead:

```
constexpr int kPingIntervalMs = 2000;
constexpr int kPingTimeoutMs  = 1000;
constexpr int kMaxMissed      = 3;
```

Each PONG reply from a server includes its current key count, RDMA bytes used, and total pool size. The coordinator tracks these and shows them in STATUS output, which gives a live view of storage pressure across the cluster without needing separate monitoring infrastructure.

When a server is declared dead, the coordinator

broadcasts a `DEAD <idx>` message to all active watch connections. Clients receive this on their watcher thread, mark `dead[pos]` as true for the corresponding server position, and from that point on skip that server in GET and PUT operations.

10.4 Single Point of Failure

The coordinator is a single-threaded TCP server with no replication. If it crashes, existing clients can still do PUTs and GETs (they already know the cluster layout), but they stop receiving failure notifications and new clients can't connect. This is acceptable for a research system and a natural target for future work — replacing it with a Raft-replicated state machine would remove the last SPOF.

11 RDMA Network Layer

11.1 libfabric and Fabric Objects

We use libfabric's `verbs` provider, which maps to InfiniBand verbs underneath. The main fabric objects are:

- `fid_fabric`: top-level fabric handle
- `fid_domain`: protection domain (analogous to IB PD)
- `fid_ep`: connected endpoint (both client and server)
- `fid_pep`: passive endpoint (server-side, for accepting connections)
- `fid_cq`: completion queue (1024 entries, `FI_CQ_FORMAT_DATA`)
- `fid_eq`: event queue (connection events: connect, accept, shutdown)
- `fid_mr`: memory region registration handle

The fabric is initialized with:

```
fi_getinfo() with hints:
    ep_type = FI_EP_MSG       // connected, reliable
    caps    = FI_MSG | FI_RMA // messaging + remote memory access
    mode    = FI_CONTEXT
    domain_attr.mr_mode =
        FI_MR_LOCAL | FI_MR_ALLOCATED |
        FI_MR_PROV_KEY | FI_MR_VIRT_ADDR
```

`FI_MR_VIRT_ADDR` means that RDMA operations use virtual addresses on the remote side rather than offset-from-zero, which simplifies the address calculation on the client (the server just reports its buffer's virtual address directly).

11.2 Connection and Operation Structs

```
struct Connection {
    struct fid_ep *ep;
    struct fid_cq *cq;
    struct fid_eq *eq;
    bool           virt_addr_mode;
    std::unordered_map<void *, struct fid_mr *> mr_map;
    std::deque<RdmaOp *> pending;
};

struct RdmaOp {
    RdmaOpType  type;      // kRecv, kSend, kWrite, kRead
    Buffer     *buf;
    size_t      len;
    uint64_t    remote_addr;
    uint64_t    remote_key;
    std::function<void(Connection &, RdmaOp &)> cb;
};
```

Each RDMA operation is described by an `RdmaOp` and carries a completion callback. The `Poll()` method drains the completion queue and invokes callbacks for completed operations:

```
void Connection::Poll() {
    struct fi_cq_data_entry cqe[kCQBatch];  // kCQBatch = 16
    int n = fi_cq_read(cq, cqe, kCQBatch);
    if (n > 0) {
        for (int i = 0; i < n; i++) {
            auto *op = (RdmaOp *)cqe[i].op_context;
            if (op->cb) op->cb(*this, *op);
        }
    }
}
```

Memory registration is done per-operation type. Control buffers are registered with `FI_SEND | FI_RECV`. Data buffers are registered with `FI_READ | FI_WRITE | FI_REMOTE_READ | FI_REMOTE_WRITE`.

11.3 Buffer Alignment

All buffers are allocated with 128-byte alignment (`kBufAlign = 128`) to satisfy NIC DMA alignment requirements and to improve cache behavior on the encoding path. In practice, RDMA hardware generally requires physical address alignment rather than virtual, but the over-alignment doesn't hurt and catches cases where the verbs provider is stricter than expected.

12 Client Library

12.1 ErasureClient

The top-level client object:

```
struct ErasureClient {
    int k, m;
    ErasureCoder ec;
    std::vector<ServerConn> servers;           // k+m connections
    std::unique_ptr<std::atomic<bool>[]> dead; // dead[i] per server
    PhaseTimes last_put_phases;
    PhaseTimes last_get_phases;

    int                         coord_fd;
    std::unordered_map<int,int> reg_idx_to_pos; // coord idx -> pos
    std::thread                 watcher_thread;
};
```

The `dead[]` array is indexed by position in `servers[]`, not by coordinator registration index. The `reg_idx_to_pos` map translates when a `DEAD` notification arrives.

