Fractal Distributed File System

Final Report for the Distributed System Course 2024/2025

Matteo Zaccarini
matteo.zaccarini4@studio.unibo.it

Contents

1. Concept

Fractal is a **distributed file system** designed for large-scale, data-intensive workloads, drawing architectural inspiration from systems like the Hadoop Distributed File System (HDFS) and Google File System (GFS).
Fractal adopts a **CP** (Consistency and Partition Tolerance) model, deliberately prioritizing data integrity and strong consistency over availability by rejecting operations whenever the required **quorum** or replica counts cannot be safely satisfied. The entire ecosystem is operated via a standalone CLI that exposes commands for file management (create, read, update, and burn) alongside diagnostic utilities such as list and status to monitor cluster health.

A centralized NameNode maintains the cluster namespace and replica locations, persisting its state to a lightweight filesystem image, while raw data blocks are distributed across the physical disks of worker DataNodes. Rather than routing payload bytes through the master, Fractal utilizes a distributed pull model: clients query the NameNode solely for a chunk location blueprint, after which all heavy chunk streaming occurs concurrently and directly with the DataNodes to maximize bandwidth utilization.

Incoming files are split into fixed-size chunks and distributed using a rack-aware allocation strategy that guarantees multi-rack fault isolation. The system operates under an immutable storage paradigm where file updates generate a timestamped replica set before executing an atomic metadata swap, rather than modifying bytes in place. Continuous heartbeat exchanges and quorum consensus protocols allow Fractal to safely coordinate multi-client concurrency, actively detect dead workers, and transparently repair stale or missing replicas on routine failure.

2. Requirements Elicitation and Analysis

Glossary of Domain-Specific Terms:

Term	Definition
NameNode	The centralized master server that solely stores namespace metadata, tracking file-to-chunk and chunk-to-node mappings
DataNode	A worker server responsible for persisting the actual physical file binaries on its local hard drive
Chunk	A fixed-size fragment (maximum 64 MB) of a file.
FSImage	The persistent JSON snapshot (`file_system_image.json`) of the entire cluster's metadata state.
Rack	A logical grouping of DataNodes representing a distinct physical topology (e.g., Rack A, Rack B) to manage failure domains.
Quorum	The minimum threshold of healthy replica acknowledgments (W=2, R=2) required to validate an operation
Read Repair	An asynchronous background process that heals missing or mismatched size/version chunks detected during file retrieval.

Functional Requirements

ID	Requirement	Acceptance Criterion
FR1	The system must ingest files from the client and distribute them as discrete chunks across worker nodes.	A file exceeding 64 MB is successfully fragmented and stored across multiple remote DataNodes
FR2	The system must retrieve and seamlessly reassemble archived chunks into the original file.	A downloaded file opens correctly and matches the byte size of the original local file
FR3	The system must display a complete directory of all archived data	Executing `list` outputs an accurate namespace of filenames
FR4	The system must allow users to update files while treating existing data as immutable.	An `update` command generates a new timestamped file version and asynchronously deletes the old, orphaned chunks.

FR5	The system must permanently erase specific files upon request.	Executing `burn` removes the file from the FSImage and purges all associated physical chunks from the DataNodes.

Non-Functional Requirements

ID	Requirement	Acceptance Criterion
NFR1	The system must prioritize Strong Consistency over Availability (CP).	An upload request is explicitly rejected if the cluster lacks the necessary active DataNodes to fulfill the replication factor
NFR2	The system must seamlessly tolerate the failure of a single DataNode without interrupting data retrieval. . .	A file download succeeds even if one DataNode containing a replica is forcefully taken offline, as the remaining replicas satisfy the R=2 quorum.
NFR3	The system must seamlessly tolerate the failure of a single DataNode during file ingestion without aborting the write operation.	A file upload completes successfully and is committed to the NameNode even if one of the three allocated DataNodes crashes during the streaming phase, provided the W=2 quorum is reached.
NFR4	The system must proactively enforce data integrity by healing missing or corrupted replicas during reads.	If a DataNode reports a chunk as missing or returns a mismatched byte size compared to the quorum, the client successfully downloads the file from healthy nodes and triggers a background upload to heal the stale node.

Implementation Requirements

ID	Requirement	Acceptance Criterion
IR1	**Go & gRPC:** selected to generate statically linked binaries with no external dependencies, and to leverage high-performance concurrent data streaming.	The system compiles to lightweight executables that communicate exclusively via Protocol Buffers.
IR2	**Docker Containerization:** Mandated to simulate a multi-rack distributed topology (1 NameNode, 4 DataNodes) locally without requiring extensive physical hardware. . .	The entire cluster infrastructure boots deterministically via a single `docker-compose.yml` file.

2.1 Relevant Distributed System Features

- **Transparency**: the system fully masks distribution complexities from the user. Through the CLI, users interact with the cluster exactly as they would with a local file system (e.g., `fractal create file.txt`). The underlying mechanics of 64 MB chunking, W=2/R=2 quorum consensus, and rack-aware placement remain completely invisible to the end user.

- **Fault Tolerance & Dependability**: by aligning with the CP side of the CAP theorem, the system guarantees Strong Consistency. It utilizes rack-aware allocation (distributing replicas across Rack A and Rack B), heartbeat monitoring (15-second timeouts), and a proactive read-repair mechanism to seamlessly heal corrupted or missing data without user intervention. Furthermore, the client implements robust consensus logic, grouping metadata responses by chunk size to neutralize "fast-but-wrong" corrupted nodes, and automatic stream failover, ensuring downloads succeed even if a DataNode crashes mid-transfer.

- **Scalability & Evolvability**: the architecture natively supports horizontal scaling. Because the NameNode dynamically allocates chunks based on incoming heartbeats (`allocator.go`), adding a new DataNode does not require code changes. An administrator only needs to spin up a new Docker container pointing to the NameNode, and the cluster will automatically begin routing new chunks to it.

- **Resource Sharing & Concurrency**: Fractal supports multiple concurrent clients. To prevent race conditions when multiple users upload or delete files simultaneously, the NameNode employs `sync.RWMutex` locks over the `ClusterState` namespace. Furthermore, data ingestion is highly concurrent, with clients broadcasting chunks to multiple DataNodes in parallel.

- **Openness & Interoperability**: the system is inherently open due to its reliance on gRPC and Protocol Buffers (`fractal.proto`). Because the API contracts are language-agnostic, external systems or frontends written in Python, Java, or Rust could trivially interoperate with the existing Go cluster by simply compiling the `.proto` file.

- **Performance & Bandwidth**: performance is optimized by aggressively protecting the NameNode from both network and disk I/O bottlenecks. The NameNode strictly separates ephemeral routing data (kept entirely in RAM for nanosecond lookups) from persistent namespace data (flushed to disk), preventing heartbeat floods from saturating the hard drive. Over the network, a strict pull model ensures the NameNode only transfers lightweight JSON blueprints. The actual heavy payload streaming (64

MB chunks sent in 64 KB increments) bypasses the NameNode entirely, creating a direct data plane between the client and the allocated DataNodes.

- **Security & Trust**: this feature is explicitly out of scope for the current prototype. The system operates within a trusted internal network, meaning there is no implemented authentication (e.g., JWT), authorization (e.g., RBAC), or cryptographic encryption for data at rest or in transit.

- **Economy**: the system is highly economical, utilizing an entirely open-source stack. Go's statically linked binaries are deployed on minimal Alpine Linux Docker images, drastically reducing memory footprint and allowing the cluster to run efficiently on commodity hardware.

3. Design

3.1 Architecture

The system employs a hybrid Master-Slave and Client-Server architectural style to satisfy the requirement for Strong Consistency (NFR1) and optimize bandwidth utilization.

- Master-Slave (Control Plane): a single NameNode acts as the authoritative master, orchestrating metadata and tracking the health of four slave DataNodes. This centralized topology was chosen over a decentralized approach to guarantee CP (Consistency and Partition tolerance) under the CAP theorem. A single source of truth prevents split-brain scenarios and drastically simplifies the mathematical Rack-Aware allocation logic (FR1) by maintaining a global view of cluster health.
- Client-Server (Data Plane): the architecture strictly decouples data from metadata. The CLI client queries the NameNode for routing blueprints, then establishes direct gRPC streaming connections with the DataNodes to transfer heavy payloads, ensuring the master never bottlenecks.

