Abstract

The need for data-intensive applications demands robust storage systems which must deliver high throughput, fault tolerance and scalability. The design goal of this project involves creating a distributed file system framework which implements the Google File System (GFS) model to store and process big files via low-cost hardware components.The system incorporates a Master Server which manages metadata operations along with chunk control and data storage replication functions. It works together with multiple Chunk Servers that keep actual data in pre-allocated fixed-size chunks (traditionally 64MB). The Client interacts with the Master by requesting metadata then conducts direct gRPC connection with Chunk Servers regarding read/write operations.Data consistency remains assured because of the replication strategy as well as the leader election process and periodic heartbeat monitoring. Clients use CLI to upload their files that get properly divided across all accessible servers for distribution. Together Docker functions as a deployment platform for the Go-developed system that provides both security and simple deployment capabilities.The project demonstrates practical experiences of designing distributed systems while building fault tolerance and load balancing capabilities and concurrent processing via worker pools. The system serves as the base for building large-scale cloud-native file storage that satisfies modern big data requirements and active distributed computing needs.

Table of Contents

Declaration i
Certificate ii
Abstract iii
Table of Contents iv

List of Figures **vi**

List of Tables **vii**

Gantt Chart **viii**

1 Introduction **1**
1.1 Project Definition 2
1.2 Project Objectives 2

2 Literature Survey **4**
2.1 Related Work 5
2.2 Tools and Technologies 6

3 Methodology **7**
3.1 Planning Methodology 7
3.1.1 Pre-Production and Planning Stage 7
3.1.2 System Architecture Design 8
3.1.3 Servers Registration & Watch Out 9
3.1.4 Chunking & File Handling Strategy 10
3.1.5 Distributed Write Workflow in ShardFS 10
3.1.6 Read Workflow in ShardFS 15
3.1.7 Delete Workflow in ShardFS 18
3.1.8 Heartbeat Mechanism & Fault Monitoring 19
Heartbeat Transmission 19
Heartbeat Payload Data 19
Failure Detection 20
Reliability in Dockerized Environments 20
3.1.9 Chunk Server Scoring & Load-Aware Prioritization 20
Score Calculation Logic 21
Score-Based Prioritization Using a Max-Heap 21
Dynamic Re-Evaluation 21
3.1.10 Worker Pool Architecture 21
Motivation 22

Benefits . . . 22
3.1.11 Deployment with Docker . . . 22
Persistent Storage with Docker Volumes . . . 22
3.1.12 Network Configuration with Docker Compose . . . 23
Key Features Enabled by Compose: . . . 23
3.1.13 Sequence Diagram . . . 24

4 Results **31**
4.1 Project Outcomes . . . 31
4.2 Learning Outcomes . . . 32
4.3 Real World Applications . . . 32
4.4 Project Demonstration . . . 33

5 Conclusion **37**
5.1 Future Developments . . . 37
5.2 Data Flow Diagrams . . . 39

Bibliography **40**

List of Figures

3.1.1 HLD Diagram of ShardFS 8
3.1.2 Upload Level 1 DFD 15
3.1.3 Download Level 1 DFD 18
3.1.4 Delete File Level 1 DFD 19
3.1.5 Network in Docker Containers 23
3.1.6 Sequence Diagram for Chunk Server Registration. 24
3.1.7 Sequence Diagram for Delete Workflow. 25
3.1.8 Download Workflow Part 1 26
3.1.9 Download Workflow Part 2 27
3.1.10 Download Workflow Part 3 28
3.1.11 Upload Workfow Part 1 29
3.1.12 Upload Workfow Part 2 30

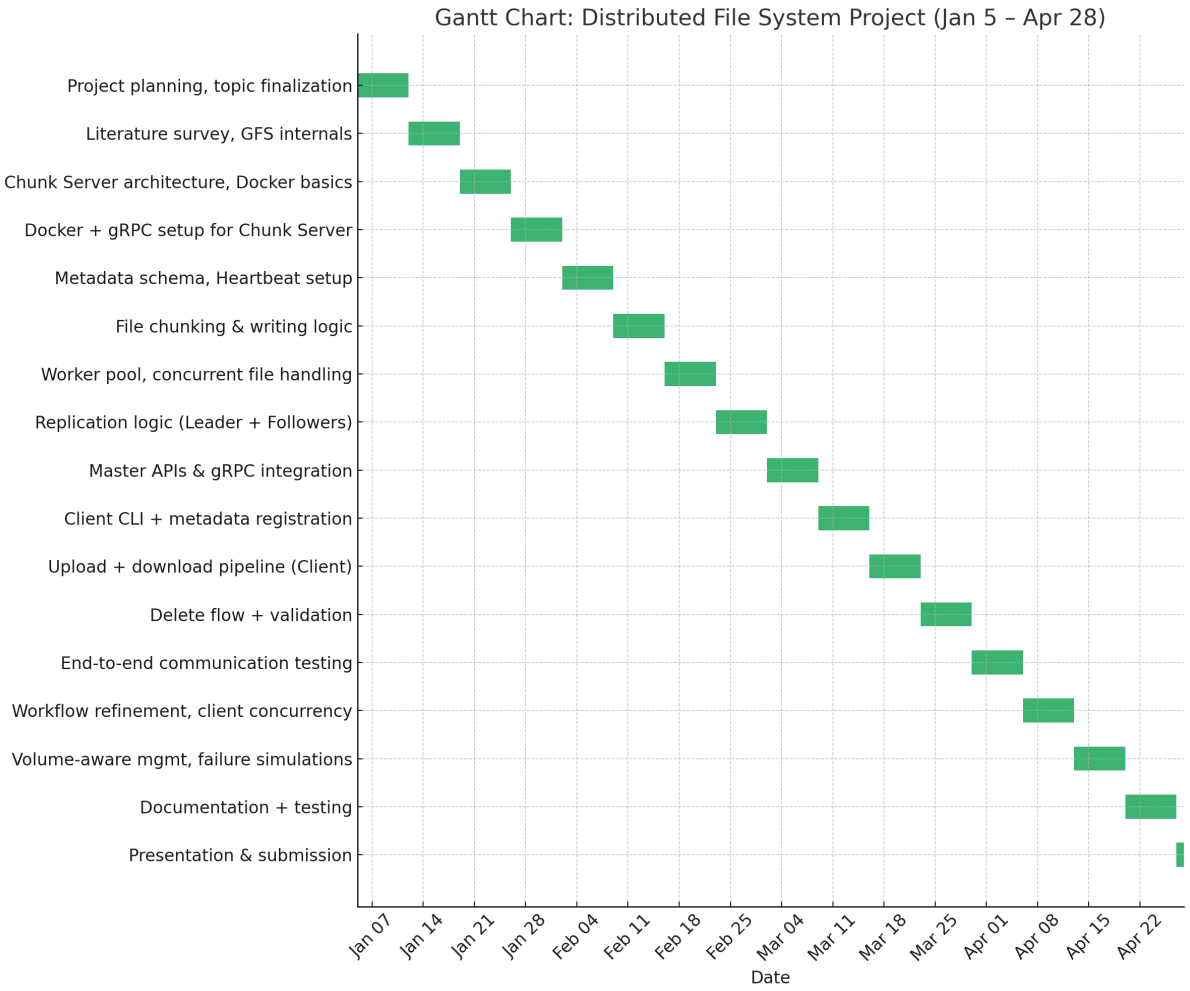
4.4.1 Chunk Server Registration Log 33
4.4.2 Chunk Server Heartbeat Log 33
4.4.3 Heartbeat Log to Master Server 33
4.4.4 Write Command Client Side 34
4.4.5 Storing Job (Leader Server) 34
4.4.6 Replication Log to Replica Servers 34
4.4.7 Leader Server Election 35
4.4.8 Replica Selection and Storage in MongoDB 35
4.4.9 Read Log Client Side. 35
4.4.10 Read Master Log. 36
4.4.11 Leader Server Read Log 36

5.2.1 Level 0 DFD 39
5.2.2 Master Server Internals 39
5.2.3 Chunk Server Internals 40

List of Tables

3.1.1 Heartbeat Payload Data Fields 20
3.1.2 Score Calculation Metrics and Weights 21

Gantt Chart



ShardFS : A Lightweight GFS-Inspired Distributed File System

Chapter 1

Introduction

The exponential growth of data in modern computing has rendered traditional file systems inadequate for meeting the performance, scalability, and fault tolerance requirements of contemporary applications. With organizations increasingly reliant on data-driven operations, cloud-native services, and machine learning pipelines, the need for robust distributed storage systems has become critical. This project presents ShardFS, a lightweight distributed file system inspired by the design principles of the Google File System (GFS). GFS was originally developed by Google to support its large-scale data processing infrastructure and is known for its sophisticated architecture, automatic failure detection, high throughput, and ability to manage extremely large files through distributed, redundant storage.

ShardFS adopts three essential components from GFS to achieve a similar level of efficiency and reliability: a Client interface, a centralized Master Server, and multiple distributed Chunk Servers. Clients begin the process by splitting files into fixed-size chunks of 64MB. These chunks, along with metadata such as file size, chunk count, and hash values, are sent to the Master Server, which is responsible for managing all metadata and orchestrating the allocation of chunks across Chunk Servers. The Master Server does not store any actual file data but plays a vital role in maintaining file-to-chunk mappings, assigning chunk IDs, electing leader servers for replication, and regularly monitoring the health of all Chunk Servers via heartbeat signals.

Chunk Servers are responsible for storing the actual chunk data and handling replication tasks. The replication protocol implemented follows a pipeline model: once the leader Chunk Server receives a chunk from the client, it forwards the data to the first follower, which then forwards it to the second follower. Acknowledgements flow in the reverse order, ensuring that full replication is completed before the client is notified of success. This model enhances reliability and consistency while distributing the replication load. To further improve performance and scalability, ShardFS leverages Golang's lightweight concurrency model using goroutines and worker pools within the Chunk Servers. These worker pools handle concurrent operations such as reading, writing, and replication efficiently, maximizing CPU utilization.

For communication between components, ShardFS uses gRPC, which ensures fast, reliable, and structured message exchange. Deployment is fully containerized using Docker, enabling platform-independent setup and simulation of a distributed multi-node environment.

Docker Volumes are used to maintain persistent chunk storage, while Docker Compose facilitates service orchestration and internal networking between the Master, Clients, and Chunk Servers.

By separating metadata management from chunk storage operations, ShardFS allows for simplified fault recovery, efficient scalability, and modular design. This system supports both write and read paths efficiently, retrieving file metadata and chunk locations as needed, and enabling parallel chunk downloads with hash-based verification during read operations. Overall, ShardFS functions as both a testable prototype and a teaching tool, demonstrating key distributed file system principles such as metadata abstraction, chunk-based storage, replication, fault tolerance, and scalable deployment. It brings the core architectural strengths of GFS into a modern, educational context, offering a practical and simplified implementation suitable for current technical exploration and learning.

1.1 | Project Definition

The primary mission of this project involves engineering and implementing an inspired version of the Google File System (GFS) distributed file system architecture. The system enables scalable fault-tolerant high-throughput file storage through server distribution of data accompanied by master server metadata control.

The system fragments large files into equal 64 MB-sized chunks before partitioning them among multiple servers through client-server protocol applications of gRPC. The system integrates concurrent operation worker pools alongside health monitoring procedures through heartbeat mechanisms and leader election protocols for replication consistency.

The project implementation uses Golang as a programming language combined with Docker containers and enables communication through gRPC and deploys services using containerized strategies. The project aims to display that storage systems mirroring GFS functionality can be achieved through contemporary development and orchestration frameworks.

1.2 | Project Objectives

- Develop a metadata-centric Master Server capable of maintaining consistent file-to-chunk mappings, tracking chunk server status, and orchestrating overall coordination within the distributed system.
- Build resistant Chunk Servers to support high-speed concurrent reading and writing that functions for data replication across policies of redundancy.
- The system should let clients upload and download files by using dynamic chunking while performing SHA-256 hashing for data verification.
- Heartbeat monitoring and replication acknowledgment strategies functioning across the system will enable the detection of failures to preserve data availability.
- A Docker-based container deployment model used to create modular, scalable, isolated environments for every component within the distributed file system architecture.

- The system uses gRPC to establish an efficient communication pipeline between clients and the master server and clients and chunk servers for structured high-performance metadata and file chunk transmission.

Chapter 2

Literature Survey

Modern Computing systems face a primary requirement to handle data efficiently at large scales effectively. Modern distributed systems need this solution primarily in applications that involve cloud infrastructure and big data pipelines and high-performance computing.Google's landmark publication of the Google File System (GFS) provided essential elements for developing storage systems that combine large-scale expansion capabilities with fault tolerance through its design. The architecture plan separates metadata handling from data storage through a distinct organizational design. The system uses fixed-size chunking in combination with leader-based replication to uphold its design solution. Many organizations around the industry have utilized this concept as the basis to store and manage massive datasets on commodity hardware systems. The file system architecture of GFS prompted the development of Hadoop Distributed File System (HDFS).

These systems were developed to handle massive-scale processing of distributed big data formations.HDFS mirrors many of GFS's design patterns, such as the master-worker model (NameNode and DataNode), block replication, and heartbeat mechanisms for health monitoring.

Studies of HDFS systems confirmed the essential assumptions of GFS which showed that the framework had its strong aspects in data-intensive workloads and its limitations, such as the single-point metadata bottleneck.Ceph and GlusterFS came after HDFS to tackle the acknowledged issues faced by GFS with its design.However different approaches tried to distribute metadata while providing POSIX-compatible access to storage but they generally incurred extra complexity.

The principles of distributed file systems inspired commercial developments that led to the creation of Amazon S3 and Google Cloud Storage through object-based models which enhance scalable access from various systems. These storage systems use RESTful APIs together with eventually consistent models for creating global access alongside well-designed replication methods and failover systems. Academic research examines how distributed file systems must choose between consistency and availability and partition tolerance based on the CAP theorem. Technical research regarding RAFT and Paxos protocols has developed election and consensus methods which influence modern systems' replication and metadata consistency practices.

