David Cloud: A Scalable Distributed Storage System with KVS-Backed File Services

Team 06

Forest Ho-Chen   Dan Kim   Stephen Kwak   David Zhan

May 24, 2026

1. System Design Overview

The project implements a fault tolerant, distributed cloud platform that provides webmail, file storage, and user management services. The system is composed of four major subsystems: a frontend load balancer, multiple frontend servers, a backend coordinator, and a set of KVS backend storage workers. The high-level overview of a request flow is as follows: the client connects to the load balancer which allocates a live server via round robin scheduling and redirects the client to it. The frontend server then accepts the connection, parses the HTTP/1.1 request and authenticates by validating the client's cookie against stored session on KVS. To fulfill the request, the frontend asks the backend coordinator to hash the row key to determine the correct tablet and primary KVS storage worker. The KVS worker executes the operation against its in-memory state (sending ACK), replicating this to its secondaries while checkpointing tablets and write-ahead logging.

1.1 Frontend

1.1.1 Load Balancer

The first component of the frontend is the load balancer which routes clients to live frontend servers. The load balancer by default listens on port 8000, running a background thread that TCP connects probes every two seconds to detect live frontends. On every client request, the balancer reads the HTTP request, picks a healthy frontend via round-robin scheduling, and sends a 302 redirect to that server.

1.1.2 Frontend Server

Once redirected by the load balancer, the frontend server renders its components to the client using four main code components: the main, API routes, the backend client, and HTML.

1. The main component accepts and parses incoming HTTP/1.1 requests, actively validating and reading the request body before passing a clean HTTPRequest object to the router. This validation step checks for valid HTTP versions, Content-Length configuration, and instills a timeout for potentially broken calls. It also manages keep-alive connections, allowing the browser to reuse the same TCP connection for multiple requests. This also runs a SMTP server on a separate thread for SMTP requests from Thunderbird.
2. The API routes module authenticates each session by validating the client's cookie against the KVS and sends this request to the correct handler for an area of application the client desires. Each of the routes that we implement are matched by (path, method) in which all but Login, Logout, Register and Admin paths are guarded by session authentication. Our application supports routes for user login management, account settings, webmail, file stroage, and admin console.
3. The backend client is the communication layer between the frontend server and the storage cluster. The client allows for a coordinator lookup for every KVS operation, connecting the frontend to the coordinator, receiving an ordered list of replica endpoints. This allows for a connection to the primary node, supporting KVS operations like KVS GET, PUT, delete column, delete row, and CPUT.

4. Finally, the HTML component takes in the data from each API route handler to produce the final HTML response sent back to the browser. These pages consist of login, register, home, webmail, drive, games, account settings and admin console. Styling is managed via CSS.

To briefly go over our session handling, they are initially created at login and registration. When a user submits their credentials, the frontend hashes the password and then compares it against the stored hash in the KVS (Check table 5.2) and on success, the server generates unique session token via hash and storing it to the SessionToUser table. On every subsequent authenticated request, the router reads the provided cookie, and calls KVS get on the session token to allow for redirection to a specified path.
Another integral portion was chunked transfer encoding on both inbound and outbound sides. On inbound, when a client sends a request with "Transfer-Encoding: chunked" header, the server reads each chunk by reading exactly chunk size number of bytes until you reach the zero-length terminal chunk. These decoded chunks are reassembled before being passed to the router. On outbound, the responses are automatically sent with chunk transfer when the message is over 20 bytes. The server splits the body into fixed-size chunks, prefixes each with its size in hexidecimal, and ends with a zero-length chunk.

1.2 KVS Storage Worker

1.2.1 Data Model and Storage

Each worker node manages an in-memory nested hash table that maps each key to another unordered map, which maps column keys to string values. This table is partitioned by tablet, where a worker is responsible for a subset of tablets, and this responsibility is managed by the coordinator. Ownership for an object is validated on every write operation before any locking or replication occurs.

1.2.2 Client-Facing Protocol

Please check appendix for further details about the protocol.