12.2 Coordinator Discovery

On startup, the client connects to the coordinator and sends `LIST`:

```
ClusterInfo coord_discover(const char *host, int port) {
    int fd = connect_tcp(host, port);
    send(fd, "LIST\n", 5, 0);

    // Parse: K 2\nM 1\n0 <addr>\n1 <addr>\nEND\n
    ClusterInfo info;
    parse_response(fd, &info.k, &info.m,
                   &info.addrs, &info.reg_idxs);
    close(fd);

    // Open watch connection (kept open)
    info.coord_fd = connect_tcp(host, port);
    send(info.coord_fd, "WATCH\n", 6, 0);

    return info;
}
```

After discovery, the client establishes RDMA connections to all $k + m$ servers and starts the watcher thread on the watch TCP connection.

12.3 Phase Timing

Every PUT and GET records timing for each phase:

```
struct PhaseTimes {
    uint64_t encode_ns;     // ISA-L encode or decode
    uint64_t ctrl_rtt_ns;   // control message round-trip
    uint64_t rdma_ns;       // RDMA write or read
    uint64_t commit_rtt_ns; // commit ack round-trip (PUT only)
};
```

These are exposed via the Python binding as `last_put_phases()` and `last_get_phases()`, which return tuples of microsecond values. The benchmark scripts use these to produce the per-phase breakdowns shown in the evaluation.

12.4 Python Bindings

The Python module is built with pybind11 and exposes a `Client` class:

```
import rdmastorage

client = rdmastorage.Client("node0", port=7777)
client.put("mykey", b"hello_world")
data = client.get("mykey")
client.delete("mykey")

# Phase timing (microseconds)
enc, ctrl, rdma, commit = client.last_put_phases()
```

The `put()` method accepts Python `bytes` objects and handles the conversion to/from `uint8_t *` internally. The `get()` method returns a `py::bytes` object. All RDMA operations happen inside the C++ layer; the Python GIL is released during blocking RDMA polls.

13 Server Implementation

13.1 Thread Model

The server uses a thread-per-connection model:

```
for (;;) {
    Connection conn = net.Accept();
    std::thread([c = std::move(conn), &shared]() mutable {
        handle_client(std::move(c), shared);
    }).detach();
}
```

Each connection gets its own thread. All shared state (slot metadata, `key_to_slot`, allocators) is protected by a single mutex. `SharedState` also holds the staging buffer with its per-slot atomic flags, which don't need the mutex.

```
struct SharedState {
    std::mutex                          mu;
    SlotEntry                           slots[kMaxSlots]; // 65536
    int                                 next_slot = 0;
    std::unordered_map<std::string, int> key_to_slot;
    MultiSlabAlloc                      multi_slab;
    Buffer                              staging_buf;      // 64 MB
    std::atomic<bool>                   staging_in_use[64];
    int                                 disk_fd = -1;
    PoolAlloc                           disk_alloc;       // 64 GB
};
```

13.2 PutRequest Handling

The full flow under the mutex:

1. Construct the shard key (`key + '\0' + shard_idx`).
2. If the key already exists, free its old allocation (overwrite path).
3. Allocate a new slot from `slots[]`.
4. Try `multi_slab.alloc(shard_len)`.
5. If that fails, try `disk_alloc.alloc(shard_len)`.
6. Register any newly created slabs on the current connection.
7. Store a `PendingPut` record keyed by `slot_idx`.

Outside the mutex, allocate a staging slot if needed (atomic scan, no lock). Then send `SlotGrantMsg` with the appropriate address and rkey.

13.3 PutCommit Handling

Look up the `PendingPut` by `slot_idx`. If `on_disk`:

1. `pwrite(disk_fd, staging[slot] + off, shard_len, disk_off)`.
2. Release the staging slot (atomic CAS).

Under the mutex: finalize the slot entry, add the key to `key_to_slot`, remove the `PendingPut`. Send `Ack`.