The introduction of Kubernetes and Docker technologies during the last few years has prompted organizations to embrace containerized microservice development for modular

service implementation including chunk servers and metadata managers. The approach improves separation together with tracking and growth capacity which matches GFS platform guidelines. The combination of tools which use gRPC-based inter-service communication creates an extremely performant solution that is language-agnostic for internal RPC mechanisms. This project benefits from past literature that guides its engineering process by integrating conventional distributed file system methods with container orchestration features along with cloud-based solutions.

A functional prototype will emerge from integrating best-practice methods described within previous research about chunk-based replication and leader-based coordination together with heartbeat monitoring and centralized metadata management.

2.1 | Related Work

Research and industrial practitioners have extensively studied distributed file systems. The Google File System (GFS) came into existence as Google's initial major distribution file system which aimed to meet the company's enormous needs for data processing. The core concepts presented in GFS publications now serve as basic elements for every distributed storage design developed since its initial release. Fundamental principles establish the use of big file partitions and deployment of a single metadata controlling component with data protection systems based on replication methods.

Hadoop Distributed File System represents the most popular open-source implementation of HDFS due to its direct inheritance of concept from GFS. The system uses a name-node centralized control point that distributes data chips across the network to achieve high parallel processing through dedicated MapReduce implementation. The HDFS system achieves top-notch scalability in production environments at Facebook and Yahoo while managing huge big data solutions.

Ceph represents a different distributed file system than GFS which decentralizes its metadata operations making the system more resilient to failures. Ceph handles strong consistency operations while it serves the petabyte level scale and supports multiple storage protocols including object, block, and file systems. The CRUSH (Controlled Replication Under Scalable Hashing) data distribution algorithm stands as a key innovative aspect in the file system.

Distributed storage occurs in Amazon S3 and Google Cloud Storage where they present object storage by replacing traditional file system interfaces. These systems provide high reliability and uptime levels because they use worldwide supporting infrastructure together with robust replication mechanisms. GFS-inspired concepts have been converted into cloud-operation user-centric services which deliver worldwide solutions to millions of end clients.

The evolution of file system design received contributions from different distributed systems including GlusterFS and MooseFS. GlusterFS achieves its scalability through translator-based architecture which combines modularity elements while MooseFS provides POSIX API integration with auto-rebalancing capability for fault-tolerance enhancement.

Today's systems such as etcd and Consul implement RAFT and Paxos distributed consensus algorithms for metadata management consistency since GFS used to handle it through

a single master. Today distributed file systems rely on these systems to deliver added guarantees and higher availability.

This project continues existing GFS architectural concepts with updates that include containers provided by Docker and gRPC communication and Go language parallel processing capabilities. Our objective is to create an educational distributed file system system through a modular and scalable prototype by analyzing and combining previous research works.

2.2 | Tools and Technologies

Interface

The system uses Cobra and Go to develop its command-line interface (CLI) The distributed file system interface runs under the Golang language to enable user access for file system operations. uploading, downloading, and deleting files. This interface facilitates communication with both the Master Server and Chunk Servers in a user-friendly and efficient manner.

Protocols

The system uses gRPC for process communication, which enables rapid transmission of structured data while making messages resistant to type errors. The communication protocols work coordinated to establish reliable metadata and file chunk transmission management.

Deployment

The Docker technology enables the creation of containerized environments for both Master Servers and Chunk Servers that provide development environments with isolated reusable conditions. The persistence of chunk server data runs through Docker volumes so that restarts do not interrupt data continuity. The approach makes the deployment process simpler and enables maintenance of scalability.

Database

The system implements MongoDB as its main storage for metadata which organizes file-to-chunk mapping data alongside chunk server status. The system implements MongoDB for centralized metadata handling and updates which supports both robustness and scalability in the distributed environment.

Backend Development and Tools

The backend utilizes all its components in Go (Golang) which optimizes distributed operations by working with goroutines and channels for concurrent management. All development tools for this application include Visual Studio Code for writing code and GitHub to manage version control and Docker CLI to manage containers and Postman for testing APIs. Visual system architecture and workflows are displayed through the usage of Eraser.io.

Chapter 3

Methodology

The methodology used to construct the distributed file system based on Google File System (GFS) adopts a modular framework with layered components together with scalability capabilities. Steps were taken to build each component which modeled genuine large-scale file system conduct while maintaining an adaptable and system architecture for future needs.

3.1 | Planning Methodology

The section details an implementation flow at every step alongside design principles and the communication framework linking system components :

3.1.1 | Pre-Production and Planning Stage

Research about Google File System (GFS) design principles and related distributed file storage systems such as HDFS and Ceph started the project's initial phase. A fundamental research foundation defined system requirements which involved metadata management and chunk segmentation and file duplication and reliable system communications.During the planning phase the project designed an overall system architecture which combines three essential elements: Client along with Master Server and Chunk Servers.

The system chose to split data management (Chunk Servers) from control functions (Master) to achieve better scalability and enhanced maintainability.

Key planning deliverables included:

Selection of Go (Golang) for implementation due to its strong concurrency support.

Implementing gRPC functions as the communication framework to conduct real-time distributed networking tasks.

Significant benefits for service containers and environment simulations come from DockerDeployments in distributed systems.

The workflow includes service orientation with modular design to test individual components independently.

The development process divided into stages which included designing metadata schema structures together with building file chunking protocols and establishing network connections and implementing the replication consistency framework. Decided to start building the write functionality first then move to develop the read path before adding delete functionality capabilities to the system. The pre-production planning created a specific sequence of implementation steps that guided the system development process in a technical and phased manner.

3.1.2 | System Architecture Design

The project starts its development by creating a three-tier system which includes the following main parts:

- Clients play the main role through their system in starting file upload/download process requests. The system performs tasks like file chunking while it produces metadata alongside controlling operations between the Master Server and Chunk Servers.
- The Master Server serves as the central management system for handling all file metadata as well as chunk placement along with chunk server health monitoring and leader election for replication processes.
- Chunk Servers work as storage nodes which accept file chunks and carry out all read/write procedures as per requests from clients and directions from the master server.

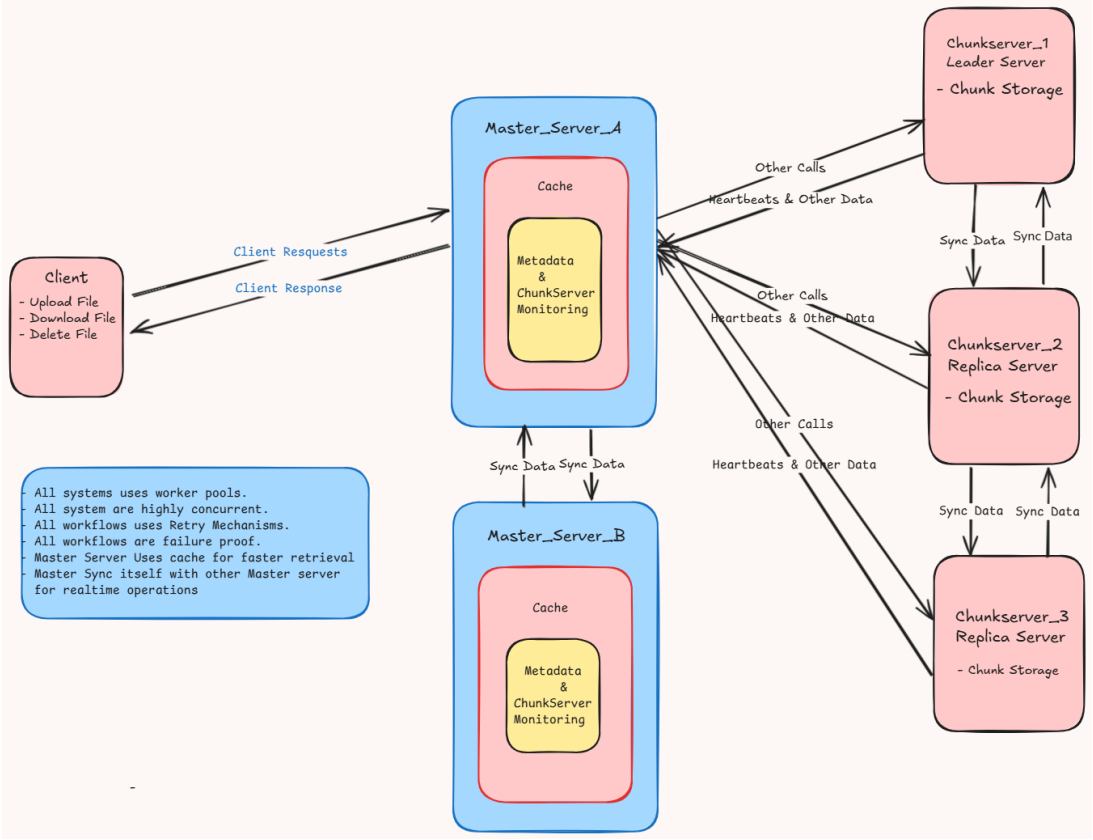


Figure 3.1.1: HLD Diagram of ShardFS

3.1.3 | Servers Registration & Watch Out

The Distributed File System implements ChunkServer registration as its essential operational base. Master server identification, heartbeat participation, and MongoDB registration establish recognition of each ChunkServer during registration. Through the Heartbeat Manager, the system maintains state awareness while achieving tolerance to failures by implementing retries.

```
message RegisterChunkServerRequest {
  string server_id = 1;
  string address = 2;
}
message RegisterChunkServerResponse {
  bool success = 1;
  string message = 2;
}
```

Sequence of Events

1. **Initial Registration Request**
   - The ChunkServer starts the registration procedure through an RPC call to `RegisterChunkServer` that includes `server_id` and `address`.
2. **Master Processes Request**
   - The Master server follows several steps when handling the request it receives.
   - Updates the `HeartbeatManager.chunkServers` in memory with the server ID and address.
   - The operation on MongoDB's `server_status` collection through `UpdateOne` uses an upsert set to true to mark the server status as active. The operation with `upsert = true` ensures the server registers through either an update or insert process.
3. **Success Path**
   - Both in-memory Heartbeat update and MongoDB operation to mark the server active as active succeed simultaneously.
   - The master server returns a response that includes `success: true`, accompanied by the message "Registered."
   - A `HeartbeatManager` on the ChunkServer starts its periodic heartbeat signals after its initialization.
4. **Failure Path**
   - The system will fail to register when a MongoDB problem occurs, or the data shows conflicts.
   - The server responded with `success: false`, and the message was "Registration failed."
   - The ChunkServer writes the error to its logs before awaiting five seconds and then attempts another `RegisterChunkServer` request.

Database Interaction

- MongoDB saves server metadata in its database's `server_status` collection.
- The fields typically updated include:
  - `server_id`
  - `address`
  - `active: true`
- The upsert operation enables dynamic server addition through its update process.

Resilience & Retry

- During a registration failure, the client records the issue before attempting the process again after waiting 5 seconds.
- The implementation provides robust availability and fault tolerance benefits while the network or master becomes unstable.

3.1.4 | Chunking & File Handling Strategy

To ensure scalability and efficient data handling, the system employs a fixed-size chunking strategy, dividing files into 64MB chunks. This chunk size balances metadata overhead with parallelism and network efficiency. Each chunk is assigned a unique Chunk ID, and a SHA-256 hash for integrity verification. The metadata, which includes chunk size, replication level, assigned servers, and timestamps, is first sent to the Master Server to coordinate storage and replication. On the client side, a multithreaded worker pool is used to concurrently split, hash, and stream chunks to the designated leader ChunkServers. This concurrent architecture boosts upload throughput and optimizes CPU usage, enabling high-performance writes even for large files.

3.1.5 | Distributed Write Workflow in ShardFS

ShardFS uses a parallel workflow with strong fault tolerance and concurrency for file uploads across a distributed system because of Google File System (GFS) and Hadoop Distributed File System (HDFS). The system uses client-side chunking and metadata registration parallel chunk uploads and chained replication with chained acknowledgments while it operates through gRPC calls and Protocol Buffers. The client-side worker pool creates maximum concurrency, whereas the pipeline replication design lowers the Master Server workload and maximizes network efficiency. An explanation of ShardFS write workflow operations includes analyzing its scalable replication system through several successive steps.

Creation and Registration of Metadata

1. The client gets the file ready for upload before the writing process starts. The chunking module (`chunking.go`) splits the file into 64MB chunks and calculates SHA-256 hashes and sizes for each chunk.
2. The client creates a `FileMetadata` object (`metadata.go`) that contains:
   - File name, chunk count, total size, and format (by default, binary).

- Hashes and arrays of chunk sizes are used to confirm integrity.
- Client ID, timestamp, redundancy level (default: 3), priority (default: 1), and compression flag (default: false).

3. Through the `RegisterFile` gRPC call (`client.go`), this metadata is transmitted to the Master Server enclosed in a `RegisterFileRequest` message.

```
message RegisterFileRequest {
  string file_name = 1;
  string file_format = 2;
  int64 total_size = 3;
  int32 chunk_count = 4;
  repeated int64 chunk_sizes = 5;
  repeated string chunk_hashes = 6;
  int64 timestamp = 7;
  string client_id = 8;
  int32 priority = 9;
  int32 redundancy_level = 10;
  bool compression_used = 11;
}
```

The Master Server distributes file IDs after selecting leader ChunkServers with replica ChunkServers (followers) for each chunk through the redundancy level calculation and server scoring system which it performs every 10-second heartbeat interval as the server scores get updated. It then stores this mapping information in a database (e.g., MongoDB, implementation-dependent). A RegisterFileResponse is returned to the Master Server by its response.

```
message RegisterFileResponse {
  string file_id = 1;
  string leader_server = 2; // Unused in client logic
  map<int32, ChunkServers> chunk_assignments = 3;
  bool success = 4;
  string message = 5;
}

message ChunkServers {
  string chunk_hash = 1;
  int32 chunk_index = 2;
  string leader = 3;
  repeated string replicas = 4; // Follower addresses
}
```

The registration task is submitted as a RegisterFileTask to a client-side worker pool (worker/task.go, worker/pool.go), which processes tasks concurrently with a fixed number of workers (e.g., 10). The worker pool ensures efficient metadata registration within a 10-second timeout, enhancing scalability for large files.