3.2 Infrastructure

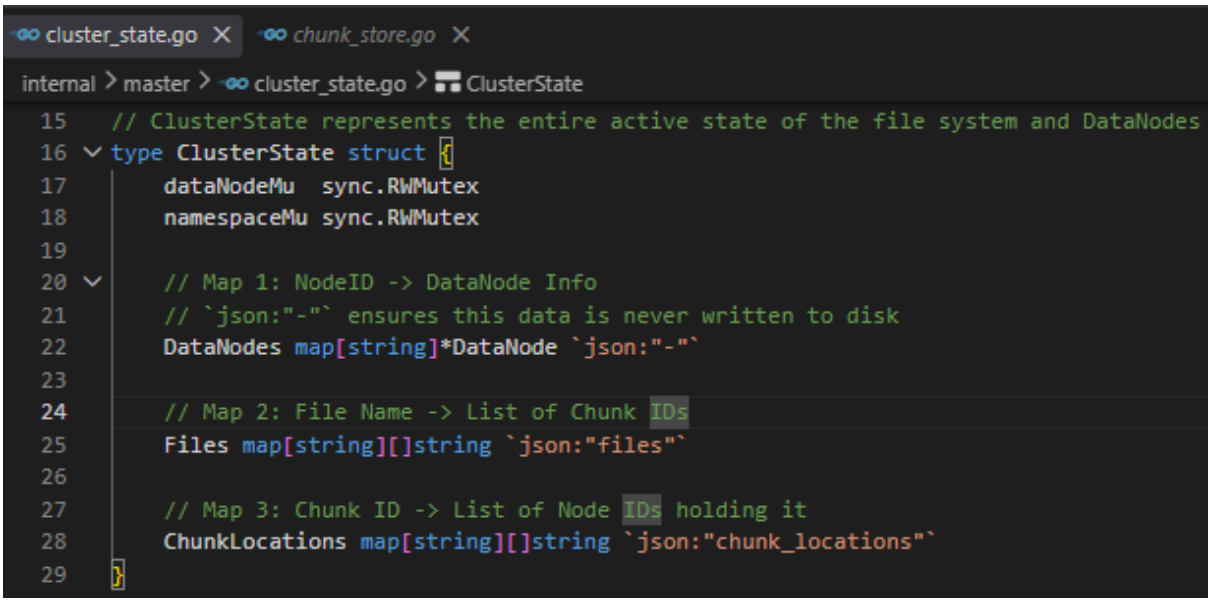
To fulfill the implementation constraint (IR2) for simulated distributed environments, the infrastructure is entirely containerized using Docker Compose.
The ecosystem relies on three component types: one NameNode, four DataNodes, and N CLI Clients. The NameNode and DataNodes are deployed on an isolated virtual bridge network (`fractal-network`). The DataNodes are logically grouped into two physical failure domains: Rack A (datanode-1, datanode-2) and Rack B (datanode-3, datanode-4). This topological distribution directly satisfies the hardware fault tolerance requirement (NFR2). The CLI Client operates from the host machine, outside the isolated network. Data is persisted using Docker Named Volumes mapped to each container.
Components locate each other utilizing Docker's internal DNS. DataNodes push heartbeats to the master using the internal hostname `namenode:9000`. For client interaction, the NameNode and DataNodes expose local ports (`9000` and `8001-8004`, respectively) mapped to the host machine.

3.3 Modelling

Class Diagram 1 *(link here)*

Domain Entities & Infrastructure Mapping

- **Namespace / Cluster State (Master Entity):** the centralized, authoritative ledger of the distributed system. It aggregates both persistent routing tables (Files and ChunkLocations) and ephemeral network health. This entity maps exclusively to the NameNode, acting as its in-memory database.



- **Worker Profile (Node Entity):** the logical representation of a storage workhorse. Modeled via the `DataNode` struct, it tracks a worker's network identity (`NodeID`, `Address`), physical location (`RackID`), and health status (`LastHeartbeat`). These profiles map exclusively to the NameNode's ephemeral RAM, populated dynamically via heartbeats.

- **File (Logical Data Entity):** the core asset the user interacts with (e.g., `video.mp4`). It does not exist as a single physical object. It maps strictly to the NameNode's persistent state as a logical ordered list of Chunk IDs.
- **Chunk (Physical/Logical Data Entity):** the fundamental, immutable 64 MB building block of the system. This entity spans both components: the *logical* Chunk topology (which IPs hold the chunk) maps to the NameNode's RAM and JSON disk, while the *physical* Chunk (the raw `.dat` binary) maps to the `ChunkStore` wrapper on the DataNodes' local file systems.

- **Rack (Topology Entity):** The physical failure domain used to guarantee fault tolerance. It maps logically into the NameNode's allocation algorithm, ensuring the orchestrator distributes Chunk replicas across disparate physical environments.

Domain Events

Domain events are categorized into cluster health, data lifecycle, and fault tolerance.

- **Cluster Health Events**
  - ☐ **Heartbeat Received:** a DataNode pings the NameNode (every 3 seconds). The NameNode reacts by updating the worker's timestamp in the ephemeral RAM state, confirming it is alive and eligible for data allocation.
  - ☐ **Node Evicted (Timeout):** triggered by a background ticker when a DataNode fails to send a heartbeat for 15 seconds. The NameNode reacts by excluding the dead node from future `AllocateDataNodes` mathematical calculations.
- **Data Lifecycle Events**
  - ☐ **Blueprint Allocated:** triggered when a client initiates an upload. The NameNode mathematically selects healthy DataNodes, locking in the topology and returning the routing blueprint to the client.
  - ☐ **File Committed:** fired when a client successfully finishes streaming all chunks to the DataNodes. The NameNode reacts by requesting an exclusive write lock (`Lock`) and flushing the new namespace mapping to the `fsimage.json` disk file.
  - ☐ **File Swapped:** occurs during the `update` command. The NameNode reacts by atomically swapping a temporary filename pointer with the target filename in the namespace, ensuring the update is seamless before the client asynchronously deletes the orphaned chunks.
  - ☐ **File Burned:** triggered when a user permanently deletes a file. The NameNode erases the entity from its persistent ledger, and the client reacts by iterating through the blueprint to send `DeleteChunk` requests directly to the respective DataNodes.
- **Fault-Tolerance & Consensus Events**
  - ☐ **Quorum Reached:** triggered inside the client during parallel network I/O. Once the W=2 or R=2 threshold of successful DataNode acknowledgments is met, the client reacts by proceeding with the data stream, ignoring any remaining slower nodes.
  - ☐ **Quorum Rejected:** fired when hardware failures exceed the system's tolerance. The NameNode or Client reacts by actively aborting the operation, sacrificing Availability to uphold Strong Consistency (CP).

- ☐ **Stale Replica Detected:** occurs during the download scatter-gather phase. The client reacts by flagging any DataNode that reports a missing chunk (`exists == false`) or a mismatched byte size compared to the consensus group.
- ☐ **Read Repair Triggered:** fired immediately as a consequence of a stale replica. The client reacts by spawning background goroutines (`sync.WaitGroup`) to stream the correct chunk data back to the corrupted node, healing the cluster while seamlessly serving the file to the user.

Message Types

The communication architecture relies entirely on Protocol Buffers serialized over gRPC to enforce strict, language-agnostic API contracts. Messages are strictly categorized by their functional intent across the Control Plane (metadata) and Data Plane (payloads).

- **Commands (State Mutators):** these are synchronous, unary requests instructing the system to alter its persistent or physical state.
  - `CreateFileRequest` / `CommitFileRequest`: instructs the NameNode to calculate a chunk allocation blueprint and subsequently finalize the file in the `fsimage.json` ledger.
  - `SwapFileNameRequest`: used specifically by the update command to atomically overwrite an existing file pointer with a newly uploaded version.
  - `DeleteFileRequest` / `DeleteChunkRequest`: directed at the NameNode and DataNodes respectively, commanding the permanent erasure of logical namespace entries and physical `.dat` binaries.
- **Queries (State Retrievers):** these are lightweight, read-only requests used to fetch metadata without modifying the cluster.
  - `GetFileLocations`: the client queries the NameNode for a specific file's routing blueprint (mapping chunk IDs to worker IPs).
  - `ListFilesRequest`: the client queries the NameNode for a directory of all archived data to display in the CLI.
  - `CheckChunkRequest`: a scatter-gather query where the client asks multiple DataNodes if they hold a specific chunk and what its exact byte size is, forming the mathematical basis for the W=2/R=2 quorum.
- **Streams (Payload Data):** unlike unary requests, these messages handle continuous I/O for heavy physical data, actively bypassing the NameNode to preserve bandwidth.
  - `ChunkData`: the core byte-carrying message. During uploads, the client acts as a Client-Streaming endpoint (`StoreChunk`), pushing 64 KB arrays sequentially until a 64 MB chunk is completed. During downloads, the DataNode acts as a Server-Streaming endpoint (`RetrieveChunk`), pushing the bytes back to the client.