13.4 GetRequest Handling

Look up the shard key under the mutex. If found on disk:

1. Allocate a staging slot.
2. `pread(disk_fd, staging[slot] + off, shard_len, disk_off)`.
3. Reply with staging address and `on_disk = 1`.

If found in RDMA pool: reply with the slab address directly.

The slab registration check happens inside the mutex on both PutRequest and GetRequest handlers to ensure any slabs created by concurrent PUTs are visible before we try to hand out their addresses.

14 Evaluation

14.1 Setup

All SHARD benchmarks ran on CloudLab `xl170` nodes (dual Intel E5-2640v4, 128 GB DDR4, Mellanox ConnectX-4 25 Gbps NIC) using RoCE v2

over a single switch with no oversubscription. For the S3 comparison, we deployed a Rust Lambda function in `us-east-1` (AZ `use1-az4`) and used SDK-level interceptors to capture only network-plus-S3 processing time, excluding any client-side overhead. Both systems ran 1000 operations per object size from 1 KB to 1 MB.

14.2 Latency vs. S3 Express One Zone

Table 2 shows the full latency breakdown. At every size we tested, SHARD's median latency is at least two orders of magnitude lower than S3 Express. The gap at 64 KB is particularly striking: 127 μs versus 9.9 ms for PUT, and 82 μs versus 5.3 ms for GET (roughly 80$\times$ in both cases). Even at 1 MB, where ISA-L encoding starts to add measurable time, we're still about 6$\times$ faster at the median.

The one place our numbers look worse relative to S3 is p99 at small object sizes (1 KB and 4 KB), where we occasionally spike into the low milliseconds. This is most likely RDMA queue pressure: when many small writes are in flight simultaneously, completion queue entries can pile up and individual operations see high tail latency. Even so, those spikes are within S3's normal operating range, so the comparison still favors SHARD.

Table 2: PUT and GET latency: S3 Express One Zone (ms) vs. SHARD (μs). S3 figures use p90; SHARD figures use p95.

	PUT				GET			
Size	avg	p50	p90/p95	p99	avg	p50	p90/p95	p99
S3 Express One Zone (milliseconds)								
1 KB	9.90	9.87	11.62	17.65	5.54	5.33	5.85	20.72
4 KB	8.59	8.37	9.81	14.19	5.18	5.10	5.71	7.84
16 KB	8.67	8.55	9.63	11.99	5.27	5.18	5.56	10.68
64 KB	10.00	9.95	11.37	15.66	5.34	5.33	5.61	8.21
256 KB	11.42	11.20	12.88	18.52	3.76	3.60	4.12	7.79
1 MB	15.29	15.04	16.67	24.41	3.90	3.72	4.44	7.59
SHARD (microseconds)								
1 KB	97.1	20.9	33.8	1934.6	14.2	13.5	20.8	24.1
4 KB	130.5	26.2	37.3	5210.6	17.3	17.1	18.1	26.9
16 KB	46.8	46.5	48.8	54.6	40.7	29.6	43.4	331.3
64 KB	199.6	127.2	138.7	3690.5	88.7	81.7	94.1	381.1
256 KB	453.1	449.4	456.9	595.3	297.4	295.4	299.3	391.5
1 MB	2666.5	2659.9	2704.1	2890.7	1176.1	1141.2	1427.2	2018.9

14.3 Phase Breakdown

Figure 5 breaks the total latency into its component phases. For PUT, the RDMA write dominates at large object sizes since it scales directly with the amount of data being transferred across three servers. At small sizes (1 KB, 4 KB), the ctrl-RTT and commit-RTT are a larger fraction of total time since the actual data transfer is nearly instant. Encode time is negligible at small sizes but grows noticeably at 1 MB as ISA-L computation scales with payload size.

The GET breakdown tells a similar story: RDMA read time dominates at large sizes, while ctrl-RTT takes a larger share of the total at small ones. Decode time for GET is comparable to encode time for PUT at each size, which makes sense since ISA-L uses the same routines.

One takeaway is that there's relatively little headroom to optimize the small object case through better data transfer, because the ctrl-RTT and commit-RTT together already account for most of the latency. Reducing those would require either pipelining requests

or cutting the number of round trips, which would require protocol changes.