Concurrent Chunk Uploads to Leader ChunkServers

The client starts uploading file chunks to Leader ChunkServers which were provided in the

`chunk_assignments` section during the `RegisterFileResponse` receipt. The transmission of each chunk occurs through a streaming `UploadChunk` gRPC call which sends 1MB data parts to maximize network efficiency (`client.go`).

```
rpc UploadChunk (stream ChunkUploadRequest) returns (ChunkUploadResponse);

message ChunkUploadRequest {
  string file_id = 1;
  string chunk_hash = 2;
  int32 chunk_index = 3;
  bytes data = 4;
  string leader = 5;
  string follower1 = 6;
  string follower2 = 7;
}

message ChunkUploadResponse {
  bool success = 1;
  string message = 2;
  string file_id = 3;
  string chunk_hash = 4;
}
```

The client sends `UploadChunkTasks` to the worker pool for processing multiple tasks simultaneously across multiple workers. The chunk needs verification because the leader ChunkServer uses the provided SHA-256 hash to check data integrity before local storage and subsequent chained replication initiation. The `ChunkUploadRequest` from the client contains the follower addresses (`follower1`, `follower2`) to allow the leader control of the replication process.

Data Redundancy Mechanism in ShardFS

The data redundancy mechanism in ShardFS adopts a chained replication model following GFS and HDFS standards which operates through an optimized network and requires minimal interaction with the Master Server. The Master Server directs the replication process during metadata registration while it determines chunk topologies which include leader and follower entities. The replication flow functions as follows:

Replication Flow

1. **Client → Leader:** The client streams the chunk data to the leader ChunkServer via the `UploadChunk` RPC. The leader writes the chunk to its local storage after verifying the SHA-256 hash.
2. **Leader → Follower 1:** After successful local storage, the leader streams the chunk to the first follower (Follower 1) using the `SendChunk` gRPC call:

```
rpc SendChunk (ReplicationRequest) returns (ReplicationResponse);

message ReplicationRequest {
  string file_id = 1;
  string chunk_hash = 2;
```

```
  int32 chunk_index = 3;
  bytes data = 4;
  repeated string followers = 5; // Remaining followers in the chain
}

message ReplicationResponse {
  bool success = 1;
  string message = 2;
  string chunk_hash = 3;
  map<string, bool> status_map = 4; // Replication status per follower
}
```

The `followers` field lists the remaining followers (e.g., Follower 2), guiding the replication chain. Follower 1 verifies the chunk's hash, stores it locally, and prepares to forward it.

3. **Follower 1 → Follower 2:** Follower 1 streams the chunk to the next follower (Follower 2) using another `SendChunk` call, with the `followers` field updated to exclude itself. Follower 2 verifies and stores the chunk, completing the replication chain.

Acknowledgment Flow

The acknowledgment process is also chained to ensure replication integrity:

1. **Follower 2 → Follower 1:** After storing the chunk, Follower 2 sends a `ReplicationResponse` to Follower 1, indicating success or failure.
2. **Follower 1 → Leader:** Follower 1 consolidates its own status (success if the chunk was stored) with Follower 2's response and forwards a `ReplicationResponse` to the leader. The `status_map` in the response tracks the replication outcome for each follower.
3. **Leader → Client:** The leader aggregates the statuses from the chain, ensuring all downstream acknowledgments (ACKs) are received. Only then does it send a `ChunkUploadResponse` to the client, confirming that the chunk was successfully stored and replicated across the pipeline.

This chained approach minimizes the leader's network load, as it only communicates with Follower 1, and ensures that replication is complete before reporting success to the client.

Benefits of the Pipeline Model

The chained replication architecture provides various beneficial features.

- The Master Server accomplishes its role by performing only metadata registration tasks and replication topology decisions while excluding data processing duties from its responsibilities. Data transfer operations move to ChunkServers through this system, which decreases the Master's workload and reduces latency.
- When clients transmit data to the leader, they accomplish their transfer in one stream, which eventually passes to all followers through the chain, reducing network costs from directly sending to all followers.

- The chained acknowledgment process guarantees replication integrity through its ability to maintain successful chunk storage at every follower before the client receives verification confirmation.
- The pipeline model enables horizontally scalable operations because new ChunkServers can both join and leave the replication network automatically without interrupting current processes.

Dynamic Adaptability and Fault Tolerance

ShardFS improves both the dynamic adaptation and fault resilience capabilities of the replication system through built-in sophisticated mechanisms.

- The Master Server uses server score recalculation, which happens every 10 seconds through ChunkServer heartbeat messages. The Master determines system optimization through server score calculations that show server conditions alongside resource data and network performance to select the best leaders and followers for different portions. The cluster experiences efficient resource use and balanced distribution of workload because of this feature.
- The Master distributes leadership responsibilities for big files across multiple leader ChunkServers to maintain high network accessibility and avoid system blockages.

Failure Handling

- The Master system discovers server failures through its heartbeats monitoring process. A ChunkServer becomes unavailable after failure to respond to heartbeats, and the Master system removes it from replication patterns.
- If the Master discovers a replication failure, it automatically directs duplicate responsibilities to available ChunkServers for maintaining the established redundancy threshold (defining the default value as 3).
- The background maintenance processes of the system repair lost or damaged replicas to deliver eventual consistency without breaking client operations.

The system features allow ShardFS to modify its behavior in response to cluster changes while maintaining its reliability during failures.

Retry and Failure Handling

ShardFS uses a detailed error handling system through multiple layers to provide reliable operation during write and replication steps:

- The client attempts three upload retries for chunks through the leader node while increasing the delay between attempts exponentially from 1 to 2 seconds and 3 seconds (client implements this behavior in `UploadChunk`). The system records failures that include the chunk hash together with server details.
- The leader retries the replication process for Follower 1 or 2 twice consecutively with a delay that sends replication tasks to server-side worker pools. The replication chain operates with fault tolerance by implementing this method.
- The `read.go` code contains a client-side failure map that monitors defective ChunkServers to eliminate further download attempts to such unreliable servers. This enhances fault isolation.

- Redundancy-based resilience determines successful chunk upload when data exists on the leader server node and one more server node is chosen based on the configuration value set to 3 by default. This tolerates partial replication failures.
- The system records all errors, starting from upload and continuing through replication and metadata registration, by logging thorough metadata such as chunk hash values, server locations, and error messages alongside client-side information. Server-side systems automatically record all failures similarly.
- Checkout and upload process the client-side worker pool functions by collecting reports on errors through `errChan` in `write.go`. The system gathers all failed tasks like chunk uploads and metadata registration to inform the user.

The system uses this approach to separate faults while reducing data losses and maintaining stable operation during critical situations.

Final Acknowledgment and Validation

The client verifies operational success by examining the error channel of the worker pool after all chunks have been uploaded and duplicated. The client logs a success message (e.g., "File write completed successfully") and informs the user after no errors are reported. The client system performs thorough error aggregation, then displays specific details and alerts the user about the upload and replication failures.

Leader ChunkServers transmit their `ChunkUploadResponse` messages as the last confirmation that both storage and replicated acknowledgments (using chained replication) executed smoothly. All relevant followers have successfully stored the chunk according to the `status_map` in the replication chain through this acknowledgment. The Master Server does not require a separate acknowledgment because the accomplishment of metadata registration and chunk uploads constitutes sufficient evidence to confirm successful write operations. All operations include maximum waiting periods, such as 10-second metadata tasks and flexible upload timeouts to avoid prolonged delays.

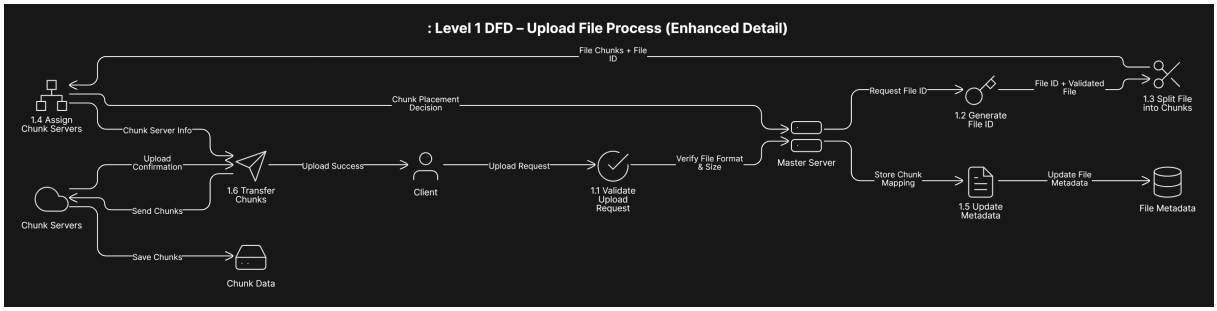


Figure 3.1.2: Upload Level 1 DFD

3.1.6 | Read Workflow in ShardFS

ShardFS employs a dependable read operation for file retrieval which makes use of its distributed storage architecture alongside its metadata system and chained replication approach used for writing. The read workflow begins with a Master Server query for chunk metadata followed by data retrieval from leader and follower ChunkServers and subsequent data verification process which runs through a client-side worker pool. The

process contains automatic retry logic together with failure tolerance features which guarantee data consistency and system availability during server outages. The diagram shows the stages which make up the ShardFS read workflow.

Metadata Query and Retrieval :

When the client asks the Master Server for file metadata in order to find the chunks linked to a file, the reading process starts. The GetFileMetadata gRPC call (client.go) is used by the client to send a GetFileMetadataRequest message:

```
message GetFileMetadataRequest {
  string file_name = 1;
  string client_id = 2;
}

message GetFileMetadataResponse {
  string file_format = 1;
  int64 total_size = 2;
  int32 chunk_count = 3;
  repeated string chunk_hashes = 4;
  map<int32, ChunkServers> chunk_assignments = 5;
  string client_id = 6;
  bool success = 7;
  string message = 8;
}

message ChunkServers {
  string chunk_hash = 1;
  int32 chunk_index = 2;
  string leader = 3;
  repeated string replicas = 4;
}
```

A GetFileMetadataResponse transmission from the Master Server includes file metadata obtained from MongoDB, which contains file format, size and chunk count data, chunk hashes, and leader and follower ChunkServer locations indexed by chunk position. The system distributes the metadata query to the client-side worker pool through task processing (worker/task.go, worker/pool.go), where ten concurrent workers operate under the 30-second timeout.

Concurrent Chunk Downloads :

The client starts parallel chunk transfers through the DownloadChunk gRPC call after receiving GetFileMetadataResponse (client.go). The system retrieves data streams while validating the input using the SHA-256 hash format.

```
rpc DownloadChunk (ChunkRequest) returns (stream ChunkData);

message ChunkRequest {
  string chunk_hash = 1;
  int32 chunk_index = 2;
}
```

```
message ChunkData {
  string chunk_hash = 1;
  int32 chunk_index = 2;
  bytes data = 3;
}
```

The client uses DownloadChunkTasks to feed the worker pool, so they execute multiple chunks in parallel while managing the download operation. The download process applies a fault-tolerant approach to its procedure.

- The client retrieves the chunk from the leader ChunkServer, which is designated in `chunk_assignments`.
- The transmission of chunk data occurs through the leader server to clients, who next verify hash code metadata using the SHA-256 method.
- Local storage of the chunk occurs when the download operation completes successfully.
- After a leader fails connection timeout or hash mismatch, the client will try three times with waiting intervals of 1 second, 2 seconds, and 3 seconds.
- If the leader's attempts fail, the client tries followers in the replicas list for up to three attempts each.
- The worker pool gathers downloaded data from the `dataChan` results and inserts error data into the `errChan` so that task completion precedes the assembly process.

Fault Tolerance & Retry Mechanism : A diagram shows how ShardFS employs fault tolerance together with retry strategies for encountering server breakdowns and network problems.

- The client makes up to three consecutive attempts with increasing delay between each try for each download request between leaders and followers. The `client.go` application logs both the sequential download attempts together with the ultimate result for debugging purposes in the `DownloadChunk` method.
- The `client read.go` file maintains a failure tracking system which records ChunkServers that experience issues due to network problems or hash verification errors. The read operation advantageously automates failure isolation by marking servers that fail to access during the current process.
- A chunk download becomes unsuccessful when all servers reach exhaustion yet the client continues processing other remaining chunks by logging the failure. The system provides notification to users about incomplete success or total failure according to the number of unattained chunks.
- The diagram indicates a background approach for fixing broken chunks. The Master Server exploits heartbeat detections of failed ChunkServers to direct available servers for replica restoration during which read processes continue without interruption.
- The worker pool aggregates errors from `DownloadChunkTask` executions via `errChan`. Failed downloads send reports to users containing detailed information about chunk hash and server addresses as well as error messages.

The robust scheme enables data access throughout failure situations while maintaining consistency according to the default redundancy setting (3) from write procedures.

Chunk Assembly and Validation

- After finishing all download actions or recording failures, the program will unite the pieces into a single file. The worker pool utilizes a waiting group (`wg`) to receive all `DownloadChunkTask` results before the client proceeds.
- The program puts together file data following an arrangement of chunks based on their `chunk_index`.
- The program uses SHA-256 hash verification against metadata to check for corruption and tampering issues (`read.go`).
- The program saves the assembled file into the destination path that the user provided.
- The client generates an error notification and user alert in case all download attempts fail, thus preventing the file from being partially created. The system verifies success by displaying a message in the log which states, "File downloaded to [destination]".