- **Events (System Signals):**
    - `HeartbeatMsg`: an asynchronous, unidirectional event fired by DataNodes every 3 seconds. It passively informs the NameNode of the worker's network identity, rack location, and active health status.

System State Information

The state of the system comprehends both the logical namespace (metadata) and the physical topology (cluster health). To optimize for nanosecond lookup speeds, this information is bifurcated into three distinct categories:

1. Persistent State (The Namespace Ledger)

This information is the core "database" of the system, flushed to `file_system_image.json` on the NameNode to survive reboots:

- **Users' File Data:** the logical mapping of a user-defined file (e.g., `document.pdf`) to an ordered list of unique physical Chunk IDs.
- **Chunk Topology Data:** the network mapping that tracks exactly which DataNode IPs currently hold the 3 replicas for every specific Chunk ID in existence.

2. Ephemeral State (The Cluster Health)

This information is highly volatile and is kept **strictly in the NameNode's RAM**, populated entirely by incoming heartbeats:

- **Node Liveness Data:** The exact timestamp of the last received heartbeat for every worker, used to calculate 15-second timeout evictions.
- **Hardware Topology Data:** The physical rack assignments (Rack A vs. Rack B) and active IP addresses of the DataNodes, required by the allocator to mathematically guarantee rack-aware fault tolerance.

3. Physical State (The Payloads)

- **Users' Binary Data:** The actual 64 MB immutable raw `.dat` binary blocks stored on the physical hard drives of the individual DataNodes."

3.4 Interaction

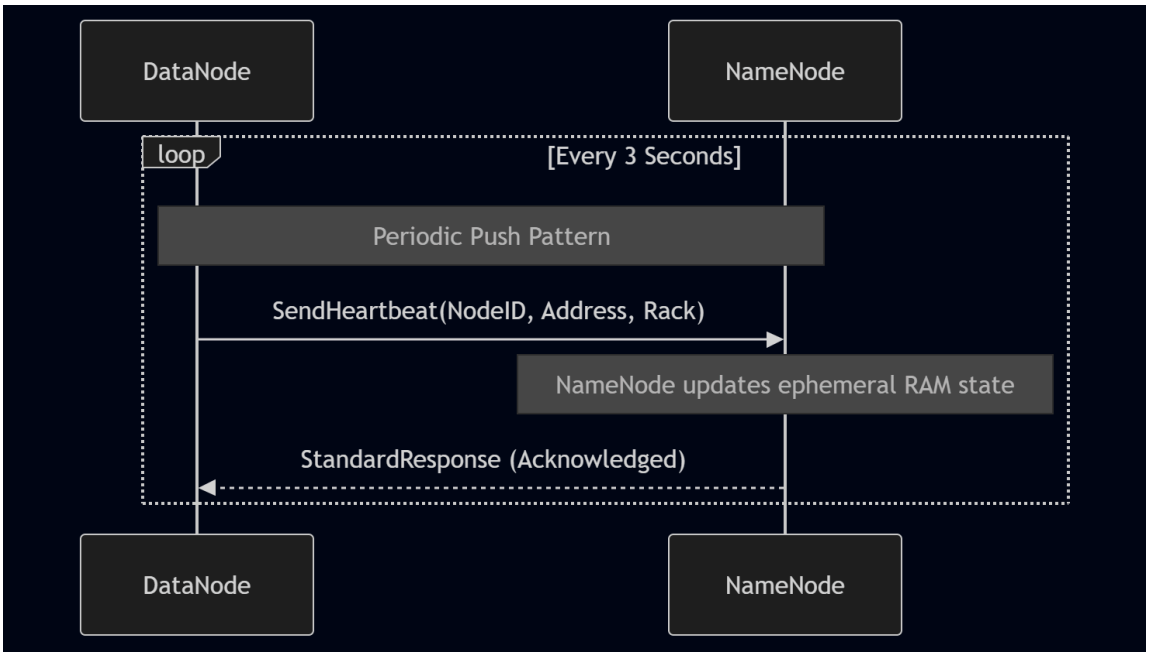
The system's communication relies entirely on gRPC over TCP, utilizing Protocol Buffers to serialize data. Communication is strictly decoupled between the Control Plane (metadata) and the Data Plane (physical files) to optimize bandwidth.

Component Communication (How, When, What)

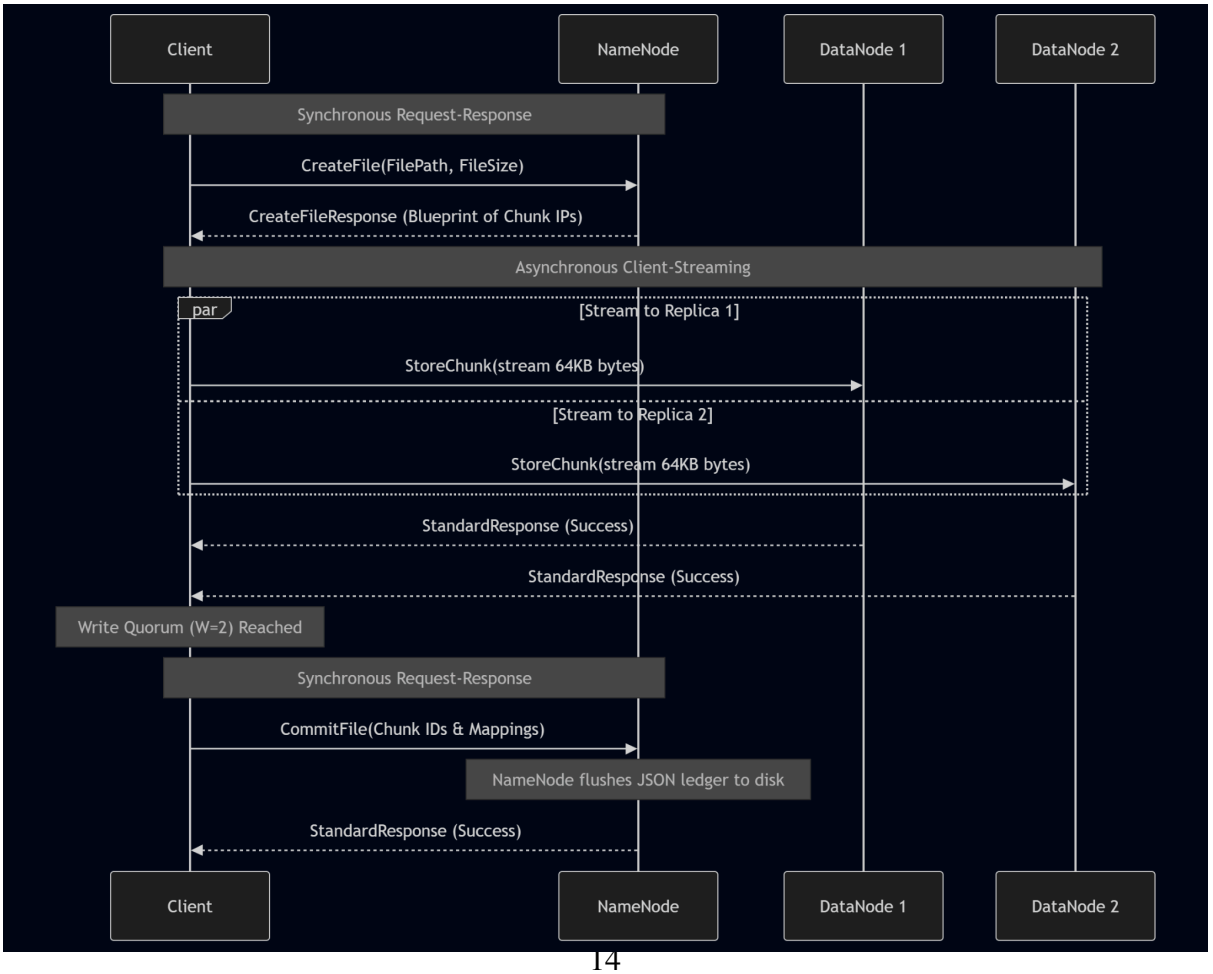
- **DataNode to NameNode:** communication occurs automatically and continuously in the background. Every 3 seconds, the DataNode triggers a network call to the NameNode. It communicates lightweight status data, explicitly sending its Node ID, network address, and rack location to prove it is alive and eligible for storage allocation.
- **Client to NameNode:** communication occurs on-demand whenever a user executes a CLI command. The client and NameNode communicate exclusively regarding metadata. For example, during a `read` operation, the client asks for file locations, and the NameNode returns a JSON-equivalent blueprint mapping chunk IDs to DataNode IPs. Physical file bytes never traverse this connection.
- **Client to DataNode:** communication occurs strictly during the physical I/O phases of `create`, `read`, or `update` commands. The client communicates directly with the specific DataNodes listed in the NameNode's blueprint. The payload consists of heavy, raw binary data (64 MB chunks) transmitted in 64 KB increments to prevent memory overflow.