1.2.3 Replication and Consistency Protocol

All mutating operations follow the same two-phase commit (2PC) structure. The primary and secondary roles for this procedure are as follows:

Primary side:

1. The primary checks whether it owns the row. If not, it immediately sends an `ERR` response to the client.
2. The primary acquires the lock on the row.
3. The primary checks whether it can perform the operation locally. If not, it immediately sends an `ERR` response to the client.
4. The primary appends a `PREPARE` record to its local WAL.
5. The primary sends `PREPARE` messages to all secondaries in parallel and waits for `ACCEPT` responses. If any secondary sends back an `ERR`, the primary sends an `ERR` response to the client.
6. If all secondaries accept, the primary appends a `COMMIT` record to its local WAL.
7. The primary sends `COMMIT` messages to all secondaries in parallel. If any node dies during this phase, the primary coordinates rollback handling.
8. The primary applies the update locally.
9. The primary releases the row lock.
10. The primary sends an `OK` response to the client.

Secondary side:

1. The secondary tries to acquire the required row and cell locks.
2. The secondary checks whether it can perform the operation locally. If not, it sends an ERR response to the primary.
3. The secondary appends a PREPARED record to its local WAL.
4. The secondary sends an ACCEPT response to the primary and waits for the primary to reply with COMMIT or ABORT.
5. If the primary sends COMMIT, the secondary appends a COMMITTED record to its local WAL.
6. The secondary applies the update locally.
7. The secondary releases its locks.
8. The secondary sends an ACK response to the primary.

1.2.4 Locking and Concurrency

The worker uses cell-level locking for fine-grained concurrency. A CellLockEntry, which wraps a pthread_mutex_t and a reference count, is stored in a global map keyed by row + "\0" + col. DELETE ROW acquires the row-level lock first, then locks all affected cell entries in sorted column order to avoid deadlock. Reference counting ensures that lock entries are destroyed only when no threads hold a reference, keeping memory usage proportional to active contention rather than the total key space.

1.2.5 Write-Ahead Logging

Every mutating operation is durably logged to a per-tablet WAL file, log/log_<node_id>_<tablet_id>.txt, before any network or in-memory state is modified. Each WAL record has the following format:

```
WAL <state> <tablet_id> <epoch> <op_id> <node_id> <primary_id>
    <num_secondaries> [<sid>...]
    <rowlen> <collen> <v1len> <v2len>\n<row><col><v1><v2>\n
```

The state field progresses through BEGIN_<OP> to PREPARE_<OP> on the primary, or PREPARED_<OP> on a secondary, then to COMMIT_<OP> or COMMITTED_<OP>, and finally to END_<OP>. An fsync is issued after every record to guarantee crash durability. After a checkpoint completes, wal_rotate() truncates the WAL for that tablet to reclaim disk space.

1.2.6 Checkpointing and Recovery

The coordinator drives a two-phase global checkpoint every 20 seconds across all alive nodes. In the prepare phase, PREPARE_CHECKPOINT <id>, each node confirms that it has no in-flight operations or pending prepared transactions. If any node is busy, the round is abandoned. In the commit phase, COMMIT_CHECKPOINT <id>, the checkpoint gate closes: new client operations block on a condition variable until the gate reopens. The node then serializes its in-memory tablet data to checkpoint/<node_id>_<tablet_id>.txt and rotates its WAL. The DONE_CHECKPOINT command reopens the gate and unblocks queued client requests.

On startup, each node executes a recovery phase by contacting its primary for each assigned tablet. If the local and primary checkpoint IDs match, only the incremental WAL records since the local last_op_id are fetched and replayed using FETCH_RECOVERY_LOG. If the primary holds a newer checkpoint, the full snapshot plus WAL are transferred using FETCH_RECOVERY_FULL, and the local state is replaced entirely. This ensures that secondaries converge to the primary's state before serving traffic.

1.3 Coordinator

1.3.1 Responsibilities

The coordinator is the central control plane of the storage cluster. It has three primary responsibilities: routing client requests to the correct storage nodes, managing the liveness of worker nodes via heartbeats, and distributing load evenly across nodes through tablet ownership.

The coordinator listens on a configurable port, which defaults to `9000`, and spawns a dedicated `pthread` per accepted connection. Each connection's thread reads a single CRLF-terminated line from the client and dispatches it to one of the command handlers. Two background threads also run for the lifetime of the process: a heartbeat reaper that periodically evicts dead nodes, and a checkpoint coordinator that orchestrates global snapshots across all alive workers.