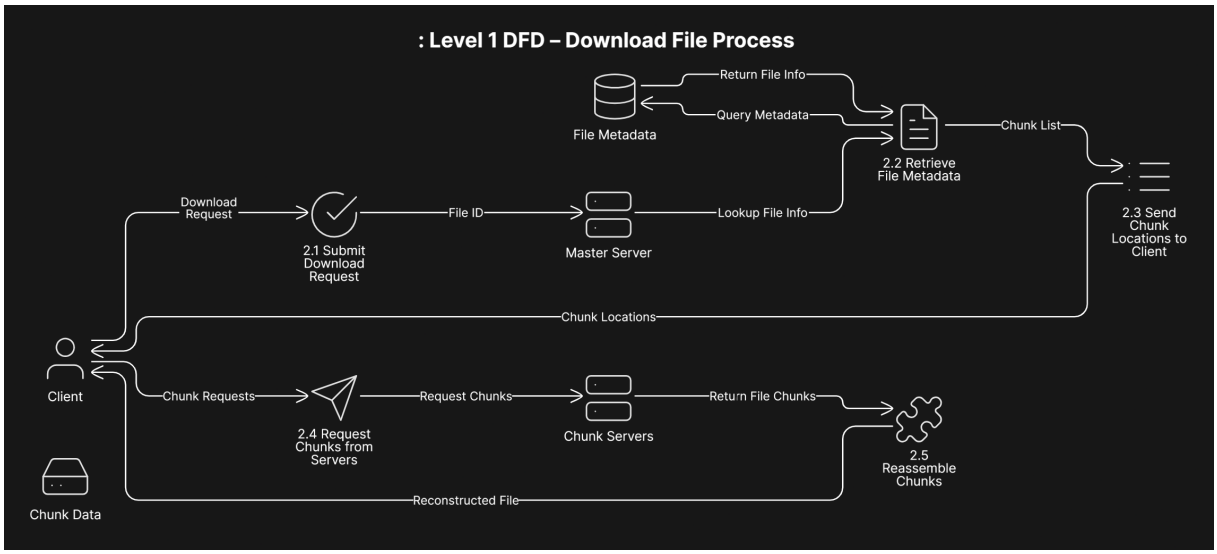


Figure 3.1.3: Download Level 1 DFD

3.1.7 | Delete Workflow in ShardFS

ShardFS delivers a brief process that ensures reliable file removal in its distributed storage infrastructure. File operations begin with the Master Server providing file metadata, after which execution moves to instruct ChunkServers about chunk deletion and concurrency management performed by the client-side worker pool.

1. Metadata Query and Deletion Request The client sends a DeleteFileRequest to the Master Server via the DeleteFile gRPC call (client.go):

```
message DeleteFileRequest {
  string file_name = 1;
  string client_id = 2;
}
```

The Master retrieves the file's metadata (e.g., from MongoDB), identifies the leader and follower ChunkServers for each chunk, and returns a DeleteFileResponse with deletion instructions.

2. Chunk Deletion

The client submits a DeleteFileTask to the worker pool (worker/task.go, worker/pool.go) for each chunk, targeting the leader ChunkServer with a DeleteChunk RPC:

```
rpc DeleteChunk (ChunkRequest) returns (DeleteResponse);

message DeleteResponse {
  bool success = 1;
  string message = 2;
}
```

The leader deletes the chunk locally and coordinates with followers to remove replicas. The worker pool processes tasks concurrently (e.g., 10 workers), with a 10-second timeout.

3. Fault Tolerance and Confirmation

If a deletion fails (e.g., server unavailable), the client retries up to 3 times with exponential backoff. A failure map (delete.go) tracks failed servers, and errors are logged. Upon success from all chunks, the client notifies the user (e.g., "File deleted successfully").

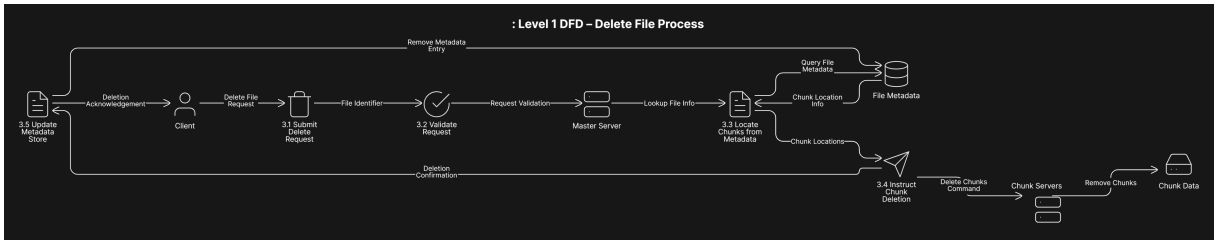


Figure 3.1.4: Delete File Level 1 DFD

3.1.8 | Heartbeat Mechanism & Fault Monitoring

The system depends on heartbeats that enable Master Servers to check the Chunk Servers' operational status. The Master operates a permanent health and resource availability check on Chunk Servers to preserve data replication functions and system performance despite server failures.

Heartbeat Transmission

- The Chunk Servers must transmit heartbeat messages to the Controller Server at a standard 10-second interval.
- The application uses gRPC to enable efficient, structured data transfer between nodes in over-the-network communication.

Heartbeat Payload Data

Each heartbeat message contains a variety of runtime metrics and metadata, including:

Field	Description
Server ID	Unique identifier for the chunk server
Free Disk Space	Available space on the container-mounted volume
CPU Usage	Real-time CPU consumption percentage
Memory Usage	Current RAM utilization on the container
Network I/O	Bandwidth usage over recent intervals
System Load	Load average (for predicting system stress)
Chunks Held	Total number of chunks stored locally
Worker Queue Length	Active tasks in the chunk server's worker pool

Table 3.1.1: Heartbeat Payload Data Fields

By integrating system-level utilities, the Master receives precise performance information from container environments by employing gopsutil and proc filesystem tools and making Docker API calls. Real-time exact assessments of Chunk Server health status are delivered to The Master through this operational system, which also functions in Docker-based environments.

Failure Detection

The Master maintains a last-seen timestamp per Chunk Server. If a server fails to send a heartbeat within a defined timeout (e.g., 30 seconds), it is flagged as inactive or failed. The Master then:

- Removes the chunk server from the active routing table.
- Triggers replication tasks to recreate lost or under-replicated chunks on other healthy servers.
- Logs the failure for audit and admin awareness.

Reliability in Dockerized Environments

- Support from Docker containers enables access to the system's storage health status because Chunk Servers operate as Docker containers. The system derives storage health information by reading statistics from mounted volumes.
- Dynamical cluster rejoining is possible for containers when they restart their operations through the functionality of persistent volume mounting. Metadata losses do not occur.

3.1.9 | Chunk Server Scoring & Load-Aware Prioritization

The Master Server achieves intelligent task allocation through its dynamic scoring system, which applies real-time evaluation criteria to multiple resource metrics for each Chunk Server to enhance system performance. The designed scores from the Controller Server make informed load distribution and chunk allocation decisions while prioritizing servers that remain healthy and underutilized.

Score Calculation Logic

The score for each Chunk Server is calculated using a weighted formula, combining key system metrics to reflect server health and readiness:

Metric	Calculation Formula	Description	Weig
Free Disk Space	$\frac{\text{FreeSpace}}{\text{TotalSpace}}$	storage capacity remaining	0.4
CPU Availability	$\frac{100-\text{CPUUsage}}{100}$	servers with lower CPU usage	0.25
Memory Availability	$\frac{100-\text{MemoryUsage}}{100}$	servers with more free RAM	0.15
Network Efficiency	$\frac{100-\text{NetworkUsage}}{100}$	servers with less I/O congestion	0.10
Load Factor	$\min\left(1.0, \frac{1.0}{\text{Load}}\right)$ (if Load > 0)	Penalizes servers under high system load	0.10

Table 3.1.2: Score Calculation Metrics and Weights

Score-Based Prioritization Using a Max-Heap

A `PriorityQueue` priority queue stores all scores in a max-heap format, enabling constant-time access to the top-performing servers and log-time updates of server metrics. The data access and update operations for server performance metrics reach their results continuously through the max-heap structure.

- The heap accepts new scores through the `heap.Push()` method execution.
- The known servers update their scores directly in the heap through the use of `heap.Fix()`.
- The `heap.Remove()` function removes servers that become inactive from its storage.

Dynamic Re-Evaluation

The solution calculates scores once every heartbeat and then saves them persistently through MongoDB in the `server_status` collection. This ensures:

- The routing system displays the current server status as part of its decision-making process.
- The server stats information maintains its integrity when the Master Server experiences a restart.

The adaptive scoring system operates within the Master Server to achieve system-wide availability, balanced load distribution, and proactive utilization management of Chunk Servers for optimized distributed file systems.

3.1.10 | Worker Pool Architecture

A distributed storage system requires effective concurrent operation management due to high volumes of file read, write, and replication demand. The Worker Pool architecture of Chunk Servers operates to execute and manage asynchronous and parallel I/O-bound tasks.

Motivation

In a typical deployment, Chunk Servers may receive multiple requests simultaneously:

- File uploads (writes)
- File reads
- Chunk replication tasks
- Background repairs or integrity checks

Executing these operations one after the other would result in both wasted CPU resources and delayed performance. A Worker Pool system that employs Go concurrency primitives allows the system to extend capacity when demand requires it.

Benefits

Advantage	Description
Parallelism	Multiple requests can be served concurrently, leveraging multi-core CPUs.
Scalability	As request volume grows, the pool size can be tuned dynamically to match system capacity.
Non-blocking I/O	Network and disk I/O are handled without stalling the main execution thread.
Resource Efficiency	Goroutines are lightweight compared to OS threads, reducing memory overhead.
Isolation & Resilience	Faults in one worker do not affect others. Panic recovery mechanisms can be used to isolate failures.

The Worker Pool model significantly boosts the responsiveness and throughput of each Chunk Server, making the system more scalable and robust under concurrent workloads.

3.1.11 | Deployment with Docker

To achieve platform independence, reproducibility, and ease of scaling, all system components—including the Master Server, Chunk Servers, and Clients—are fully containerized using Docker. This containerized deployment eliminates system-specific dependencies and creates a portable, isolated environment suitable for development, testing, and production. To ensure platform independence, reproducibility, and ease of scaling, all system components—including the Master Server, Chunk Servers, and Clients—are fully containerized using Docker. This containerized deployment abstracts away system-specific dependencies and provides a portable, isolated environment for development, testing, and production use.

Persistent Storage with Docker Volumes

To ensure that chunk data remains intact across container restarts and crashes, Docker Volumes are employed:

- Each Chunk Server is mounted with a dedicated Docker volume for storing chunk files.

- Volumes map to host directories or use Docker-managed storage backends, depending on configuration.

3.1.12 | Network Configuration with Docker Compose

The system is orchestrated using Docker Compose, which simplifies the management of multi-service deployments. It allows defining and launching the entire stack with a single command (`docker-compose up`).

Key Features Enabled by Compose:

Feature	Benefit
Service Discovery	Containers communicate via assigned service names (e.g., `master`, `chunk1`).
Isolated Internal Network	Services interact over a private Docker network, enhancing security.
Replicated Setup	Easily simulate a multi-node environment with multiple chunk servers.
Environment Injection	Supports runtime configuration through environment variables.

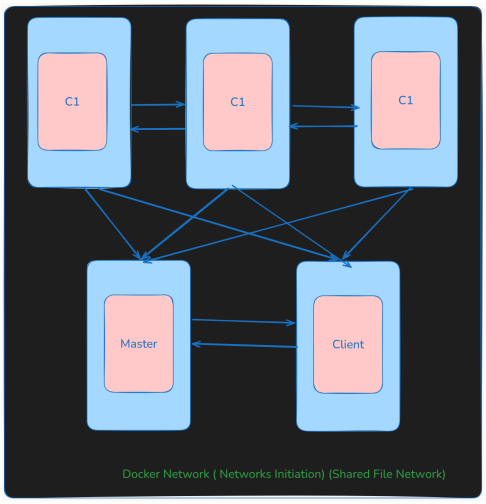


Figure 3.1.5: Network in Docker Containers .

3.1.13 | Sequence Diagram

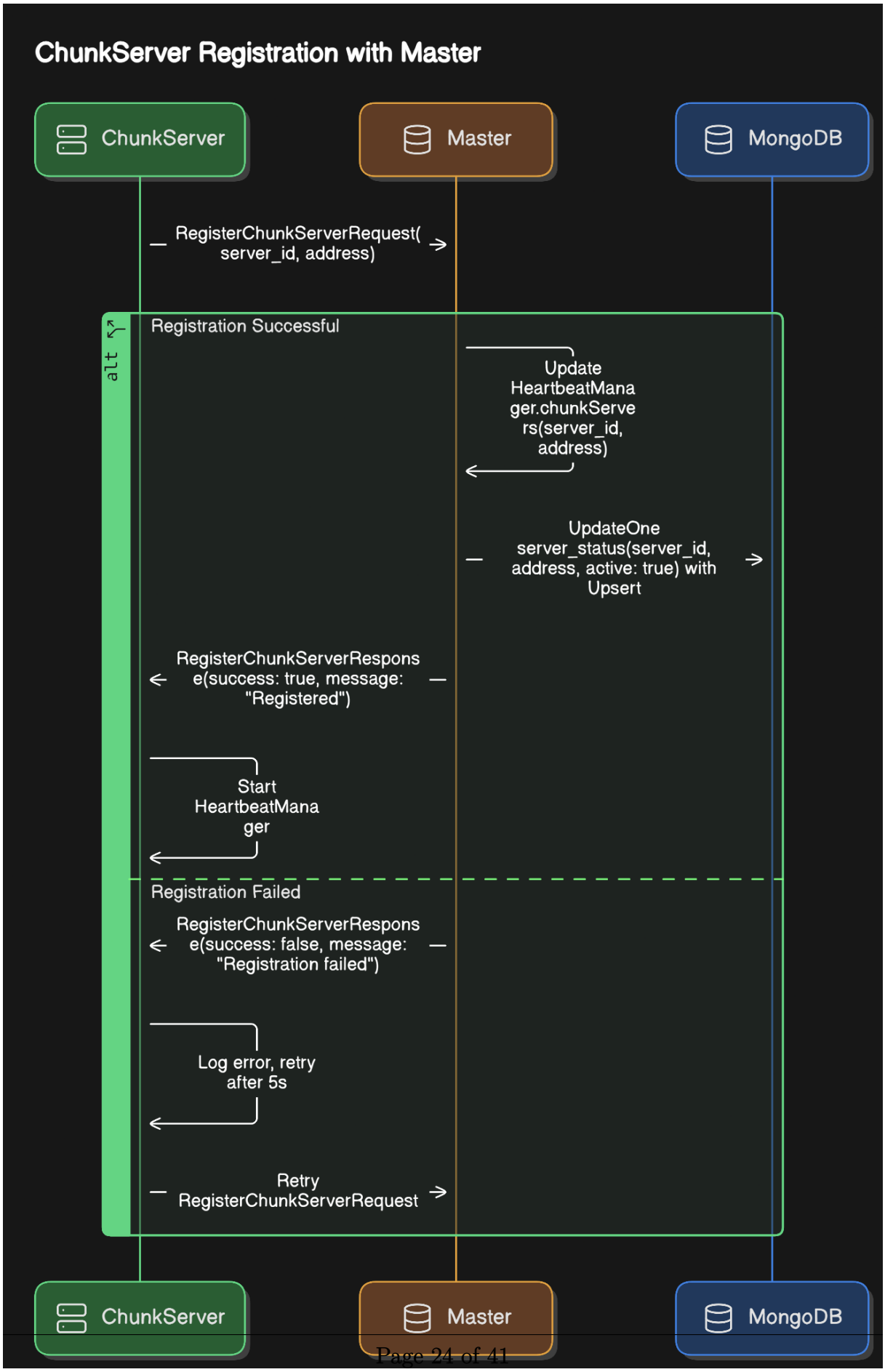


Figure 3.1.6: Sequence Diagram for Chunk Server Registration.