Interaction Patterns

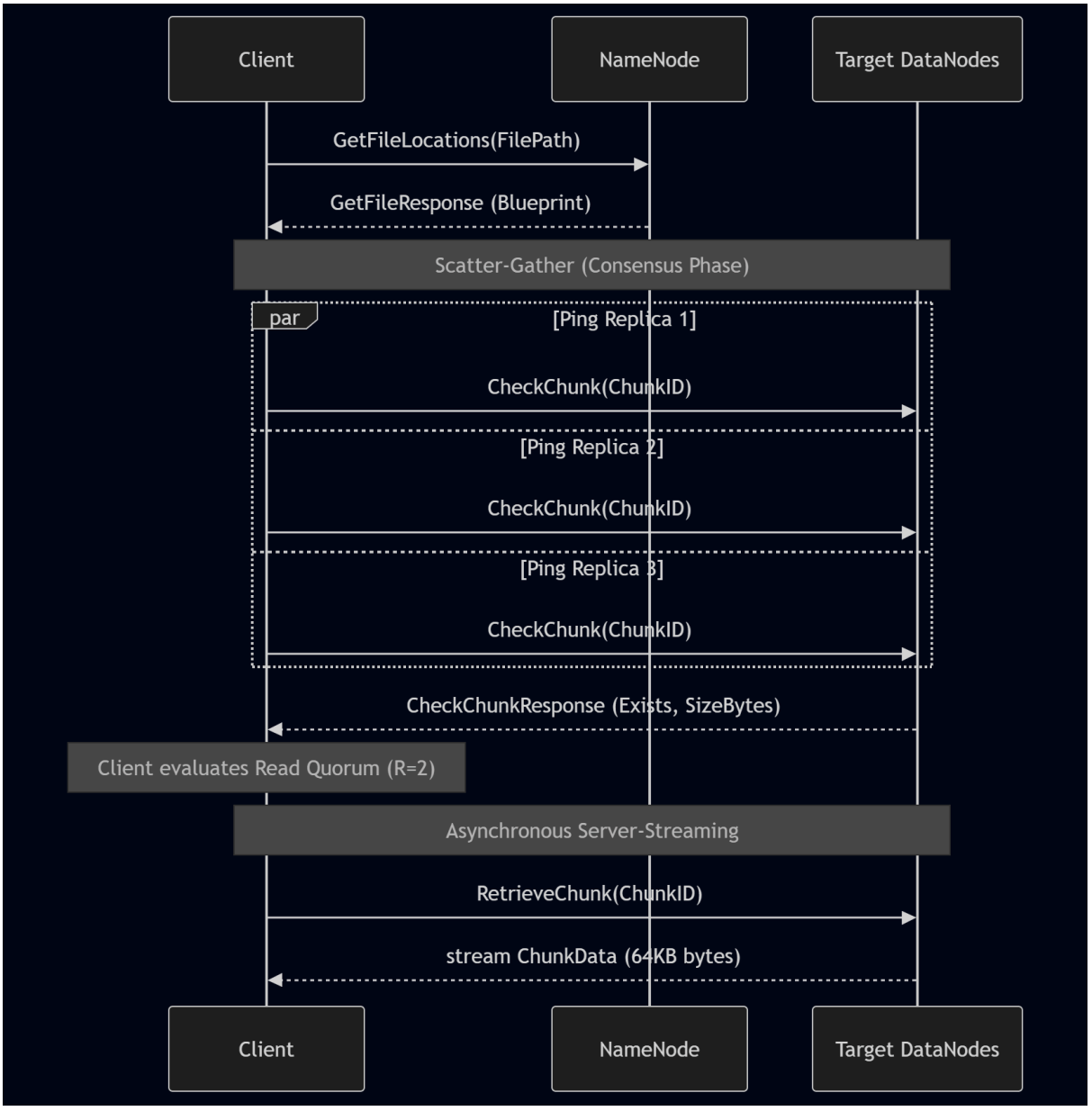
- **Periodic Push:** enacted by the DataNodes for cluster health monitoring. The DataNodes act as active publishers, pushing state information (heartbeats) to the passive NameNode without waiting for the NameNode to request it.
- **Synchronous Request-Response:** enacted between the Client and the NameNode. This follows a standard unary RPC pattern where the client blocks and waits for a definitive response before proceeding. Examples include querying the file directory (`ListFiles`) or asking for chunk locations (`GetFileLocations`).
- **Asynchronous Streaming:** enacted between the Client and DataNodes for heavy file transfers. To avoid loading entire 64 MB chunks into RAM at once, the system uses gRPC streams. Uploads utilize a **Client-Streaming** pattern, where the client continuously pushes bytes until EOF, while downloads utilize a **Server-Streaming** pattern, where the DataNode continuously pushes bytes back to the client.
- **Scatter-Gather (Consensus & Ping):** enacted by the Client to mathematically verify the W=2/R=2 quorum. Before downloading, the client *scatters* concurrent unary requests (`CheckChunk`) to multiple DataNodes simultaneously using Go routines, and then *gathers* the responses via Go channels to determine which nodes are healthy and which contain corrupted or missing data.



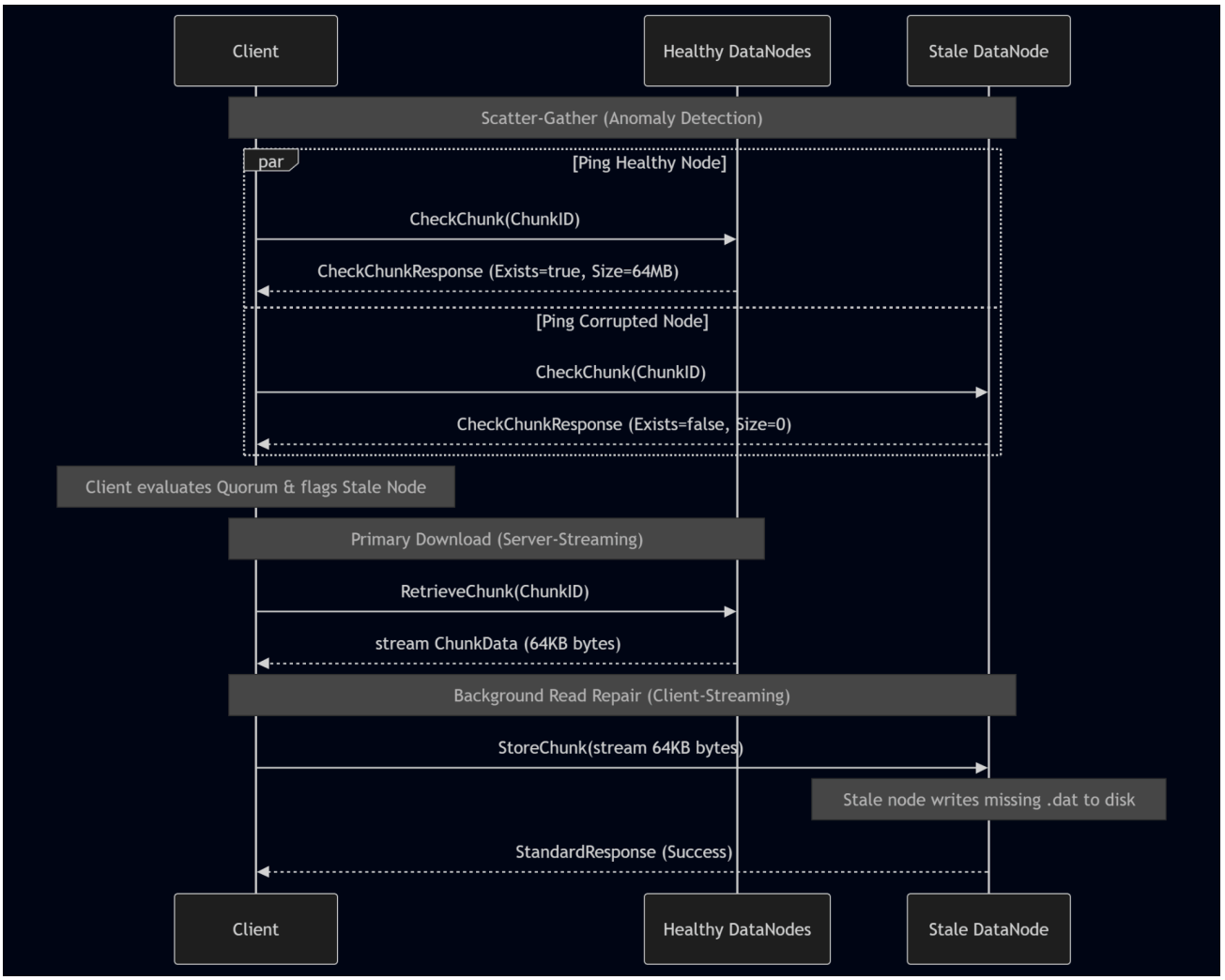
Sequence Diagram 1: illustrates the continuous, unidirectional background communication where DataNodes act as active publishers to prove their liveness to the NameNode (Periodic Push).