The `LOOKUP` command is the most critical path. Given a rowkey, the coordinator hashes it modulo the tablet count to determine which tablet owns the key, then returns the primary and all alive replica endpoints for that tablet, along with the current epoch. Frontend servers call this before every KVS operation so they always connect to the correct primary.

`GET_NODE_STATUS` returns a full view of every registered node, including its liveness, which tablets it primaries, and which tablets it holds as a secondary. The admin console uses this to display cluster health in the browser.

`ADMIN_KILL` marks a node dead in the coordinator's state, sends it an `ADMIN_SHUTDOWN` TCP command, increments the epoch, and immediately rebalances tablet ownership. `ADMIN_RESTART` forks a new child process using the launch command registered for that node in the workers file, which causes the worker to reboot and re-register via heartbeat.

1.3.2 Heartbeat Management

Each KVS worker sends a `HEARTBEAT <nodeid>` to the coordinator on a regular interval over a persistent TCP connection. On receipt, the coordinator updates `last_heartbeat_ms` for that node. A background reaper thread wakes every `500 ms` and sweeps all nodes: any alive node whose last heartbeat was more than `4000 ms` ago is marked dead. When any liveness change occurs, the epoch is incremented and tablet assignments are recomputed. If a previously dead node later sends a heartbeat, it is marked alive again, the epoch increments, and tablets are redistributed to include it.

1.3.3 Load Distribution via Tablets and the Hash Ring

The coordinator partitions the key space into a configurable number of tablets, which defaults to `16`. Every rowkey is mapped to a tablet by computing `hash64(rowkey) % num_tablets`. A tablet is the unit of ownership: each tablet is assigned to a primary node and up to `replica_count - 1` secondary nodes.

Tablet-to-node assignment uses consistent hashing with virtual nodes. When `recompute_tablet_assignments_locked()` runs, it performs the following steps:

1. Collect all currently alive node IDs.
2. For each alive node, generate `VIRTUAL_NODES_PER_NODE` virtual ring entries, which defaults to `64`. Each virtual node is keyed by hashing the string `"<node_id>:<v>"` with `std::hash<std::string>`, producing a 64-bit position on the ring.
3. Sort all virtual node entries by their 64-bit position, forming the consistent hash ring.
4. For each tablet `t`, hash the string `"tablet:<t>"` to get its position, then walk clockwise from that position to collect `min(replica_count, alive_node_count)` distinct physical nodes. The first node encountered becomes the primary, and subsequent distinct nodes become secondaries.

The 64 virtual nodes per physical node ensure that when a node joins or leaves the cluster, only a proportional fraction of tablets are reassigned, avoiding full reshuffling. The primary for a tablet is always `tablet_to_replicas[t][0]`, and this ordering is preserved in every `LOOKUP` response, so frontends always know which endpoint to write to.

2. Features Implemented

2.1 Core Features

- **Frontend web server**: Users can go to the frontend servers directly, or use the frontend load balancer to get to a frontend server that is more free. Following this, they can go to the mail tab, the compose tab, the drive, or the admin console. Using the information from the backend, the router calls the HTML maker that generates the HTML code to display the page. This HTML maker combines the data into a format that is human readable and not cluttered.

- **Webmail**: Our email service allows users to send emails to other DavidCloud users. The email service also allows users to receive emails from external SMTP clients on the same machine such as Thunderbird, and allows users to send emails outside of DavidCloud through the DNS Lookup to MX records external system or the email relay system. Additionally, we have the SMTP server and client. These work with the frontend router for sending external emails and the backend client for receiving external emails from Thunderbird, Telnet, or some other SMTP server. If an email is received through the SMTP server, it will be stored in the backend by using the backend client and searching for the user in the KVS. If a DavidCloud user wants to send an email out, the SMTP server and client will either do a DNS lookup and MX records or use an email relay.