14.4 YCSB Workload Sweep

We tested workloads A (50/50 read/write), B (95% reads), C (100% reads), D (95% reads, newest keys), and F (read-modify-write) across cluster sizes from 2 to 6 nodes using a 64 KB object size and a Zipfian key distribution with theta = 0.99.

p50 latency stays in the 12–27 μs range regardless of cluster size. Workload C is consistently the fastest since GETs skip the commit round-trip. Write-heavy workloads like A and F are a bit slower since they need all $k + m$ servers to acknowledge the commit. The gap between workloads is smaller than we expected, which suggests the network RTT dominates over the per-operation CPU work at this cluster size.

p99 latency spikes noticeably at $N = 4$. This is where the erasure parameters shift from $(2, 1)$ to $(3, 1)$, adding one more server to contact per PUT. The p99 settles back down at $N = 5$ and $N = 6$ as the extra server becomes normalized in the tail distribution. The spike is a natural consequence of the discrete k/m steps — with more cluster sizes tested, you'd see a sawtooth pattern.

Throughput (Figure 3) trends down gradually with cluster size, from around 5,600 ops/sec at $N = 2$ to about 5,000 at $N = 6$. This is expected: more nodes means more parallel RDMA writes per PUT, so each operation takes slightly longer. Workload B degrades the least because most of its operations are reads, which only touch k servers.

14.5 Latency Scaling with Object Size

Figure 4 shows how PUT and GET latency scale across all percentiles as object size increases from 1 KB to 1 MB. The pattern is roughly linear at large sizes (dominated by RDMA transfer time) and flat at small sizes (dominated by round-trip overhead). The divergence between p50 and p99 at small sizes is the queue pressure effect mentioned above.

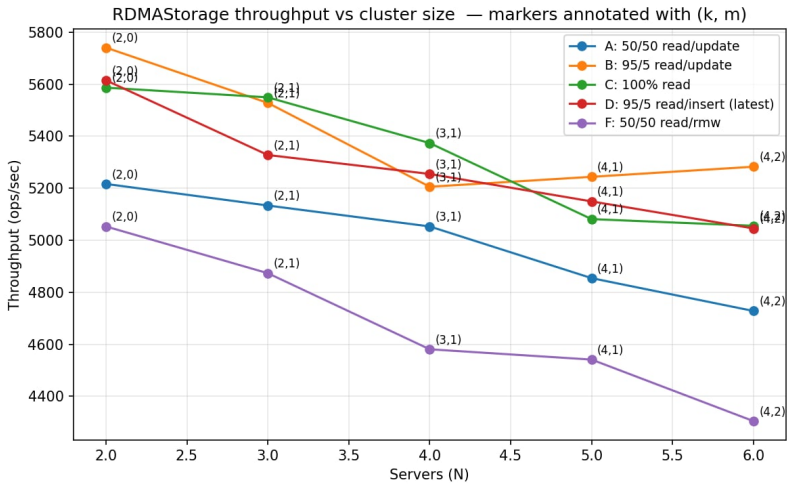

Figure 3: Throughput (ops/sec) vs. cluster size across YCSB workloads. Markers are annotated with the erasure coding parameters (k, m) in use.