Sequence Diagram 2: illustrates the `create` command lifecycle. It visually separates the synchronous metadata negotiation (Synchronous Request-Response) from the heavy asynchronous chunk streaming (Asynchronous Streaming).



Sequence Diagram 3: illustrates the `read` command lifecycle, highlighting how the client orchestrates consensus (W=2/R=2)(Scatter-Gather) before triggering the physical download (Asynchronous Streaming)



Sequence Diagram 4: illustrates Read-Repair interaction

3.5 Behaviour

Component Behaviors & Statefulness

- **The Client (Stateless Orchestrator):** Operates as an ephemeral CLI application. Upon execution, it queries the NameNode for metadata blueprints, directly streams binary chunks to/from DataNodes, and mathematically evaluates Read/Write quorums. It retains no persistent state between terminal commands, cleanly tearing down network connections upon completion.
- **The NameNode (Stateful Control Plane):** Functions as the centralized brain of the cluster. It passively listens for gRPC requests, responding to client namespace queries

and reacting to incoming worker telemetry. It must maintain the entire cluster topology, active worker list, and file mapping blueprints continuously in RAM.

- **The DataNode (Stateful Data Plane):** Operates as the physical storage worker. It runs an autonomous background goroutine to transmit 3-second heartbeats to the Master. Concurrently, it passively listens for client network streams to read, write, or verify raw `.dat` chunks on its local host filesystem. DataNodes only understand isolated chunk IDs and are entirely unaware of the global file namespace.

System State Updates (Who, When, How)

- **Global Namespace Updates:** The NameNode strictly controls the logical file state. When a client issues a structural mutation (`CreateFile`, `CommitFile`, `SwapFileName`, `DeleteFile`), the NameNode acquires an exclusive `sync.RWMutex` write lock, mutates the in-memory map, and synchronously flushes the new mapping architecture to `file_system_image.json` on disk to guarantee persistence.

State Diagram 1 *(link here)*: the structural mutations of a file within `ClusterState` and `file_system_image.json`

- **Cluster Topology Updates:** The NameNode dynamically updates the network routing table exclusively in RAM. This occurs constantly as DataNodes push their state via `HeartbeatMsg` payloads (updating liveliness and active rack metrics), or when the NameNode's allocator explicitly mutates the state by evicting a worker following a 15-second network timeout.

State Diagram 2 *(link here)*: DataNode Liveness State models ephemeral worker health tracked in the NameNode's RAM.

- **Physical Data Updates:** The physical storage state on the hard drives is updated by the DataNodes, but the action is dictated entirely by the Client. When a user uploads a file or when the background `triggerReadRepair` pipeline fires, the Client pushes stream buffers directly to the target DataNodes, forcing them to mutate their local storage state.

State Diagram 3 *(link here):* Physical Chunk Storage State models the binary `.dat` file lifecycle on a DataNode's physical hard drive inside `ChunkStore`.

3.6 Data and Consistency Issues

Data Storage Requirements

- **Logical Metadata (Namespace Mapping):** maps human-readable file paths to ordered lists of physical chunk IDs (e.g., `"thesis.pdf" -> ["chunk-0", "chunk-1"]`). Stored on the NameNode to reconstruct files during downloads and update operations.
- **Physical Location Metadata (Block Blueprint):** maps unique chunk IDs to the network locations of DataNodes hosting replicas (e.g., `"chunk-0" -> ["datanode-1:8001", "datanode-3:8003"]`). Stored on the NameNode to route client I/O streams and verify replication topology.
- **Worker Node State:** tracks active DataNode IDs, network addresses, rack affiliations (`Rack-A`, `Rack-B`), and heartbeat timestamps. Maintained in-memory on the NameNode to detect worker failures and enforce rack-aware allocation.
- **Physical Binary Chunks:** The actual raw byte fragments of user files (split into 64 MB blocks). Stored across the local file systems of the DataNodes inside dedicated storage paths (`/app/data/datanode-X`).

Persistent Storage Strategy & Data Model

- **In-Memory Key-Value with Document Persistence (NameNode):**
  - *Storage Model:* metadata is structured as high-speed, in-memory Go key-value maps (`map[string][]string`).
  - *Persistence Mechanism:* flushed as a structured Document format (`file_system_image.json`) to disk on state mutations (`CommitFile`, `DeleteFile`, `SwapFileName`).
  - *Rationale:* Minimizes read/write latency to O(1) memory operations, removing database driver overhead. On crash recovery, the NameNode reloads this document sequentially to restore cluster state.
- **Direct Flat-File Storage (DataNodes):**
  - *Storage Model:* Stored as raw `.dat` binary files on the host file system via the `ChunkStore` abstraction.
  - *Rationale:* Eliminates database serialization overhead, enabling zero-copy streaming of 64 KB buffers directly from disk descriptors over gRPC streams.

Database Queries & Concurrency Model

- **Querying Components & Operations:**
  - **CLI Client:** Issues read queries (`GetFileLocations`, `ListFiles`, `GetClusterStatus`) during file downloads, namespace inspections, and health checks. Issues write mutations (`CreateFile`, `CommitFile`, `SwapFileName`, `DeleteFile`) during ingestion, overwrites, and purges.
  - **DataNodes:** Issue heartbeat mutations (`SendHeartbeat`) to the NameNode every 3 seconds to update liveliness and rack metadata.
- **Concurrency Architecture:**
  - The NameNode protects its in-memory state using dual `sync.RWMutex` primitives (`dataNodeMu` and `namespaceMu`).
  - *Concurrent Reads:* Multiple download requests and status checks acquire non-blocking shared read locks (`RLock`), allowing simultaneous metadata retrieval without throughput bottlenecks.
  - *Concurrent Writes:* Heartbeat ingestion and state mutations acquire exclusive write locks (`Lock`), serializing namespace mutations to prevent race conditions during updates and file commits.

Shared Data Across Components

- **Chunk Blueprints (NameNode → Client):** The client must receive the physical mapping of chunks to worker IPs to perform decentralized, peer-to-peer binary transfers directly with DataNodes.
- **Heartbeat Signals (DataNodes → NameNode):** Workers share their IP, assigned rack, and active status so the NameNode maintains cluster awareness.
- **Chunk Metadata & Verification Pings (DataNodes → Client):** Workers share existence status and exact byte sizes (`CheckChunk`) with the client to mathematically evaluate Read Quorum (R=2) and trigger Read Repair if size mismatches are detected

3.7 Fault-Tolerance

Data Replication and Topology

- **Replication Strategy:** The system inherently replicates every 64 MB chunk across 3 distinct DataNodes to safeguard against localized hardware failures and data corruption.

- **Rack-Aware Distribution:** To survive large-scale outages, the NameNode allocates these three replicas strictly across multiple physical failure domains. The first replica is assigned to a node in one rack (e.g., `Rack-A`), while the second and third are routed to a separate rack (e.g., `Rack-B`). This guarantees continuous availability even if an entire network switch loses power.