- **Cloud storage**: The DavidDrive service provides a distributed cloud-storage system built on top of our replicated key–value store. Each authenticated user is given a private hierarchical filesystem supporting folder navigation, file upload/download (of up to 500MB per file), inline preview for common formats such as text, images, and PDFs, and standard filesystem operations including create, rename, move, and delete. Internally, the filesystem is implemented using an inode-style abstraction layered over the KVS, where directories and files are represented as rows containing metadata, directory entries, and file contents. To support scalability, small files are stored inline while larger files are transparently chunked into fixed-size segments across the storage cluster. All file operations are routed through the coordinator to the correct primary replica, ensuring consistency through replicated writes, write-ahead logging, and recovery support. DavidDrive additionally enforces per-user isolation through root inode namespaces and normalized path resolution, preventing unauthorized traversal outside of a user's filesystem.

- **Replication**: This implements a primary based replication model with 2-PC protocol for consistency. When the frontend sends a write operation to the primary KVS worker, the primary begins by confirming it still owns the tablet requested by the coordinator. If it does, it acquires a **cell level** lock on the row and column being modified (cell level granularity). Then it appends a PREPARE record to its local write-ahead log (WAL), sending out the PREPARE message to all secondary replicas in parallel. Each secondary node receives the PREPARE, acquires its own cell lock to validate the operation, and responds ACCEPT to the primary. With all ACCEPTS from the secondaries, the primary appends a COMMIT to its WAL and sends to all secondaries. Secondaries do the same, updating its WAL and updating locally to send ACK back to the primary.

- **Check-pointing**: The check-pointing has a central coordinator, which triggers a distributed checkpoint across all alive KVS nodes every 20 seconds. This process contains 3 main phases: prepare, commit and done. In the prepare phase, the coordinator sends a prepare API with a checkpoint ID to all alive nodes. Each node then checks for any in-flight operations, where check-pointing only proceeds if there are none. Once nodes are ready, the coordinator proceeds to the commit phase where it sends a commit API to all nodes. The node then takes a snapshot of its entire tablet state and writes it to disk. Note that checkpoint files are stored separate per node and per tablet for independent reads and restoration. Once written, the nodes respond via ACK, and with all in unison, the coordinator sends "done" at which point each node unblocks any write operations that were waiting behind the checkpoint gate and resumes normal operation. To prevent any overwrites, the workers acquire a "CheckpointGateGuard" at the start of every write operation, which blocks operations during the prepare-to-commit window and tracks count of in-flight operations.

- **Recovery**: This is performed on a per-tablet basis. The first step is that the restarted node recomputes its local view of the consistent hash ring using the same deterministic hashing logic as the coordinator (keep a list of all known node IDs). This determines which tablets the node is responsible for. Then for each of those tablets, the recovering node contacts the coordinator to find the primary to sync. The primary responds with its current

checkpoint ID, epoch and last committed operation ID for that tablet. The recovering node compares this against its own local checkpoint. If the checkpoint versions match, then the recovering node loads it checkpoint and updates its missing tail of the log using its last operation ID to receive the WAL entries it missed. However, if the checkpoint ID differs, the recovering node receives the primary's full checkpoint snapshot and WAL. Once all tablets have been recovered into memory, the node sends heartbeats to the coordinator which allows for it to be marked alive.

2.2 Extra-Credit Features

- **HTML Styling**: We did this both through stylesheets in CSS and in inline styling.
- **Email**: We implemented an address book and folders for the webmail user interface. The address book consisted of a new key-value addition to our schema, which was the "UserDirectory" in table 5.2. Whenever DavidCloud has more than 1 user registered and a user clicks on the "To" text box to write an email, a drop-down menu containing the names of everyone in the directory appears. For folders, we implemented 3 separate tabs for incoming inbox, sent inbox, and trash. All inboxes support emails sent between users, SMTP and relay emails. The trash folder allows for a "dual delete" scheme, where you can only permanently delete from the trash folder.
- **Games**: We added three fun games to our project to allow people to have fun! These games are all mostly made by Forest Ho-Chen, and include *Confess*, *Love Sees Differences*, and *We Live We Love We Lie*. We even managed to get a special version of *We Live We Love We Lie* out (v2.0.0 preview) specifically for DavidCloud. DavidCloud supports third-party hosted WebGL applications through integrated iframe rendering.
- **File System**: We added support for uploads/downloads of files up to 500 MB through a chunked storage design built on top of our distributed KVS. Instead of storing an entire large file in a single value, files exceeding the inline storage threshold are automatically split into fixed-size chunks before being written to the backend. Each chunk is stored independently in the KVS while file metadata tracks the total file size, number of chunks, chunk size, and content type. During downloads or previews, the frontend transparently reconstructs the original file by sequentially retrieving and combining these chunks.