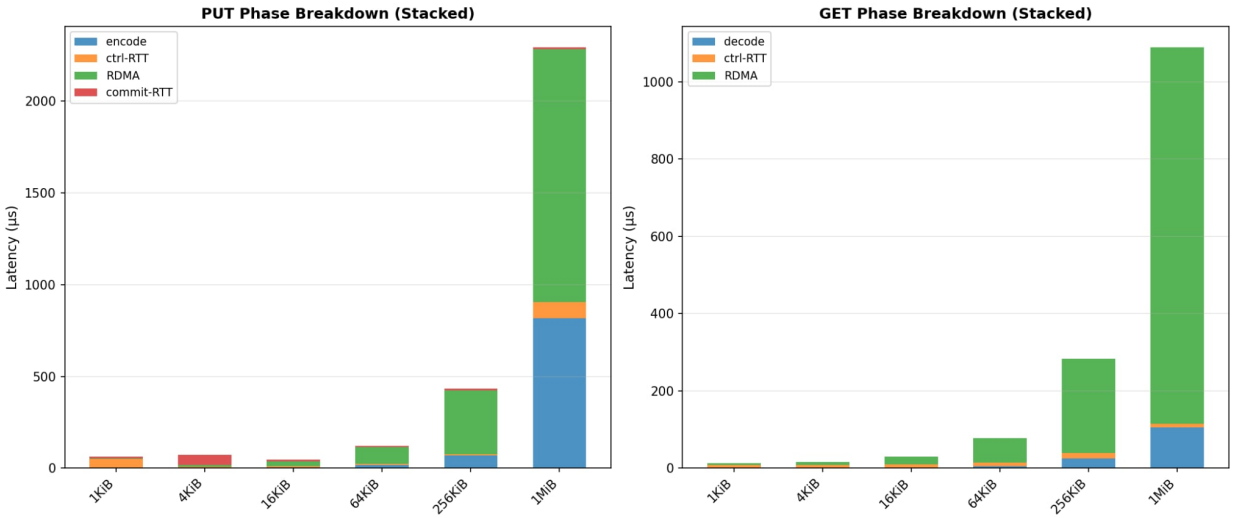

Figure 4: PUT and GET latency (avg/p50/p95/p99) vs. object size on a 3-node SHARD cluster.

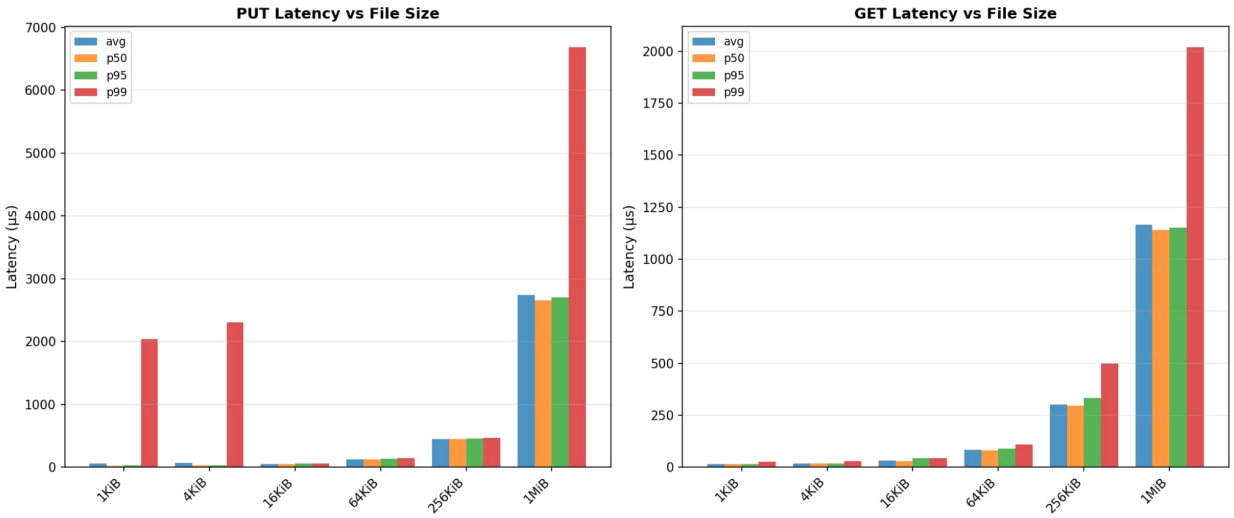

Figure 5: Per-phase latency breakdown for PUT and GET across object sizes. PUT phases: encode, ctrl-RTT, RDMA write, commit-RTT. GET phases: ctrl-RTT, RDMA read, decode.

15 Cost Analysis

RDMA storage only makes sense if you're actually going to save money compared to S3 at some throughput level. Here's what the math looks like.

The cheapest EFA-enabled EC2 instance is the c7g.16xlarge (Graviton 3, 64 vCPU, 128 GiB RAM, 30 Gbps EFA) at $2.32/hr on-demand. A 3-node cluster runs $5,012/month. You'd replace the coordinator with AWS Cloud Map ($0.10/node/month plus $1/million API calls), adding almost nothing. With the $(k = 2, m = 1)$ erasure code, each node's 128 GiB gives 256 GiB of usable RDMA storage, with 1.5× overhead versus 3× for full replication.

On the S3 side, current pricing is $0.00113 per 1,000 PUTs and $0.00003 per 1,000 GETs plus data transfer fees. For a 50/50 workload at 64 KB, that works out to about $0.70 per million operations. At our measured throughput of ∼5,000 ops/sec, S3 would cost around $9,000/month in request fees alone,

nearly double the EC2 bill, with 80× worse median latency.