Heartbeats, Timeouts, and Network Resilience

- **Worker Heartbeats:** Every DataNode runs an autonomous goroutine (`StartHeartbeat`) that transmits its active status and rack identity to the NameNode every 3 seconds via gRPC.
- **Dead Node Eviction:** The NameNode validates these pings dynamically. If a DataNode's `LastHeartbeat` ages past 15 seconds, the `AllocateDataNodes` logic flags the worker as dead and prevents any new chunks from being assigned to its IP address.
- **Client Network Timeouts:** Before downloading files, the client probes worker health using a strict 2-second `context.WithTimeout` barrier (`DataNodePingTimeout`). If a DataNode hangs or drops packets, the client severs the connection immediately rather than freezing the terminal application.

Error Handling and State Recovery

- **Write Failover:** During uploads, the client streams data to all targeted DataNodes concurrently. If a worker crashes mid-transfer, the client verifies if the Write Quorum (W=2) was reached by the surviving nodes. If at least two replicas are written successfully, the client completely masks the error and commits the file; otherwise, it aborts the operation to prioritize system consistency.
- **Read Failover:** If a network interruption occurs while streaming a chunk to the user, the client catches the EOF or transmission error and seamlessly retries the download using the next healthy replica in the routing list.
- **Automated Read Repair (Self-Healing):** The client cross-references physical byte sizes across replicas prior to downloading. If a DataNode reports a missing chunk or a size mismatch, the client retrieves the correct data for the user and subsequently triggers an asynchronous background thread (`triggerReadRepair`). This thread re-uploads the verified bytes to the broken node, healing the cluster silently.
- **Master Node Recovery:** The NameNode avoids state loss by synchronously writing all namespace mutations to a local `file_system_image.json` file. Following a

crash, it deserializes this JSON document back into RAM to instantly restore the mapping blueprint

3.8 Availability

Fractal guarantees high availability and performant I/O routing through in-memory state management and rack awareness allocation, while strictly adhering to the CP side of the CAP theorem during severe network partitions.

Caching Mechanisms

- The Master node does not query the physical hard drive for every client request. Instead, it loads the entire `ClusterState` (file namespace, chunk blueprints, and active worker statuses) directly into RAM using native Go maps. This in-memory caching enables O(1) lookup complexity, allowing the NameNode to serve `GetFileLocations` blueprint requests instantly without disk latency bottlenecks. The disk is strictly reserved for synchronous persistence (`SaveToDisk`) during state mutations.

Load Balancing Strategies

- **Randomized Allocation Shuffling:** When allocating chunks, the NameNode groups healthy workers by their `RackID` and actively randomizes the arrays using Go's `rand.Shuffle`.
- **Hotspot Prevention:** By shuffling the node list before picking the target IPs, the system naturally balances new chunk placements across all available workers in a rack, preventing sequential writes from overwhelming a single DataNode.

Network Partitioning & CAP Theorem Behavior

- **Strict CP Compliance:** Fractal explicitly prioritizes Consistency over Availability (CP) when network partitions occur.
- **Upload Rejection:** If a partition separates the NameNode from its workers (causing the healthy node count to drop below the required replication factor (3)), the allocator aggressively aborts operations, explicitly logging that it is sacrificing availability for consistency.
- **Quorum Enforcement:** During client-DataNodes transfers, if a network partition prevents the client from contacting at least 2 identical replicas (W=2 or R=2), the

client aborts the network stream immediately to guarantee the user is protected from stale or divergent data.

3.9 Security

Authentication

Fractal operates without user or node authentication. All gRPC connections, whether Client-to-NameNode or Client-to-DataNode, utilize `insecure.NewCredentials()`.

Authorization & Access Control

There is no Role-Based Access Control (RBAC) or localized Access Control List (ACL) enforcement. The system does not recognize distinct user identities or POSIX-style file ownership. The cluster maintains a flat, globally accessible namespace. Any client capable of resolving the NameNode's IP address possesses implicit administrative privileges, meaning any user can execute universal mutations or download any existing file

Cryptographic Schemas & Data Protection

- **Data-in-Transit:** gRPC communication streams are transmitted in plaintext. Transport Layer Security (TLS) and mutual TLS (mTLS) are intentionally omitted.
- **Data-at-Rest (Storage):** Neither metadata nor physical file chunks are encrypted on the storage drives. The NameNode persists its state as a plaintext `file_system_image.json` document, and DataNodes write raw, unencrypted `.dat` binary files directly to the host filesystem.
- **Data Integrity Constraints:** Fractal bypasses cryptographic hashing (such as SHA-256 or MD5 checksums) for file verification. Instead, it mathematically evaluates Read Quorum (R=2) by cross-referencing the exact byte sizes of physical replicas during the `CheckChunk` phase. If a size mismatch is detected, the system immediately assumes corruption and initiates the background Read Repair pipeline without incurring the CPU overhead of cryptographic token generation.

4. Implementation

Network Protocol

The system exclusively utilizes **gRPC** operating **over TCP** for all network communications. File system operations demand guaranteed delivery, strict packet ordering, and data integrity. TCP provides these guarantees inherently, whereas UDP would require implementing complex, custom packet-loss recovery mechanisms to ensure a 64 MB chunk is not corrupted during transit.

Standard REST over HTTP/1.1 is inadequate for heavy binary transfers because it is fundamentally stateless and request-response driven. gRPC, built on HTTP/2, natively supports multiplexing and long-lived streams. This is critical for Fractal's Data Plane: it allows the client to open a `ClientStreaming` connection to push 64 KB byte arrays sequentially (`StoreChunk`), and a `ServerStreaming` connection to pull data back (`RetrieveChunk`) without the overhead of opening and closing HTTP connections for every block.

In-Transit Data Representation

All data exchanged between the NameNode, DataNodes, and the CLI client is strictly serialized using **Protocol Buffers** (`fractal.proto`).

While JSON is used on the NameNode for writing the `fsimage.json` to disk (because it deserializes easily into Go maps), it is highly inefficient for network transit. If the system used JSON or XML to transmit the file chunks, the raw binary data would have to be Base64 encoded, which bloats the payload size by approximately 33% and wastes CPU cycles on encoding/decoding.

Protobuf natively supports a `bytes` data type. In the `ChunkData` message, the file fragment is transported as raw binary (`bytes data = 2;`), ensuring maximum network bandwidth efficiency during the 64 MB transfers.

Protobuf enforces strong typing. By defining explicit messages like `HeartbeatMsg` and `GetFileRequest`, the compiler generates native Go interfaces (`fractal.pb.go`, `fractal_grpc.pb.go`) that guarantee the NameNode, DataNodes, and Client perfectly understand the data structures being transmitted, eliminating runtime parsing errors.

Database Strategy & Querying

The system intentionally avoids traditional relational databases (SQL) and heavyweight NoSQL engines (like MongoDB or Redis) to minimize deployment dependencies and

network latency. Instead, the NameNode relies on Go's native map data structures (`map[string][]string`) acting as a high-speed, in-memory key-value store.
Because the state resides entirely in RAM, "querying" the database does not require a query language like SQL or GraphQL. A CLI request to fetch a file blueprint simply executes a direct O(1) hash map lookup on the NameNode (e.g., `chunkIDs := n.State.Files[req.FilePath]`).

To persist this in-memory database, the NameNode leverages Go's standard `encoding/json` package. Whenever a state mutation occurs (like a file commit), the entire namespace is serialized and flushed to a local Document-style JSON file (`file_system_image.json`).
The DataNodes bypass database engines entirely, operating directly on the host operating system's file system. Physical `.dat` chunks are queried and manipulated using standard Go `os` package system calls (`os.Open`, `os.Create`, `os.Remove`) based on their exact file paths

Authentication

Authentication mechanisms (such as OAuth 2.0, JSON Web Tokens (JWT), or API keys) are explicitly unimplemented. The system is designed to operate within a trusted, isolated virtual network (`fractal-network`). It assumes that any client capable of routing traffic to the NameNode's IP address and port (9000) has been inherently authenticated at the network perimeter (e.g., via a corporate VPN or firewall). Consequently, all gRPC connections are established using `insecure.NewCredentials()`.

Authorization

Similar to authentication, fine-grained access control is not enforced within the cluster. The architecture does not define distinct user roles (Role-Based Access Control) or evaluate dynamic policies (Attribute-Based Access Control). Any client utilizing the CLI possesses equivalent, absolute administrative privileges, allowing them to freely execute destructive commands, like `burn` or overwrite critical data via `update`, across the entire namespace.

4.1 Technological details

- **Go (v1.26.2):** serves as the foundational programming language for the NameNode, DataNode, and CLI client. Exploited for its native concurrency primitives (`goroutines` and `channels`) to easily orchestrate the parallel scatter-gather network pings and background read-repair uploads without blocking the main execution thread. Utilized to compile 100% statically linked standalone binaries (`CGO_ENABLED=0`) that require no external host libraries.