3. Team Member Contributions

Team Member	Component(s)
Forest Ho-Chen	Webmail service, SMTP integration, Frontend HTTP Server
Dan Kim	Recovery, Checkpointing, Replication, Admin
Stephen Kwak	Frontend HTTP server, Webmail service, Admin
David Zhan	Cloud storage, KVS, Load Balancer

4. Design Decisions and Challenges

4.1 Key Design Decisions

Decision 1: Replication strategy The coordinator can detect a failed node via regular heartbeat; if a node is down, the tablet assignments are recomputed and the failed node is excluded from all replicas until it is restarted. The first alive node in a replica is the primary by default. Secondaries can also be promoted to primary, ensuring the system runs as long as at least one node is alive in a replica set.

Decision 2: Frontend routing / load balancing For the frontend routing, we decided to have a main page, then separate sections for the mail, drive, accounts, games, and admin console. A lot of the pages load through a GET request, though the mail and drive also use POST requests to send data. For the load balancer, we made the design decision to use a round robin load balancer with the load balancer on port 8000 redirecting to the frontends on 8080, 8081, and 8082 in a round robin fashion.

4.2 Major Challenges

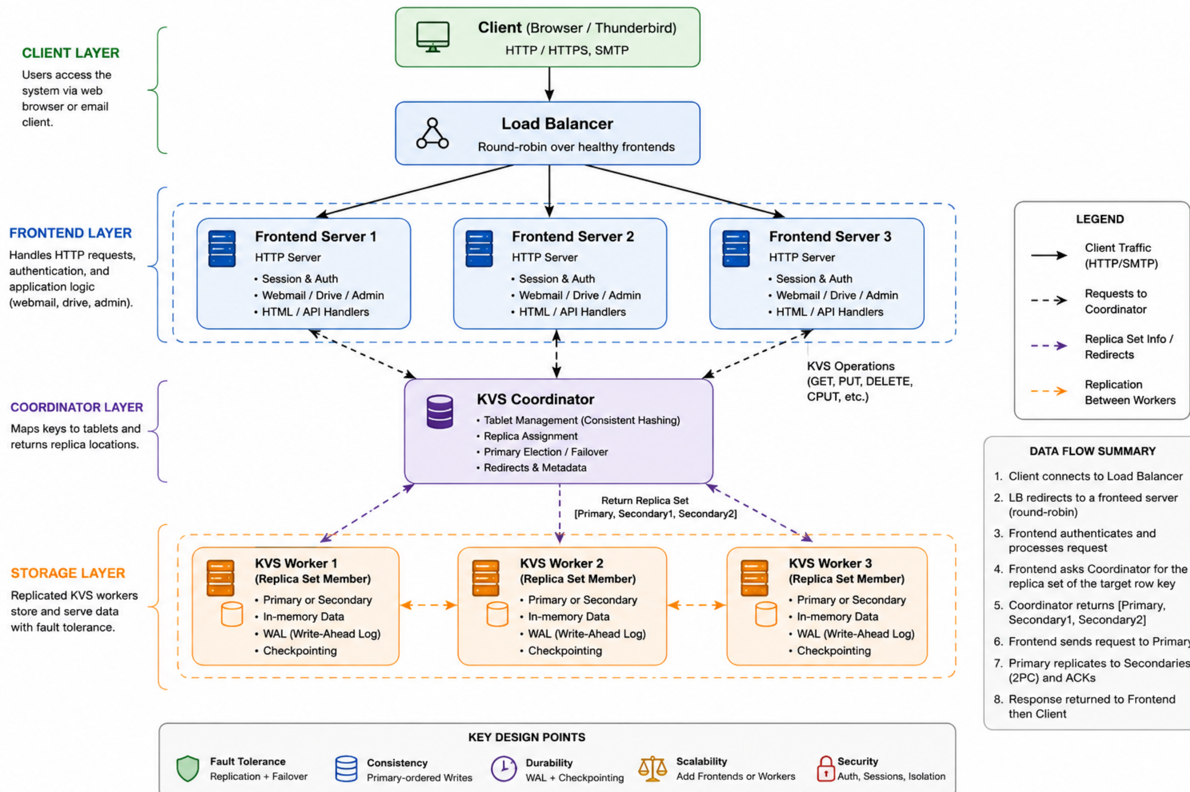
Challenge 1: Recovery One of the more nuanced challenges in the storage layer was designing a correct recovery protocol for a rejoining node. The naive approach, which scans the node's local `checkpoint/` and `log/` directories to discover which tablets it previously owned, is insufficient. When a node rejoins after a crash or restart, the consistent hash ring may have shifted during its absence, meaning the rejoining node may now be responsible for tablets for which it has no local state. The recovery protocol must therefore determine the node's intended tablet set proactively, not reactively.