The break-even point is around **2,800 ops/sec**: below that, S3 is cheaper because you're not paying for idle instances; above it, SHARD wins on both cost and latency. Our system clears that bar easily across every workload we tested. If you need sub-millisecond storage at any meaningful throughput, this approach makes sense economically, not just technically.

Table 3 summarizes the comparison.

Table 3: Monthly cost comparison at 5,000 ops/sec (64 KB, 50/50 read/write).

System	Infra cost	Request cost	Total/month
S3 Express One Zone	$0	≈$9,072	≈$9,072
SHARD (3-node c7g.16xlarge)	$5,012	≈$0	≈$5,012

16 Related Work

FaRM [1] is the most direct ancestor. FaRM uses one-sided RDMA for both reads and writes across a primary-backup scheme, targeting sub-microsecond latency on InfiniBand. Our system makes a different trade-off: we use erasure coding instead of replication for fault tolerance, which costs more per operation (because all $k + m$ shards must be written) but uses less storage. FaRM also assumes a more tightly controlled cluster; our coordinator design is simpler and targets CloudLab/EC2-style environments.

HERD [2] showed that two-sided RDMA sends can be faster than one-sided operations for small key-value operations because they avoid the client-side memory registration and avoid cache pollution on large reads. For our use case with objects up to 1 MB, one-sided reads and writes are clearly the right choice.

Pilaf [3] built a KV store on one-sided RDMA reads (cuckoo hashing in server memory). The key insight was that reads-only operations could completely bypass the server CPU. Our GET protocol takes the same approach but extends it to support erasure-coded objects of variable size, which requires more metadata management.

RAMCloud [1] targets complete in-memory durability with log-structured storage and RDMA-based replication. It achieves lower per-operation latency than SHARD for small objects but at higher cost (full replication) and with more complex crash recovery.

Azure Erasure Coding [5] is a production deployment of Reed-Solomon erasure codes at large scale. Their work motivated our choice of ISA-L and validated the storage efficiency argument, though their system runs on conventional TCP storage networks rather than RDMA.

17 Future Work

Several directions would be natural next steps.

Concurrency control. Right now there's no protection against concurrent writes to the same key from multiple clients. Two clients writing the same key simultaneously could interleave their shard commits, producing a corrupt or mixed object. The simplest fix would be per-key versioning in the slot metadata and an optimistic concurrency check at commit time. A heavier approach would be client-level leases coordinated through the coordinator.

Multi-drive disk tier. The disk spill tier uses a single NVMe file per server. With multiple drives, shards could be striped across them for higher sequential bandwidth, similar to RAID-0 but without the failure risk since erasure coding already handles node-level faults.

Replicated coordinator. The coordinator is the one true single point of failure. Replacing it with a Raft-replicated state machine (or using an existing system like etcd) would make the cluster fully fault-tolerant. The protocol is simple enough that the Raft log entries would just be REGISTER/DEAD events.

Client-side caching. Hot keys are expensive when every GET requires a control round-trip plus RDMA

reads from multiple servers. A client-side cache with a TTL-based invalidation scheme could serve repeated reads for popular keys at near-zero latency. The degraded-read path already handles stale shard locations gracefully, so cache invalidation on failures is less of a concern than in a traditional cached system.

Adaptive erasure parameters. The current ($m = \lfloor n/3 \rfloor$) formula is fixed. In practice, you might want more parity when the cluster is small (where a single failure is a larger fraction of total storage) and less parity when it's large (to get better write throughput). The coordinator already computes k/m dynamically from n; the formula could be made configurable or adaptive based on observed failure rates.

References

[1] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro, "FaRM: Fast Remote Memory," in *Proc. NSDI*, 2014.

[2] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA efficiently for key-value services," in *Proc. SIGCOMM*, 2014.

[3] C. Mitchell, Y. Geng, and J. Li, "Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store," in *Proc. USENIX ATC*, 2013.

[4] A. Das, I. Gupta, and A. Motivala, "SWIM: Scalable Weakly-consistent Infection-style Process Group Membership Protocol," in *Proc. DSN*, 2002.

[5] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure coding in Windows Azure Storage," in *Proc. USENIX ATC*, 2012.

[6] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proc. ACM SoCC*, 2010.