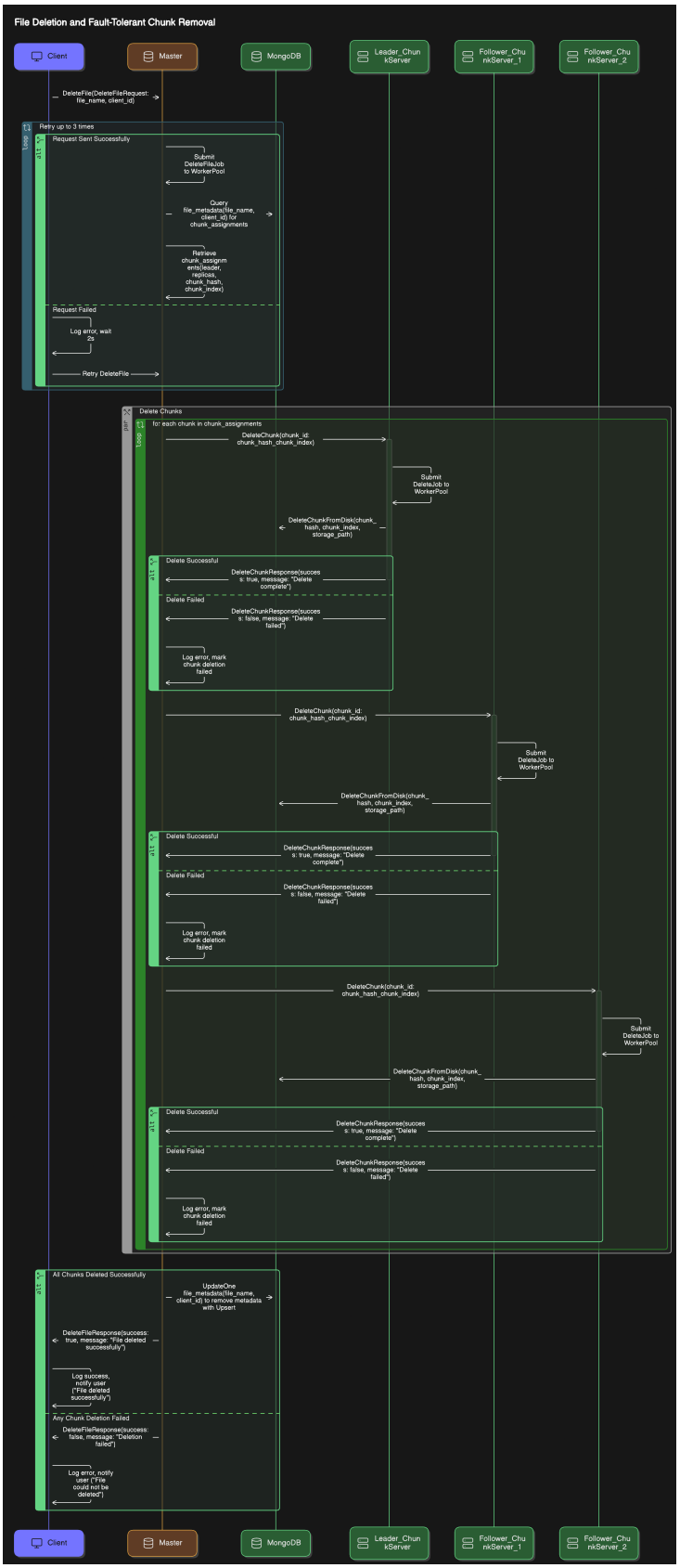


Figure 3.1.7: Sequence Diagram for Delete Workflow.