Tablet Discovery via Local Hash Ring Reconstruction. At boot, before the recovering node registers with the coordinator, the recovery phase queries the coordinator for the current tablet assignments using `LIST_TABLETS`, alive node endpoints using `GET_NODE_STATUS`, and the replica count using `GET_REPLICA_COUNT`. It then rebuilds the consistent hash ring locally using the same virtual-node scheme as the coordinator, with the recovering node included as if it were already alive. This produces the full intended replica set for each tablet. The union of tablets where the recovering node appears is the authoritative set it must synchronize before serving traffic.

Challenge 2: Webmail + External mail Another challenge we had was getting the webmail working. We had a basic version of webmail working for the first demo, but getting the SMTP working was a bit difficult due to Thunderbird, Telnet, and the automated tests all acting slightly differently, so we had to make sure that our system could handle each of those. We also had to make sure that extra line breaks would be considered by our system. After we got that working, we struggled a bit with external emails, as the email servers would often reject our requests. We eventually got the email to the SEAS emails working through the DNS lookup and MX records, but the external personal emails would not work using this method since Gmail would reject us, so we had to add external email relays to handle personal emails. We also had some issues with deleting emails and getting the SMTP parts working with the KVS and accounts, but after some experimentation, we eventually got it working and fully integrated.

5. Appendix

5.1 Architecture Diagram



5.2 Schema

Table	Row Key	Column	Description
UserTable	user:<username>	auth:password_hash	Hashed password
		session:last_login	Timestamp of last login
		drive:root_folder_id	User's root Drive folder ID
SessionToUser	session:<session_id>	meta:username	Username for this session token
MailboxIndex	mailbox:<username>	meta:all	Serialized list of all message IDs
		meta:sent	Serialized list of sent message IDs
		meta:trash	Serialized list of trashed message IDs
		msg:<msgid>	Serialized header blob for fast rendering
MailRow	mail:<user>:<msgid>	meta:from	Sender address
		meta:to	Serialized list of recipients
		meta:subject	Subject line
		meta:timestamp	Send/receive timestamp
		meta:flags	Status flags (read, starred, etc.)
		body:text	Plain-text body
		body:raw	Raw message bytes
UserDirectory	directory:all	meta:users	Append-only list of all usernames
FolderIndex	folder:<user>:<folderid>	meta:name	Human-readable folder name
		meta:parent	Parent folder ID
		meta:children_files	Serialized list of child file IDs
		meta:children_folders	Serialized list of child folder IDs
		child:file:<fileid>	Filename of child file
		child:folder:<fid>	Name of child folder
FileMetadata	filemeta:<user>:<fileid>	meta:name	Filename
		meta:parent	Parent folder ID
		meta:size	Total file size in bytes
		meta:content_type	MIME type
		meta:created_at	Creation timestamp
		meta:updated_at	Last-modified timestamp
		meta:num_chunks	Total number of chunks
		meta:chunk_size	Chunk size in bytes
		meta:num_chunk_rows	Number of chunk row groups
FileChunk	filechunks:<user>:<fid>:<rg>	chunk:<chunkno>	Raw bytes for this chunk
CoordTablet	tablets:index	meta:all	Serialized list of all tablet IDs
		tablet:<tabletid>	Per-tablet metadata
AdminGlobalNode	nodes:index	meta:all	Serialized list of all node IDs
		node:<nodeid>	Extra per-node metadata
AdminNode	node:<nodeid>	meta:address	Host and port
		meta:role	frontend / backend / coordinator
		meta:status	Alive or dead
		meta:last_heartbeat	Timestamp of last heartbeat
		meta:tablets	Serialized list of hosted tablet IDs
UserMeta (fs)	__user__:<user_id>	root_inode	Inode ID of user's / directory
		next_inode	Counter for next inode ID allocation
DirInode (fs)	inode_<n>	__type__	Fixed value "dir"
		<child_name>	Child's inode string (e.g. inode_12)
FileInode (fs)	inode_<n>	__type__	Fixed value "file"
		__data__	Complete file contents