- **gRPC (v1.81.0) & Protocol Buffers (v1.36.11):** *gRPC* replaces standard REST over HTTP/1.1 to facilitate high-performance, long-lived multiplexed TCP connections. Exploited specifically for its native `ClientStreaming` and `ServerStreaming` capabilities, allowing the system to continuously pump 64 MB files across the network in 64 KB increments. *Protobuf* exploited to generate strict, type-safe API contracts (`fractal.pb.go`) across the distributed components, and to compress the network payloads significantly better than JSON or XML.

- **Cobra (v1.10.2):** the `spf13/cobra` Go package is exploited to build the entire CLI architecture. Provides out-of-the-box scaffolding for command routing (e.g., `rootCmd.AddCommand`), argument validation (e.g., `cobra.ExactArgs(1)`), and automated help generation for the user-facing tools (`create`, `read`, `update`, `list`, `burn`).

- **Docker & Docker Compose:** exploited to simulate a complex distributed topology, complete with physical failure domains (Rack A, Rack B) and isolated bridge networking (`fractal-network`), locally on a single host machine. Leverages Docker Named Volumes to guarantee that the NameNode's `fsimage.json` and the DataNodes' `.dat` binaries persist across container destruction events.

5. Validation

5.1 Automatic Testing

The automated validation framework isolates core mathematical logic through unit tests before subjecting the fully integrated system to automated network sabotage.

Unit Testing Strategy

Individual components are tested in strict isolation using Go's native `testing` package without requiring network I/O or active databases.

Test Suite	Automation & Rationale	Tested Requirement
`TestRackAwareness`	**Rationale:** Validates the allocator distributes replicas across disparate failure domains. **Automation:** Passes mocked `DataNode` slices (Rack-A, Rack-B) and asserts the returned IPs span multiple racks.	Hardware Fault Tolerance
`TestEvaluateReadQuorum`	**Rationale:** Verifies the W=2/R=2 mathematical consensus logic. **Automation:** Feeds mock network `pingOutcome` channels (simulating missing or corrupted bytes) to ensure the client correctly separates healthy IPs from stale IPs.	Quorum Consensus

`TestChunkStore_Operations`	**Rationale:** Ensures DataNodes safely manipulate physical files. **Automation:** Leverages Go's `t.TempDir()` to generate a safe, ephemeral folder to execute and verify `CreateChunk`, `RequestChunkSize`, and `DeleteChunk` disk commands.	Data Persistence

End-to-End Integration & Chaos Engineering

System communication and distributed corner cases (crashes, partitions) are evaluated via automated chaos engineering. Instead of mocking the network, the `helpers_test.go` suite utilizes Go's `os/exec` package to dynamically execute `docker-compose up --build -d` directly within the test runner. This boots a production-equivalent, containerized cluster complete with the custom virtual bridge network.
During tests, specific DataNode containers are actively killed via `docker-compose stop` to simulate severe hardware failures mid-operation.

1. `TestWriteQuorum_FailsWhenQuorumNotReached`

- **Rationale:** Validates that the system rejects file uploads if the Write Quorum (W=2) cannot be met, protecting against partial writes during major outages.
- **Automation Strategy:** The test orchestrates a production-like environment by using Go's `os/exec` to run `docker-compose up --build -d`. It then issues a `docker-compose stop` command targeting three specific workers (`datanode-1`, `datanode-2`, `datanode-3`) to simulate multiple simultaneous hardware crashes before executing the `UploadFile` client function. It asserts that the client returns an error rather than silently failing.
- **Execution:** Executed via the Go testing toolchain targeting the `chaos_write_quorum_test.go` suite
- **Tested Requirement:** CP Theorem compliance (Consistency over Availability) and fault tolerance during severe write corner cases.

2. TestWriteQuorum_SuccessWhenOneNodeDies

- **Rationale:** Verifies write fault tolerance by ensuring a single node crash does not disrupt user uploads, provided two nodes remain available to satisfy the quorum.
- **Automation Strategy:** Boots the Docker cluster, gracefully stops a single container (`datanode-1`), and triggers `client.UploadFile`. The test asserts that the error is `nil`, proving the system seamlessly masks the failure from the user.
- **Execution:** Executed via the Go testing toolchain targeting the `chaos_write_quorum_test.go` suite.
- **Tested Requirement:** High availability during partial system degradation and graceful write error handling.

3. TestReadQuorum_FailsWhenTwoReplicasDie

- **Rationale:** Ensures strong consistency during downloads by proactively aborting operations if the Read Quorum (R=2) is compromised due to multiple node partitions.
- **Automation Strategy:** The test first uploads a dummy file using `uploadSimulation`, fetches the chunk blueprint from the NameNode via `GetFileLocations`, and identifies the specific IPs holding the replicas. It parses the container names and dynamically stops two of the host containers before asserting that `client.DownloadFile` aborts with an error.
- **Execution:** Executed via the Go testing toolchain targeting the `chaos_read_quorum_test.go` suite.
- **Tested Requirement:** Read consistency constraints and protection against stale or corrupted data during network partitions.

4. TestReadQuorum_SuccessWhenOneReplicaDies

- **Rationale:** Proves read fault tolerance by verifying the client can transparently reconstruct a file even if one of the physical replica hosts crashes mid-operation.
- **Automation Strategy:** Following a successful upload simulation, the test queries the NameNode for chunk locations, isolates exactly one container hosting a replica, and kills it using Docker Compose. It asserts that `client.DownloadFile` succeeds without surfacing errors to the user.
- **Execution:** Executed via the Go testing toolchain targeting the `chaos_read_quorum_test.go` suite.
- **Tested Requirement:** Seamless failover routing and network resilience during download corner cases.

5. `TestReadRepair_RestoresMissingChunk`

- **Rationale:** Validates the active data healing architecture by simulating silent data corruption and verifying the cluster patches itself in the background.
- **Automation Strategy:** The test uploads a file and fetches its blueprint. It establishes a rogue gRPC connection directly to one target DataNode and calls `DeleteChunk` to sabotage the physical disk. After forcing a client download, the test explicitly pings the sabotaged DataNode via `CheckChunk` to assert that the background repair thread successfully restored the missing data.
- **Execution:** Executed via the Go testing toolchain targeting the `chaos_read_repair_test.go` suite.
- **Tested Requirement:** Automated self-healing, bit-rot resilience, and eventual consistency reconciliation.

Test Execution

Developers trigger the test suites using standard Go commands:

- **Unit Tests:** `go test ./internal/...` executes the isolated component tests instantly.
- **Chaos Tests:** `go test ./tests/...` triggers the Docker container orchestrations and network sabotage suites.

6. Deployment

Software Prerequisites:

- **Docker Engine & Docker Compose**: required to host and orchestrate the isolated distributed cluster infrastructure.
- **Go (Version 1.26.2)**: required on the host machine to compile the CLI client from source.

Installation Instructions & Expected Outcomes

1. **Boot the Cluster**: Navigate to the project root and execute `docker-compose up --build -d`.
    - *Expected Outcome:* Docker builds the multi-stage images and launches one NameNode and four DataNode containers in the background.

```
PS C:\Users\zacca\Desktop\Laurea Magistrale Ing. Informatica Matteo\PRIMO ANNO\PRIMO SEMESTRE\Distributed Systems\fractal> docker compose up -d --build
[+] Running 7/7
 ✓ fractal-base-image              Built
 ✓ Network fractal_fractal-network  Created
 ✓ Container namenode               Started
 ✓ Container datanode-2             Started
 ✓ Container datanode-1             Started
 ✓ Container datanode-3             Started
 ✓ Container datanode-4             Started
```

2. **Install the CLI**:

   On Mac/Linux: execute `make build`.
    - *Expected Outcome:* the Go compiler generates a binary and moves it to `/usr/local/bin/fractal`, enabling the `fractal` command globally.

```
zacca@PerlaNera:/mnt/c/Users/zacca/Desktop/Laurea Magistrale Ing. Informatica Matteo/PRIMO ANNO/PRIMO SEMESTRE/Distributed Systems/fractal$ cd script/ ; make build
Building Fractal CLI for Mac/Linux...
Build complete! Moving to /usr/local/bin...
You can now use the 'fractal' command anywhere!
```

   On Windows execute: `.\script\make`.