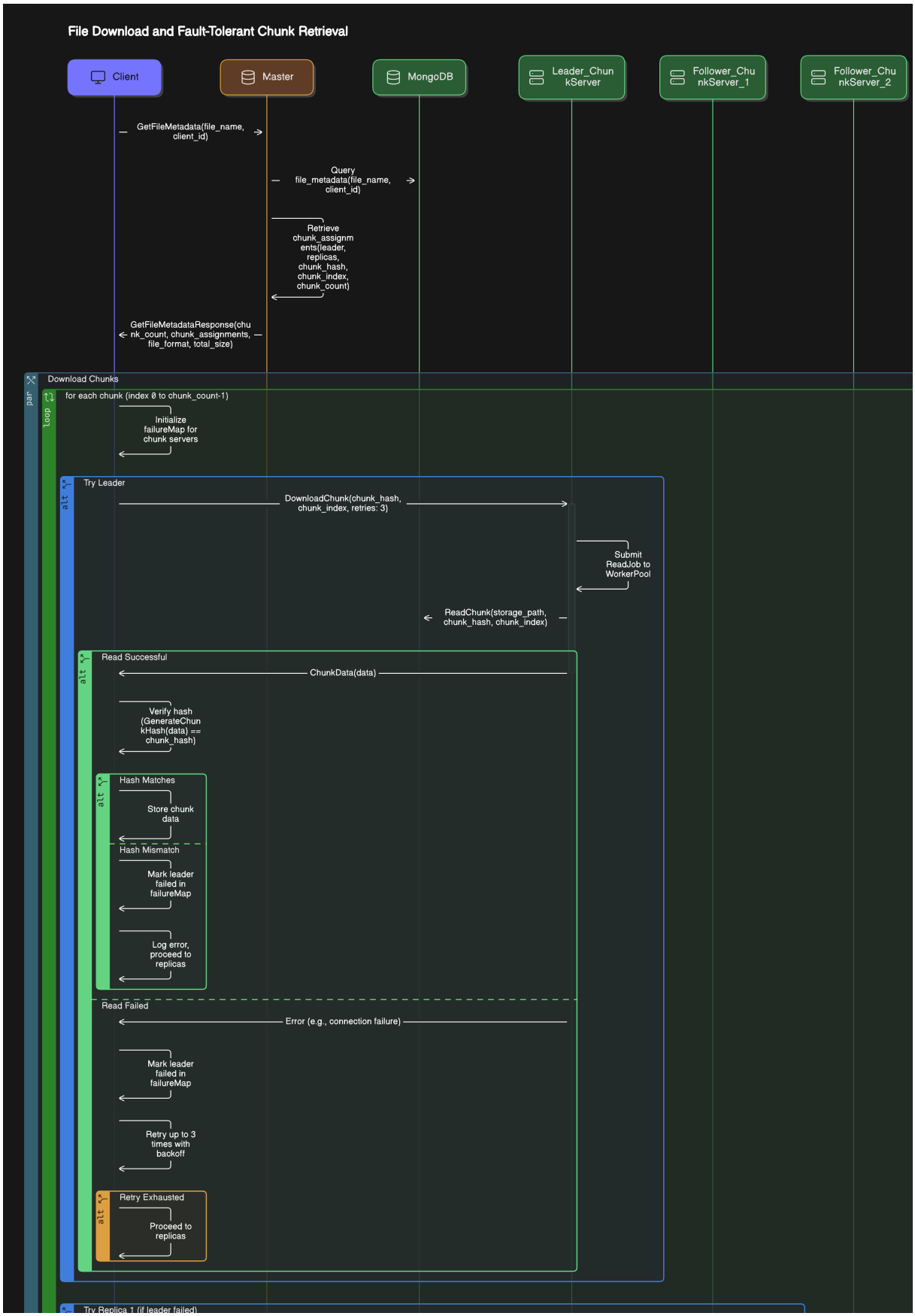


Figure 3.1.8: Download Workflow Part 1

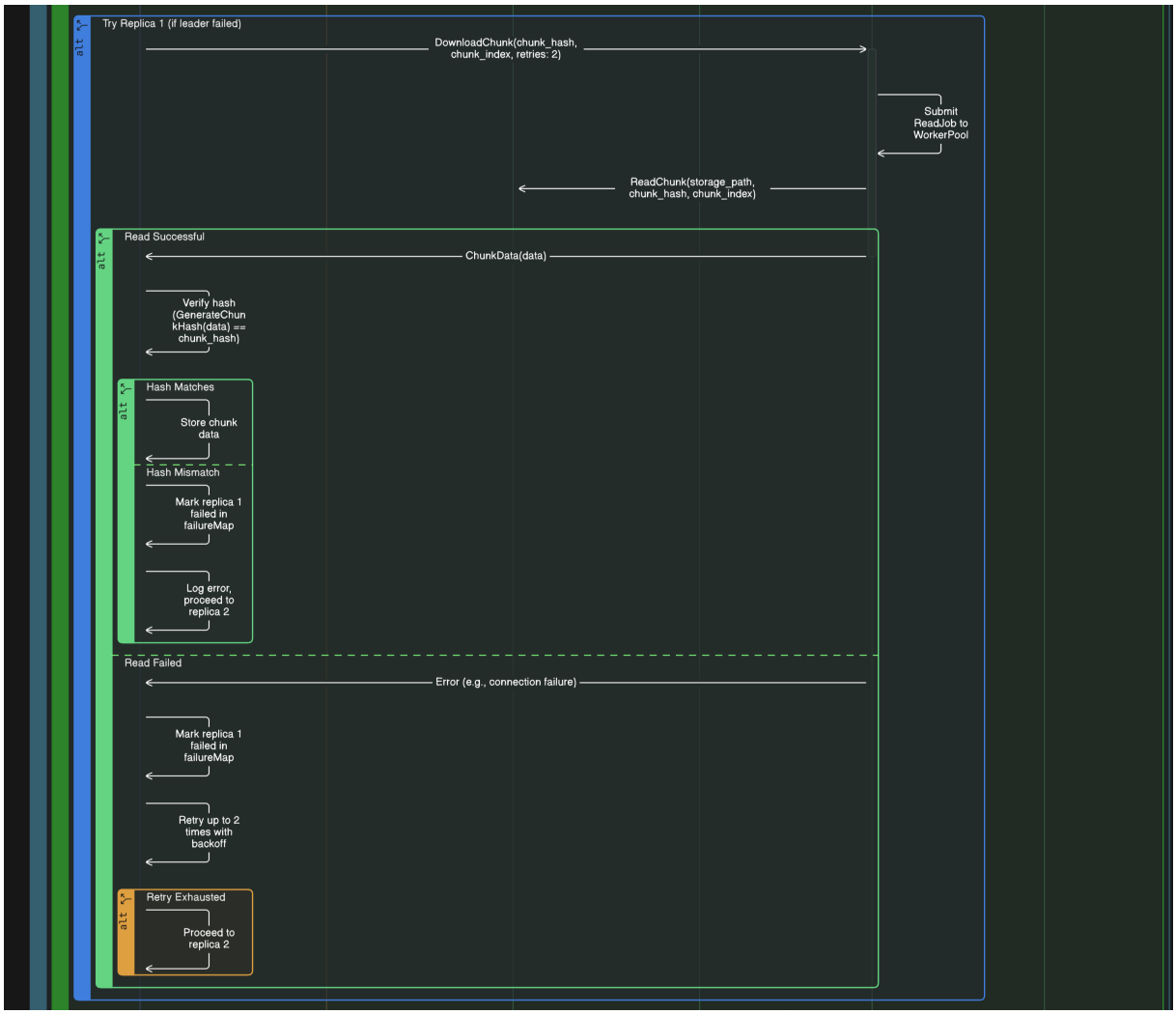


Figure 3.1.9: Download Workflow Part 2

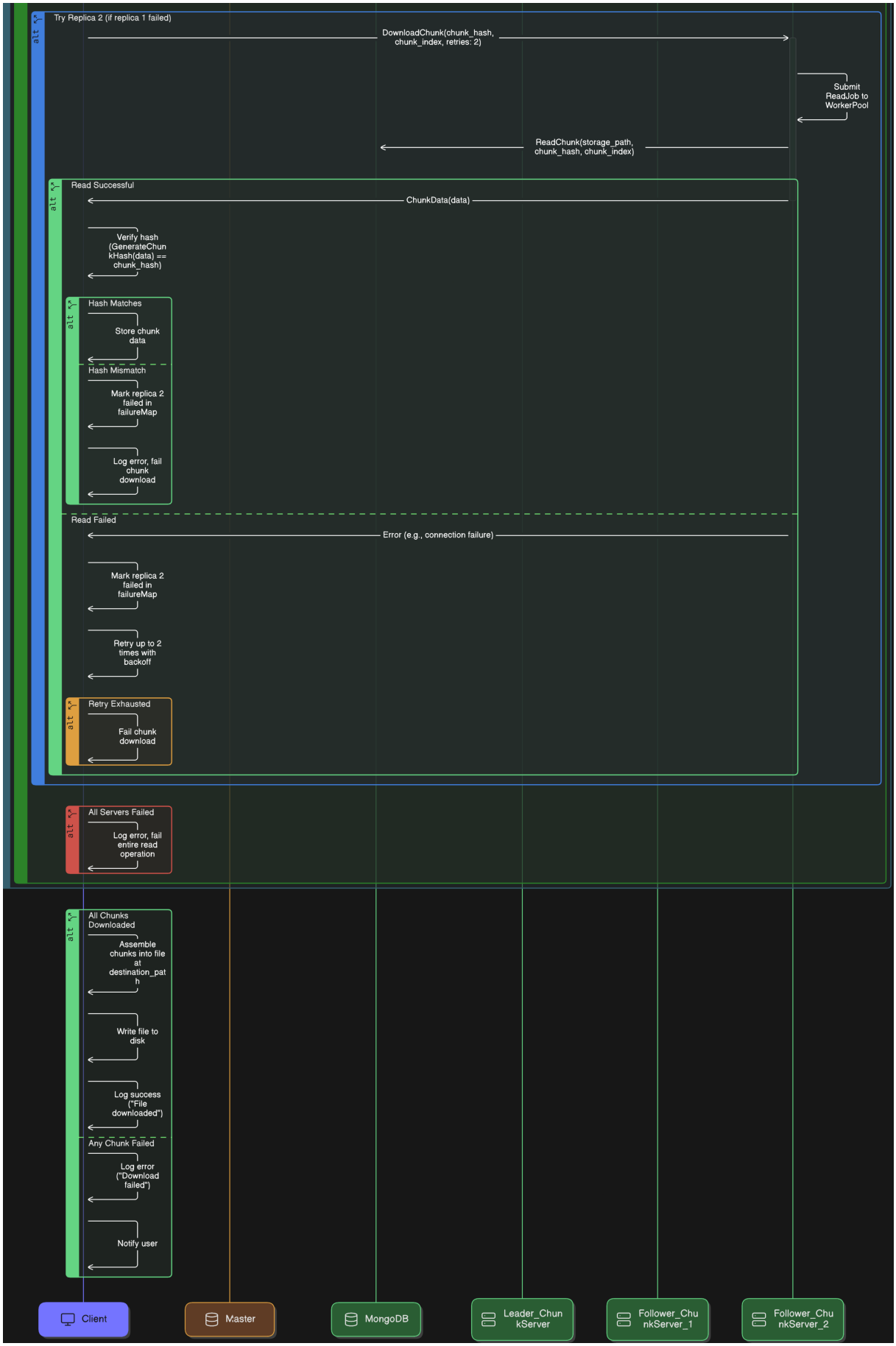


Figure 3.1.10: Download Workflow Part 3

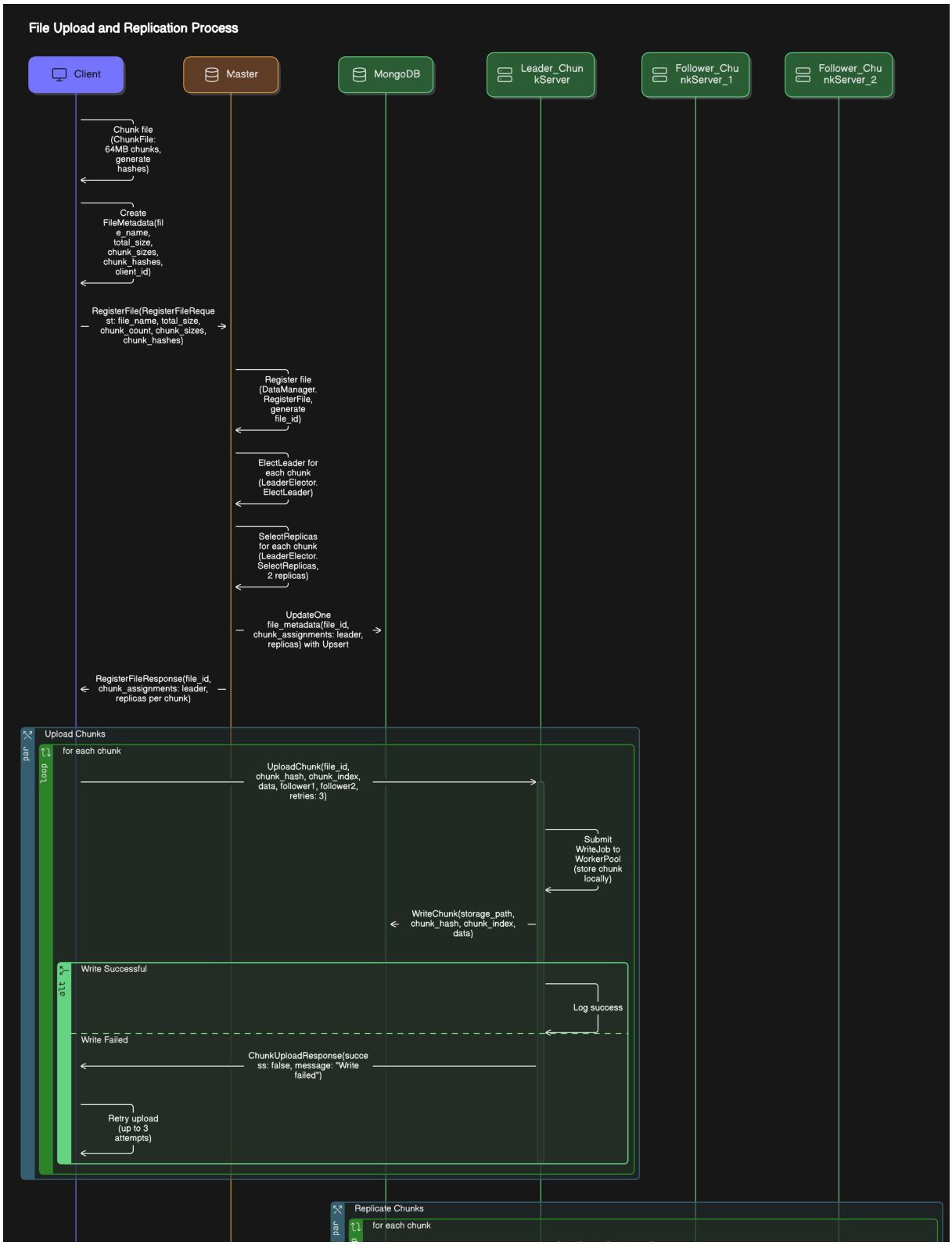


Figure 3.1.11: Upload Workfow Part 1

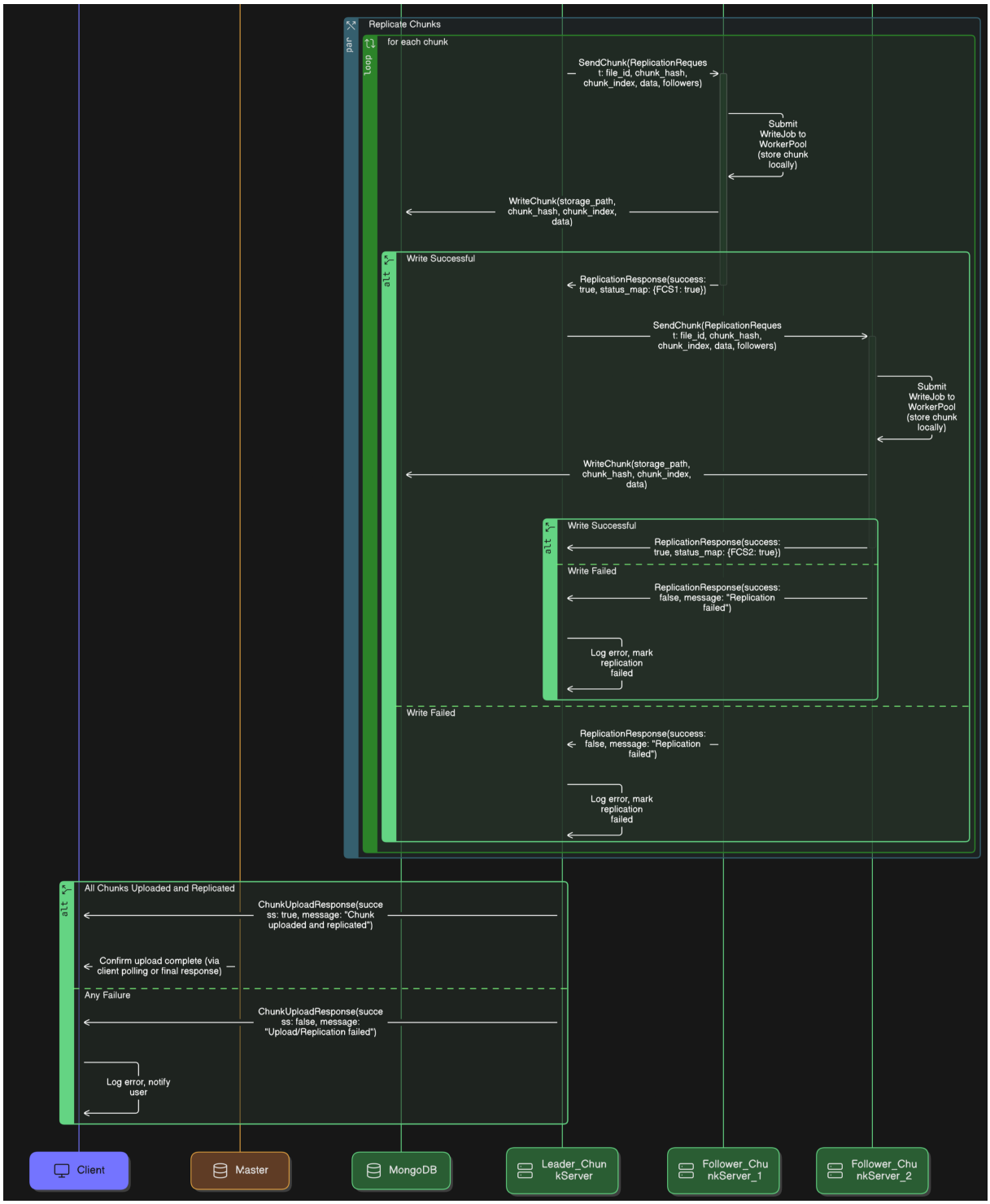


Figure 3.1.12: Upload Workfow Part 2

Chapter 4

Results

Our distributed file system based on GFS principles builds a robust distributed system that accommodates file storage functions, chunking and data replication systems, and failure detection mechanisms on multiple servers. The project validates distributed file storage functionality with current technology implementations of Go, gRPC, and Docker. System development alongside extensive testing proved the effectiveness of its large data transfer management capabilities, metadata preservation, and fault-tolerant heartbeat and data backup functions.

The section presents vital project results alongside team members' acquired capabilities and describes environments that can utilize this system implementation.

4.1 | Project Outcomes

The project successfully delivered a robust, scalable, and fault-tolerant distributed file system modeled after the Google File System (GFS). The system demonstrates high availability, efficient file handling, and modular deployment. Key outcomes include

- **Client-Server Communication System:** The client-server communication system lets users transport and retrieve significant file assets using gRPC protocol and chunked file methodology.
- **Chunk-Based File Division:** The file system splits documents into sections of 64MB for optimized distribution capabilities and better read/write speed alongside simultaneous operation features.
- **Metadata Management by Master Server:** Through the Master Server, clients receive central management of file-to-chunk associations, monitor ChunkServer health patterns, and have leadership roles in initiating replication guidelines.
- **Replication & Fault Tolerance:** The leader-based file replication system creates duplicate chunks across multiple nodes, which protects stored information from disaster events and nodal failures.
- **Heartbeat Monitoring:** Through periodic heartbeat signal collection from each Chunk Server, the Master detects system failures immediately to initiate necessary replication.

- **Dockerized Deployment:** All system elements, including Clients, Master Servers, and Chunk Servers, run inside Docker containers while Docker Compose manages their deployment in an isolated and scalable environment.

4.2 | Learning Outcomes

The project provided a comprehensive learning experience in distributed system design, fault tolerance, and modern infrastructure technologies. The team gained hands-on experience in the following areas:

- **Go (Golang) :** The development of Go skills included effective utilization of goroutines and channels to perform concurrency tasks. The project needed worker pools for replication duties and heartbeat supervision and parallel chunk processing solutions to enhance system performance and responsiveness.
- **Dockerization & Deployment:** The Docker technology spanned across the Master Server and Chunk Servers as well as the Client application. The implementation of Docker provided independent operational environments that Docker Compose could easily coordinate while enabling persistent storage through Docker volumes.
- **Distributed System Architecture:** I developed a decentralized file storage system emulating Google File System features to achieve redundancy, efficiency, and scalability. The data storage system used chunk segmentation for file organization while leader election determined chunk replication and included data redundancy to maintain consistent high availability.
- **gRPC & Protocol Buffers:** The system uses gRPC for high-speed and efficient component-to-component communication between its various system components. The transmission of structured data between Client, Master and Chunk Servers used Protocol Buffers for reliable and secure data serialization.
- **Testing & Fault Simulation:** A test simulation to detect node failures allowed the team to evaluate system resilience by testing data replication and recovery systems. The testing process involved data re-replication and server leader chunk reassignment when servers failed to demonstrate how systems could function smoothly throughout disasters.
- **System Debugging:** Successfully solved problems, including Docker containers' communication difficulties, latency problems, and resource limits in Docker environments.

4.3 | Real World Applications

The distributed file system designed in this project mirrors core principles used in real-world, large-scale infrastructure. Its architecture is applicable to various domains:

- **Cloud Storage Platforms:** Similar to systems like Google Cloud Storage (GCS), Amazon S3, and HDFS, this design supports scalable file storage with replication and fault tolerance for enterprise cloud services.

- **Media & CDN Services:** Content Delivery Networks (CDNs) and streaming platforms, including Netflix or YouTube, use distributed storage systems to access their large video and audio file libraries worldwide quickly.
- **Machine Learning Pipelines:** The model training process of TensorFlow Extended (TFX) alongside distributed training tools relies on analogous architectures to handle large dataset distribution across nodes for fast processing.
- **Scientific Research Computing:** The scientific fields of genomics and climate modeling together with physics simulations deploy high-throughput distributed file systems to handle and store extensive research-oriented data.
- **Enterprise File Systems:** Large organizations with multiple data centers establish internally distributed file storage systems to achieve data availability, redundancy, and fast access to departments nationwide and regionally.
- **Big Data & Analytics:** Hadoop and Apache Spark frameworks need distributed storage to process data in parallel so their operation requires this essential architecture at scale.