5.3 Protocol

Connection	Command	Response	Description
Frontend/Worker ↔ Coordinator	LOOKUP <rowkey>	OK <tabletid> <epoch> ...	Returns tablet ownership and replica locations for a given row key.
	LIST_TABLETS	OK <num_tablets> ...	Lists all tablets with epoch and replica metadata.
	GET_NODE_STATUS	OK <num_ids> ...	Returns cluster-wide node liveness and tablet assignment info.
	ADMIN_KILL <nodeid>	OK	Simulates node failure for testing.
	ADMIN_RESTART <nodeid>	OK	Restarts a previously failed node.
	HEARTBEAT <nodeid>	OK	Updates coordinator with node liveness timestamp.
	PREPARE_CHECKPOINT <checkpoint_id>	READY	Requests all replicas prepare for checkpoint creation.
	COMMIT_CHECKPOINT <checkpoint_id>	ACK	Finalizes checkpoint creation.
	DONE_CHECKPOINT <checkpoint_id>	n/a	Signals checkpoint completion.
	GET_REPLICA_COUNT	OK <replica_count>	Returns configured replication factor.
Frontend ↔ Worker	PUT <rowlen> <collen> <vallen>	OK / -ERR	Inserts or overwrites a value.
	GET <rowlen> <collen>	OK <vallen> / -ERR	Retrieves a value.
	CPUT <rowlen> <collen> <oldlen> <newlen>	OK / -ERR	Conditional put (compare-and-swap semantics).
	DELETE_ROW <rowlen>	OK / -ERR	Deletes all columns for a row.
	DELETE_COL <rowlen> <collen>	OK / -ERR	Deletes one column from a row.
Primary ↔ Secondary	PREPARE_PUT ...	ACCEPT / -ERR	Phase 1 of replicated PUT.
	PREPARE_DELETE_ROW ...	ACCEPT / -ERR	Phase 1 of replicated row deletion.
	PREPARE_DELETE_COL ...	ACCEPT / -ERR	Phase 1 of replicated column deletion.
	PREPARE_CPUT ...	ACCEPT / -ERR	Phase 1 of replicated conditional put.
	COMMIT_PUT ...	ACK / -ERR	Phase 2 commit for PUT.
	COMMIT_DELETE_ROW ...	ACK / -ERR	Phase 2 commit for row deletion.
	COMMIT_DELETE_COL ...	ACK / -ERR	Phase 2 commit for column deletion.
	COMMIT_CPUT ...	ACK / -ERR	Phase 2 commit for conditional put.
	ABORT_DELETE_ROW ...	ACK / -ERR	Aborts replicated row deletion.
	ABORT_DELETE_COL ...	ACK / -ERR	Aborts replicated column deletion.
	ABORT_CPUT ...	ACK / -ERR	Aborts replicated conditional put.
	SYNC_TABLET <tablet_id>	SYNC_INFO ... / NOT_FOUND	Returns checkpoint and log metadata for replica recovery.
	FETCH_RECOVERY_LOG <tablet_id> <after_op_id>	RECOVERY_LOG ...	Transfers WAL entries for incremental recovery.
	FETCH_RECOVERY_FULL <tablet_id>	RECOVERY_FULL ...	Transfers full checkpoint + WAL for replica rebuild.

Table 1: Distributed system communication protocol