    - *Expected Outcome:* the script compiles `fractal.exe` to a newly created `C:\Fractal` directory. The user must manually append this path to their Windows Environment Variables to use the CLI globally.

```
PS C:\Users\zacca\Desktop\Laurea Magistrale Ing. Informatica Matteo\PRIMO ANNO\PRIMO SEMESTRE\Distributed Systems\fractal> ./script/make
Building Fractal CLI...
Build complete! fractal.exe has been installed to C:\Fractal
```

Environment Configuration

The system requires zero manual configuration files; all parameters are injected as environment variables directly via the `docker-compose.yaml` file.

Component	Injected Environment Variables	Purpose
NameNode	`NAMENODE_PORT`, `FSIMAGE_PATH`	Defines the listening port (`9000`) and the hard drive path where the persistent JSON state is saved.
DataNodes	`DATANODE_ID`, `DATANODE_PORT`, `DATA_DIR`	Assigns a unique network identity, exposed port (e.g., `8001`), and physical storage path for `.dat` chunks.
DataNodes	`NAMENODE_ADDRESS`, `RACK_ID`	Tells the worker where to push heartbeats (`namenode:9000`) and assigns its physical failure domain (`rack-A` or `rack-B`).

Containerization Engineering

The `Dockerfile` is architected using a two-stage build pattern to maximize security and minimize footprint.

- **Stage 1 (Builder):** uses `golang:1.26-alpine` to download dependencies and compile the Go source code. The build commands specifically flag `CGO_ENABLED=0` and `GOOS=linux` to generate 100% statically linked binaries. This ensures the resulting executables do not rely on external C libraries present on the host operating system.
- **Stage 2 (Runner):** starts from a fresh, empty `alpine:latest` image. It installs basic network certificates (`ca-certificates`), copies only the compiled `namenode` and `datanode` binaries from the builder stage, and completely discards

the Go compiler and original source code. This drastically reduces the attack surface and image size.

The `docker-compose.yaml` file defines a custom bridge network named `fractal-network`. Deploying all containers onto this isolated network allows them to leverage Docker's internal DNS resolution. Instead of relying on hardcoded, fragile IP addresses, the DataNodes are engineered to push their heartbeats to the `NAMENODE_ADDRESS=namenode:9000` environment variable. Docker automatically resolves the `namenode` hostname to the correct internal container IP.

Containers are inherently ephemeral; if a container crashes or is explicitly destroyed, its internal file system is wiped. To prevent catastrophic data loss, the deployment script engineers strict volume mapping. Five distinct named volumes (e.g., `namenode_data`, `datanode1_data`) are declared at the bottom of the script and mounted to the `/app/data` directories inside the respective containers. This guarantees that the NameNode's `file_system_image.json` and the DataNodes' physical `.dat` chunks persist safely on the host machine across container lifecycles.

The `docker-compose.yaml` script explicitly controls the boot sequence of the cluster. All four DataNodes include a `depends_on: - namenode` directive. This guarantees that the Docker engine fully initializes the NameNode container before booting the worker nodes, preventing startup race conditions where DataNodes attempt to fire heartbeats at a master server that is not yet listening. Furthermore, port mapping (e.g., `8001:8001`) selectively punches holes through the container isolation, allowing the external CLI client running on the host machine to stream data directly into the isolated workers.

The deployment script completely eliminates the need for hardcoded configuration files inside the image. Every container's behavior is uniquely shaped at runtime via the `environment` block. This allows a single, generic `datanode` binary to be spun up as four distinct entities, dynamically injecting parameters like physical topology (`RACK_ID=rack-A` or `rack-B`), storage paths (`DATA_DIR`), and network identities (`DATANODE_ID`).

7. User Guide

Before interacting with the Fractal CLI, ensure the distributed cluster is fully booted via Docker Compose and the executable is installed globally on your host machine, as detailed in Section 6. Once the NameNode and DataNodes are active, the file system is managed entirely through the terminal.

Command	Syntax	Expected Outcome
Help	`fractal --help`	Prints the global help menu, listing all available commands, usage syntax, and active flags.
Status	`fractal status`	Queries the NameNode's RAM and prints a live health table of the cluster, showing active DataNodes count, active DataNode IPs, rack assignments, and the timestamp of their last heartbeat.
Create	`fractal create [path/to/local/file]`	Ingests the local file, fragments it into 64 MB chunks, streams replicas to the DataNodes (enforcing W=2 Quorum), and registers the file in the NameNode. Prints a success confirmation.
Read	`fractal read [remote_file_name]`	Fetches the routing blueprint, validates the R=2 Read Quorum, and seamlessly streams the binary data back from the DataNodes. The reassembled file is saved to an automatically generated `downloads/` directory.

Update	`fractal update [path/to/local/file]`	Uploads the file under a temporary timestamped name (e.g., `v1718293_file.txt`), atomically swaps the metadata pointer in the NameNode, and asynchronously purges the orphaned chunks from the DataNodes.
Burn	`fractal burn [remote_file_name]`	Eradicates the file entirely. The NameNode deletes the logical namespace mapping, and the client sends commands to the target DataNodes to permanently delete the physical `.dat` binaries from their local drives.
List	`fractal list`	Queries the NameNode and prints a formatted table displaying all archived files and their respective chunk counts. Returns a specific prompt if the cluster is empty.

If you wish to test the system's chunking and streaming performance before uploading real files, you can instantly generate dummy data directly from the terminal.
To generate a 1 GB raw binary file:

- On Mac/Linux: use `dd if=/dev/urandom of=random-1gb.bin bs=1M count=1024 status=progress`
- On Windows: use `fsutil file createnew random-1gb.bin 1073741824`

8. Future works

The following subsections outline unimplemented features, potential architectural improvements, and the specific engineering strategies proposed to integrate them in future development cycles.

Garbage Collection and Block Reports

Currently, if a file pointer is deleted but a DataNode is offline, orphaned physical chunks may remain on disk indefinitely.

- **Implementation:** I would implement an HDFS-style Block Report. Every hour, DataNodes would send a BlockReportMsg containing an array of all local Chunk IDs. The NameNode would compute a diff against its ClusterState namespace. Any reported chunk ID missing from the persistent ledger would trigger a DeleteChunkRequest sent back to the DataNode to free physical storage.

Under-Replication Healing

When a node dies, the cluster should autonomously self-heal to restore the replication factor to 3. Currently, the system relies on client-side Read-Repair, which requires a user to actively download the file to trigger healing.

- **Implementation:** I would introduce a background ReplicationMonitor goroutine in the NameNode. When a 15-second heartbeat timeout evicts a DataNode, this monitor would scan the ChunkLocations map, identify all chunks that fell below the optimal replication factor, and issue a new ReplicateChunkCommand. This would instruct healthy DataNodes holding the surviving replicas to stream the data to a new rack-aware target.

High Availability (SPOF Mitigation)

The centralized NameNode is a Single Point of Failure (SPOF). If the container crashes, the Data Plane is rendered completely inaccessible.

- **Implementation:** I would implement a SecondaryNameNode acting as a warm standby. It would periodically connect to the Primary via gRPC to download the fsimage.json and a running EditLog (recording mutations between JSON flushes). If the Primary drops offline, a consensus protocol (like Raft) would automatically promote the Secondary to Primary, ensuring zero downtime.

Namespace Federation

Because the ClusterState is held entirely in RAM for O(1) routing, the system's total file capacity is strictly bounded by the NameNode's physical memory.

- **Implementation:** To scale horizontally, I would implement HDFS-style Federation. The architecture would support multiple active NameNodes, each managing a distinct logical volume (e.g., NameNode A handles /users, NameNode B handles /archives). The CLI client would maintain a static Mount Table to route gRPC requests to the correct NameNode based on the file path.

Data Integrity (Checksums)

The system currently trusts that TCP prevents network corruption, but it cannot detect "bit rot" (silent hardware corruption on the physical hard drives).

- **Implementation:** I would integrate SHA-256 cryptographic hashing. The client would compute a hash for each 64 MB chunk prior to upload and store it in the NameNode's metadata. During a download stream, the client would recompute the hash on the fly. A mismatch would instantly trigger the existing Read-Repair scatter-gather mechanism.

Security (Authentication & Authorization)

Operating exclusively on a trusted network assumption limits deployment viability in multi-tenant environments.

- **Implementation:** I would secure the gRPC channels using TLS certificates. For Authentication, I would inject a JWT (JSON Web Token) into the gRPC request metadata using Interceptors (middleware). For Authorization, I would extend the File domain entity to include UNIX-style POSIX permissions (owner, group, read/write bits), allowing the NameNode to dynamically reject unauthorized mutations before requesting a Mutex lock.