4.4 | Project Demonstration

```
master_server_container  | 2025/04/23 01:59:36 📞 Received RegisterChunkServer for chunk_server_3
master_server_container  | 2025/04/23 01:59:36 📩 Submitting RegisterChunkServerJob for chunk_server_3
master_server_container  | 2025/04/23 01:59:36 👷 Worker executing job type: 0
master_server_container  | 2025/04/23 01:59:36 🎉 Registering server chunk_server_3 at chunk_server_container_3
:50054
master_server_container  | 2025/04/23 01:59:36 🔒 Acquired lock for chunkServers check
master_server_container  | 2025/04/23 01:59:36 ✅ Server chunk_server_3 not found, proceeding with registration
master_server_container  | 2025/04/23 01:59:36 🔓 Released lock, updated chunkServers and serverAddresses
master_server_container  | 2025/04/23 01:59:36 📦 Preparing MongoDB update for server_status: chunk_server_3
master_server_container  | 2025/04/23 01:59:36 ✅ Updated server_status for chunk_server_3, modified: 0
master_server_container  | 2025/04/23 01:59:36 ✅ Registered server chunk_server_3, total servers: 3
master_server_container  | 2025/04/23 01:59:36 ✅ RegisterChunkServer completed for chunk_server_3
```

Figure 4.4.1: Chunk Server Registration Log

```
chunk_server_container_1  | 2025/04/23 02:01:18 💓 [02:01:18] Heartbeat from chunk_server_1 | CPU: 0.50% | Memo
ry: 9.83% | Free: 950166 MB / Total: 1031018 MB | Load: 0.21 | Chunks Stored: 0
```

Figure 4.4.2: Chunk Server Heartbeat Log

```
master_server_container  | 2025/04/23 01:59:38 📊 Updating score for chunk_server_1, initial pq size: 0
master_server_container  | 2025/04/23 01:59:38 📊 Pushing new server chunk_server_1 to pq
master_server_container  | 2025/04/23 01:59:38 📊 Pushed to heap, new pq size: 1
master_server_container  | 2025/04/23 01:59:38 💓 Heartbeat [chunk_server_1]: CPU=0.50%, Mem=9.08%, Free=950166
B, Total=0B, Load=0.36, Net=9.70KB/s, Chunks=0, Score=+Inf
```

Figure 4.4.3: Heartbeat Log to Master Server

```
> write /data/reduce4.pdf
[write.go][DEBUG] Checking file: /data/reduce4.pdf
2025/04/23 02:02:16 [write.go][DEBUG] MASTER_ADDRESS env var detected: master_server_container:50052
2025/04/23 02:02:16 [write.go][INFO] Master address: master_server_container:50052
2025/04/23 02:02:16 [write.go][DEBUG] Initializing DFS client
Connecting to master server at master_server_container:50052
2025/04/23 02:02:16 [write.go][DEBUG] Chunking file: /data/reduce4.pdf
2025/04/23 02:02:16 [write.go][DEBUG] Total chunks created: 1
2025/04/23 02:02:16 [write.go][DEBUG] Chunk hash: ae84cc48ff0005c5a79e095039e54e91dce5c116f65746e1593e819d21830
ce9, size: 190711
2025/04/23 02:02:16 [write.go][DEBUG] Total file size: 190711 bytes
2025/04/23 02:02:16 [write.go][DEBUG] Generated file metadata
2025/04/23 02:02:16 [write.go][DEBUG] Registering file with master
Registering file /data/reduce4.pdf with master
2025/04/23 02:02:16 📦 [write.go] File registered: FileID=file_gfs-client_/data/reduce4.pdf_1745373736
2025/04/23 02:02:16 ⏳ [write.go] Preparing to upload chunk 0
2025/04/23 02:02:16 🎯 [write.go] Leader server for chunk 0: chunk_server_container_1:50051
2025/04/23 02:02:16 🔁 [write.go] Replication targets for chunk 0: chunk_server_container_2:50053, chunk_server
_container_3:50054
2025/04/23 02:02:16 📋 [write.go] Chunk hash for chunk 0: ae84cc48ff0005c5a79e095039e54e91dce5c116f65746e1593e8
19d21830ce9
2025/04/23 02:02:16 📦 [write.go] Chunk index for chunk 0: 0
2025/04/23 02:02:16 🎉 [write.go] File write completed successfully
2025/04/23 02:02:16 🔍 Starting upload for chunk ae84cc48ff0005c5a79e095039e54e91dce5c116f65746e1593e819d21830c
e9 to chunk_server_container_1:50051 with up to 3 retries
2025/04/23 02:02:16 🔁 Attempt 1/3 to upload chunk ae84cc48ff0005c5a79e095039e54e91dce5c116f65746e1593e819d2183
0ce9 to chunk_server_container_1:50051
2025/04/23 02:02:16 ✅ Chunk ae84cc48ff0005c5a79e095039e54e91dce5c116f65746e1593e819d21830ce9 uploaded successf
ully to chunk_server_container_1:50051
```

Figure 4.4.4: Write Command Client Side

```
chunk_server_container_1  | 2025/04/23 02:02:16 🚀 Starting UploadChunk stream handler...
chunk_server_container_1  | 2025/04/23 02:02:16 📡 Receiving data for chunk (hash: ae84cc48ff0005c5a79e095039e5
4e91dce5c116f65746e1593e819d21830ce9, index: 0), file 'file_gfs-client_/data/reduce4.pdf_1745373736'...
chunk_server_container_1  | 2025/04/23 02:02:16 📩 Received complete data stream for chunk (hash: ae84cc48ff000
5c5a79e095039e54e91dce5c116f65746e1593e819d21830ce9, index: 0) of file 'file_gfs-client_/data/reduce4.pdf_17453
73736' at leader 'chunk_server_container_1:50051'
chunk_server_container_1  | 2025/04/23 02:02:16 💾 Submitting write job for chunk (hash: ae84cc48ff0005c5a79e09
5039e54e91dce5c116f65746e1593e819d21830ce9, index: 0) to worker pool
chunk_server_container_1  | 2025/04/23 02:02:16 📨 [Pool] Job submitted: 0 for chunk '%!s(MISSING)'
chunk_server_container_1  | 2025/04/23 02:02:16 👷 [Worker 4] Processing job: 0 for chunk '%!s(MISSING)'
chunk_server_container_1  | 2025/04/23 02:02:16 ✅ [Worker 4] Chunk (Hash: 'ae84cc48ff0005c5a79e095039e54e91dce
5c116f65746e1593e819d21830ce9', Index: 0) written successfully
chunk_server_container_1  | 2025/04/23 02:02:16 ✅ Successfully stored chunk (hash: ae84cc48ff0005c5a79e095039e
54e91dce5c116f65746e1593e819d21830ce9, index: 0) at leader 'chunk_server_container_1:50051'
```

Figure 4.4.5: Storing Job (Leader Server)

```
chunk_server_container_1  | 2025/04/23 02:02:16 🚀 [StartReplication] Starting replication at index 0 for chunk
 (hash: ae84cc48ff0005c5a79e095039e54e91dce5c116f65746e1593e819d21830ce9, index: 0, file: '')
chunk_server_container_1  | 2025/04/23 02:02:16 📡 [StartReplication] Attempting to connect to follower 'chunk_
server_container_2:50053' (index 0)
chunk_server_container_1  | 2025/04/23 02:02:16 📩 [StartReplication] Sending chunk (hash: ae84cc48ff0005c5a79e
095039e54e91dce5c116f65746e1593e819d21830ce9, index: 0) to follower 'chunk_server_container_2:50053'
chunk_server_container_1  | 2025/04/23 02:02:16 📬 [StartReplication] Received response from follower 'chunk_se
rver_container_2:50053': success=true, message='Replication successful'
chunk_server_container_1  | 2025/04/23 02:02:16 🔌 [StartReplication] Closed connection to follower 'chunk_serv
er_container_2:50053'
chunk_server_container_1  | 2025/04/23 02:02:16 ✅ [Worker 1] Replication successful for chunk (hash: ae84cc48f
f0005c5a79e095039e54e91dce5c116f65746e1593e819d21830ce9, index: 0)
chunk_server_container_1  | 2025/04/23 02:02:16 ✅ Replication completed for chunk (hash: ae84cc48ff0005c5a79e0
95039e54e91dce5c116f65746e1593e819d21830ce9, index: 0)
chunk_server_container_1  | 2025/04/23 02:02:16 📦 UploadChunk process complete for chunk (hash: ae84cc48ff0005
c5a79e095039e54e91dce5c116f65746e1593e819d21830ce9, index: 0) of file 'file_gfs-client_/data/reduce4.pdf_174537
3736'
```

Figure 4.4.6: Replication Log to Replica Servers

```
master_server_container  | 2025/04/23 02:02:16 🔍 Iteration 0: ServerID=chunk_server_1, Score=+Inf, FreeSpace=9
50166
master_server_container  | 2025/04/23 02:02:16 ✅ Evaluating chunk_server_1: score=+Inf, freeSpace=996321263616
 bytes
master_server_container  | 2025/04/23 02:02:16 ✅ Selected leader: chunk_server_1, score=+Inf, freeSpace=996321
263616 bytes
master_server_container  | 2025/04/23 02:02:16 📦 Leader chunk_server_1 for chunk 0, address=chunk_server_conta
iner_1:50051
master_server_container  | 2025/04/23 02:02:16 ✅ Server chunk_server_1 can store chunk 0: size=190711, remaini
ng=996321072905
master_server_container  | 2025/04/23 02:02:16 ✅ Server chunk_server_1 can handle 1 chunks
```

Figure 4.4.7: Leader Server Election

```
master_server_container  | 2025/04/23 02:02:16 ⚠️Skipping chunk_server_1: is leader
master_server_container  | 2025/04/23 02:02:16 ✅ Candidate chunk_server_2: score=+Inf, freeSpace=950166 bytes
master_server_container  | 2025/04/23 02:02:16 ✅ Candidate chunk_server_3: score=+Inf, freeSpace=950166 bytes
master_server_container  | 2025/04/23 02:02:16 📊 Sorting 2 candidates by score
master_server_container  | 2025/04/23 02:02:16 ✅ Selected replica: chunk_server_2, score=+Inf, freeSpace=95016
6 bytes
master_server_container  | 2025/04/23 02:02:16 ✅ Selected replica: chunk_server_3, score=+Inf, freeSpace=95016
6 bytes
master_server_container  | 2025/04/23 02:02:16 🔓 Released lock for replica selection
master_server_container  | 2025/04/23 02:02:16 ✅ Assigned chunk 0 to leader chunk_server_1, replicas [chunk_se
rver_2 chunk_server_3]
master_server_container  | 2025/04/23 02:02:16 🎉 Assigned 1 chunks for file_gfs-client_/data/reduce4.pdf_17453
73736
master_server_container  | 2025/04/23 02:02:16 📦 Storing file metadata in MongoDB for file_gfs-client_/data/re
duce4.pdf_1745373736
```

Figure 4.4.8: Replica Selection and Storage in MongoDB

```
> read /data/reduce4.pdf /data/reduce4.pdf
Connecting to master server at master_server_container:50052
2025/04/23 02:08:26 [read.go][DEBUG] Initializing DFS client
2025/04/23 02:08:26 📞 Connecting to Master Server at master_server_container:50052 for metadata of /data/reduc
e4.pdf
2025/04/23 02:08:26 Calling Get File Metadata of Client
2025/04/23 02:08:26 📩 Received File Metadata for: /data/reduce4.pdf
2025/04/23 02:08:26 👤 Client ID: gfs-client
2025/04/23 02:08:26 📝 Format:
2025/04/23 02:08:26 📦 Chunk Count: 1
2025/04/23 02:08:26 📐 Total Size: 190711 bytes
2025/04/23 02:08:26 📦 [Chunk 0] Assignment Info:
2025/04/23 02:08:26      ├─ Chunk Index   : 0
2025/04/23 02:08:26      ├─ Chunk Hash    : ae84cc48ff0005c5a79e095039e54e91dce5c116f65746e1593e819d21830ce9
2025/04/23 02:08:26      ├─ Leader Server : chunk_server_container_1:50051
2025/04/23 02:08:26      └─ Replicas      : [chunk_server_container_2:50053 chunk_server_container_3:50054]
2025/04/23 02:08:26 📡 [Chunk 0] Attempting to download from leader: chunk_server_container_1:50051
2025/04/23 02:08:26 🚀 [DownloadTask] Starting chunk download | ChunkHash: ae84cc48ff0005c5a79e095039e54e91dce5
c116f65746e1593e819d21830ce9 | ChunkIndex: 0 | Server: chunk_server_container_1:50051 | Retries: 1
2025/04/23 02:08:26 ae84cc48ff0005c5a79e095039e54e91dce5c116f65746e1593e819d21830ce9_0
2025/04/23 02:08:26 ✅ [DownloadTask] Successfully downloaded chunk | ChunkHash: ae84cc48ff0005c5a79e095039e54e
91dce5c116f65746e1593e819d21830ce9 | ChunkIndex: 0 | Bytes: 190711
2025/04/23 02:08:26 📬 [DownloadTask] Result sent to channel for chunk | ChunkHash: ae84cc48ff0005c5a79e095039e
54e91dce5c116f65746e1593e819d21830ce9 | ChunkIndex: 0
2025/04/23 02:08:26 ✅ [Chunk 0] Successfully downloaded from leader: chunk_server_container_1:50051 | Size: 19
0711 bytes
🎉 File successfully downloaded to /data/reduce4.pdf
```

Figure 4.4.9: Read Log Client Side.

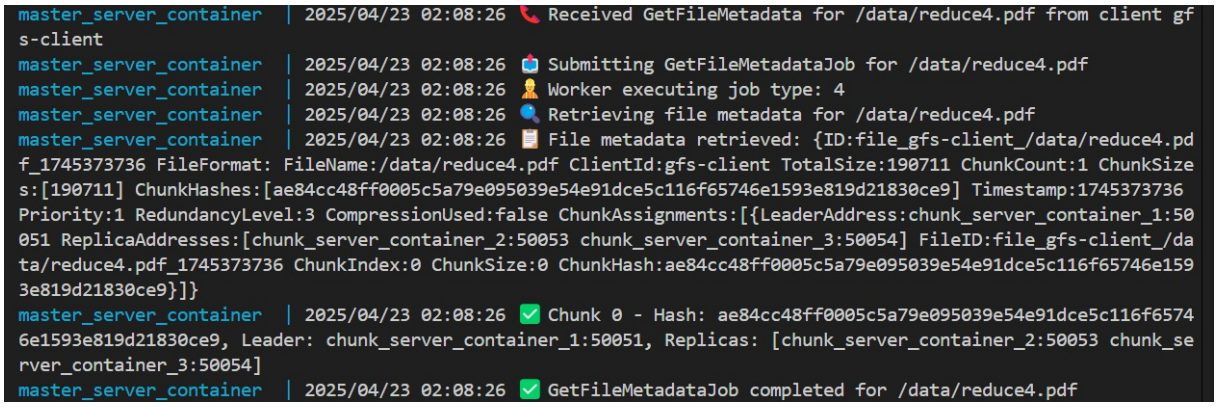


Figure 4.4.10: Read Master Log.

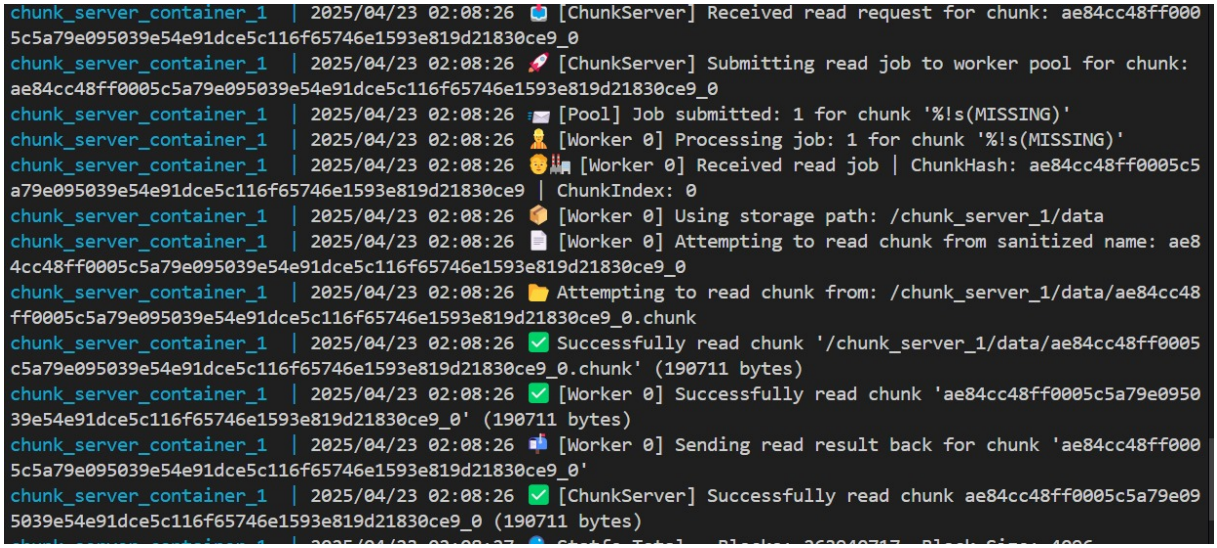


Figure 4.4.11: Leader Server Read Log

Chapter 5

Conclusion

The project demonstrates the realization of a distributed file system that derives its features from the Google File System (GFS) framework. The system operates with present-day technologies Go, gRPC, and Docker implementation to execute file storage operations effectively while maintaining a performance boost with scalability and fault tolerance.

A system architecture based on a Client, Master Server, and Chunk Servers connects with defined communication rules and replication strategies. The system delivers exceptional performance in distributed operations and node failure scenarios because it includes features such as 64MB chunking and leader-based replication paired with heartbeat monitoring and Docker-based deployment.

The project also demonstrates the effective use of concurrency through worker pools, volume-based chunk storage, and real-time health reporting via heartbeat mechanisms. This effort replicates many of the architectural insights of GFS while tailoring the system for modular testing, extensibility, and educational exploration.

The established platform provides essential features to create a production deployment-ready distributed file storage system to support extended capabilities, including database metadata storage, chunk compression ability, and enhanced security implementations. Implementing this system offers practical value and valuable educational experience when designing large-scale systems.

5.1 | Future Developments

- **Multi-Master Architecture (High Availability)** The system requires multiple master servers that use Raft or Paxos for leader election to prevent failures at one point. Distributed systems should use automation to switch masters for achieving high availability.
- **Intelligent Caching Layer** Apply local or distributed caching frameworks (Redis or in-memory) specifically for chunks that experience high frequency of use. The system needs faster read performance alongside lighter chunks server operations and network usage.
- **Monitoring & Dashboard (Admin UI)** Construct an online dashboard that displays system status, chunk distribution information, replication status monitoring,

and network traffic information. The system requires real-time monitoring through Prometheus/Grafana integration for immediate alerting functions.

- **Role-Based Access Control (RBAC)** The application requires authentication and authorization functionality for both client and admin users. The file system requires JWT or OAuth2 authentication technology as a protection measure to support multiple user access.
- **Support for Namespaces and Directory Hierarchies** To mimic a conventional hierarchical file system, include virtual directory trees and file path resolution. Improve file organization and user experience.
- **Version Control & Snapshots** Track file versions and allow rollback to previous states. Useful for debugging, audit trails, or collaborative editing.
- **Support for MIME Types and Non-Text Files** Expand the support for file formats to include binaries, video, and images. Permit chunking and replication strategies to be optimized using MIME.
- **Searchable Metadata and Tagging System** Implement a searchable index for file/chunk metadata. Enable tagging and custom metadata to improve file discovery.
- **Background Replication & Recovery Service** Run asynchronous background tasks to repair under-replicated chunks. Maintain data consistency with low priority tasks to reduce write bottlenecks.
- **Geo-Distributed Chunk Replication** Allow placement of chunk replicas across geographically distributed data centers. Improve availability and access latency for globally distributed clients.

5.2 | Data Flow Diagrams

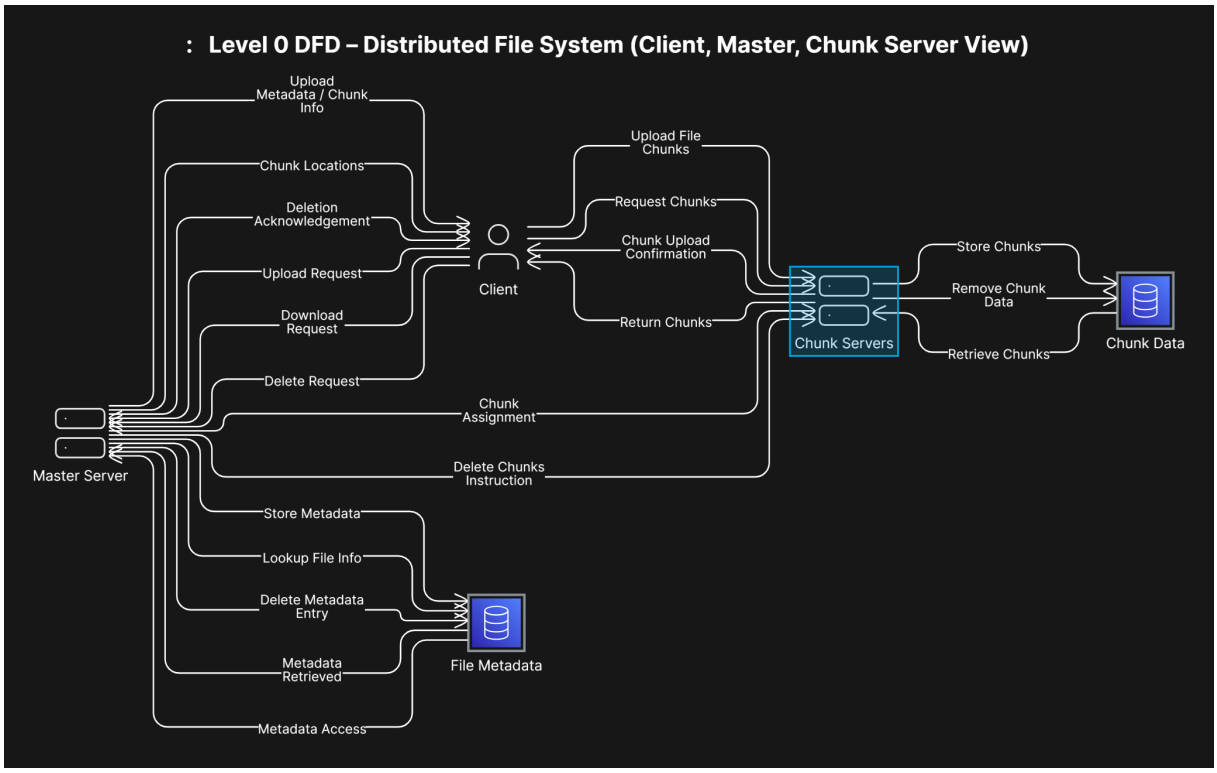


Figure 5.2.1: Level 0 DFD

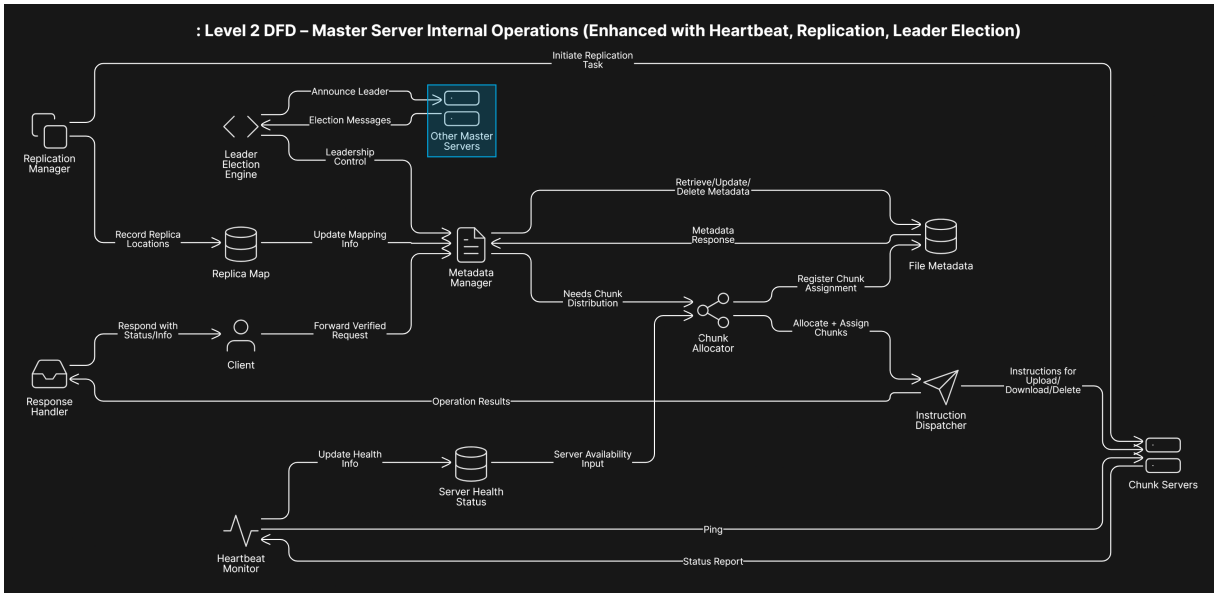


Figure 5.2.2: Master Server Internals

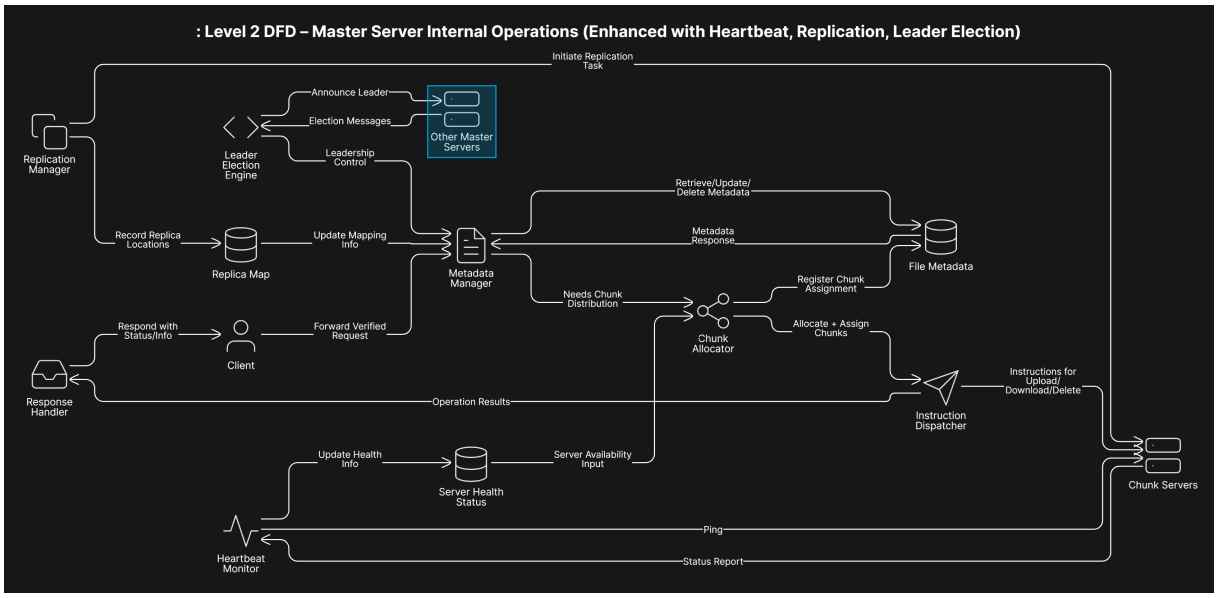


Figure 5.2.3: Chunk Server Internals

Bibliography

[1] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 29–43, 2003.

[2] Apache Software Foundation, "Hadoop distributed file system (hdfs)." https://hadoop.apache.org/, 2023.

[3] A. S. Tanenbaum and M. V. Steen, "Distributed systems: principles and paradigms," 2007.

[4] Golang.org, "The go programming language." https://golang.org/, 2023.

[5] Docker Inc., "Docker documentation." https://docs.docker.com/, 2023.

[6] Protocol Buffers, "Protocol buffers - google's data interchange format." https://developers.google.com/protocol-buffers, 2023.

[7] grpc.io, "grpc - a high-performance, open-source universal rpc framework." https://grpc.io/, 2023.

[8] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes: Lessons learned from three container-management systems over a decade," 2016.

[9] M. Kleppmann, "Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems," 2017.

[10] J. Dean and L. A. Barroso, "The tail at scale," 2013.

[11] D. E. Knuth, *The TEX Book*. Addison-Wesley Professional, 1986.