UNIVERSITY OF PISA

COMPUTER ENGINEERING MASTER DEGREE

Distributed Systems and Middleware Technologies

Distributed Storage System

Professors:

Alessio Bechini

José Corcuera

Group Members:

Kevin Boni

Alice Orlandini

ACADEMIC YEAR 2024/2025

Contents

1 Introduction

A **Distributed Storage System** is a network-based storage solution designed to distribute, store, and replicate data across multiple nodes efficiently. Instead of keeping entire files on a single server, the system **splits files into fixed-size chunks** and **distributes them across multiple slave nodes**, ensuring scalability, fault tolerance, and load balancing.

This project aims to develop a Distributed Storage System composed of four key components: **clients**, a **load balancer** and multiple **master and slave nodes**.

The client interacts with the system to **upload** and **download** files. When a file is uploaded, the load balancer selects a master node using a certain policy. The selected master node is then responsible for splitting the file into chunks, tracking their locations, and managing system-wide **load balancing**. The master checks the load of each slave node and assigns chunks accordingly. Once a chunk is stored on a selected slave node, the master also instructs the node to **replicate the chunk** to another designated slave node, ensuring redundancy and fault tolerance.

For file retrieval, the client sends a request to the load balancer, which forwards it to a master node. The master identifies the slave nodes storing each chunk of the requested file and, for every chunk, **selects the optimal slave node by monitoring their current load**. It then responds to the client with a list of IP addresses, one for each chunk, corresponding to the chosen slave nodes. The client subsequently contacts these slave nodes directly to retrieve and reconstruct the original file.

2 System Architecture

The Distributed Storage System is organized into four main components:

- **Client**: A Java-based graphical application that allows users to upload, download, share, and delete files.
- **Load Balancer**: Implemented with NGINX, it distributes incoming client requests among available master nodes to ensure load balancing.
- **Master Nodes**: Written in Erlang, master nodes handle file chunking, metadata management, user authentication, access control, and coordination of chunk storage and replication. They do not store file data directly, but maintain the global state and metadata in a replicated Mnesia database.
- **Slave Nodes**: Also written in Erlang, slave nodes are responsible for storing and replicating file chunks as instructed by the master. They serve chunk download requests from clients and manage local chunk storage.

Communication between client and server is via HTTP, while master-slave communication uses Erlang IPC. The architecture ensures redundancy, load balancing, and fault tolerance through chunk replication and dynamic load monitoring.

Load Balancer and Master Node Scalability

The system uses an NGINX load balancer to distribute client requests among multiple master nodes. The NGINX configuration uses the `ip_hash` directive to provide sticky sessions, ensuring that requests from the

same client IP are routed to the same master node. This allows the system to scale horizontally by simply adding more master nodes and updating the NGINX configuration, without requiring changes to the client or the rest of the system. Each master node is stateless with respect to file data and relies on a replicated metadata database (Mnesia) for consistency and high availability.

3 File Operations

This section describes the main file operations available to users: upload, download, share, and delete.

3.1 File Upload

- The user selects a file to upload using the client interface.
- The client sends the file to the backend, authenticated with a JWT token.
- The master node splits the file into fixed-size chunks and assigns them to slave nodes based on their current load.
- The master instructs the primary slave to store the chunk and replicate it to a secondary slave for redundancy.
- The master updates the metadata to track chunk locations and replicas.
- The client is notified of the upload result.

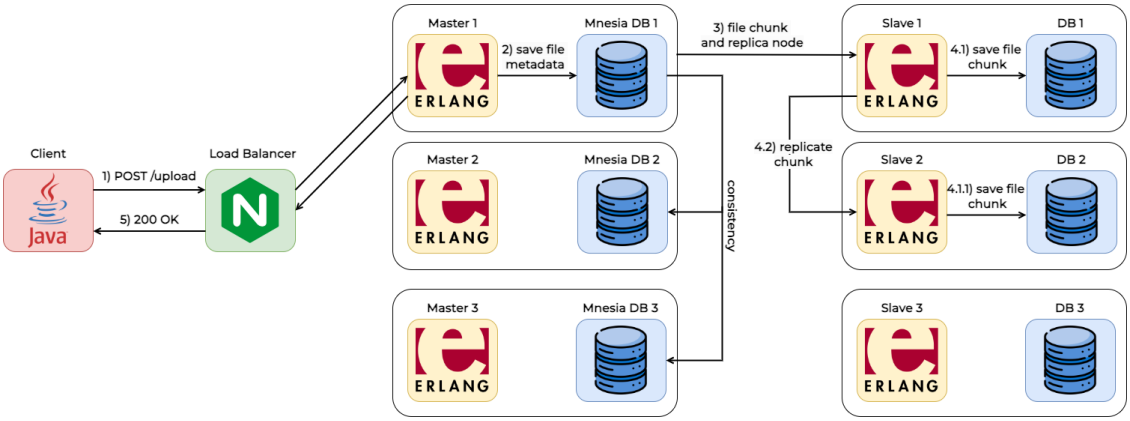

Figure 1: Anatomy of the Upload of a file

3.2 File Download

- The user selects a file to download from the client interface.
- The client requests the file from the backend, authenticated with a JWT token.
- The master node checks user permissions and, for each chunk, determines the most suitable slave node:
  - It queries all candidate slave nodes to retrieve their current load.
  - Each slave reports its number of *pending requests* and *possible requests*.
  - The estimated load is computed as:

$$\text{Total Requests} = \text{Pending} + 0.5 \times \text{Possible}$$

  - The master selects, for each chunk, the slave node with the lowest *Total Requests* to avoid overloading any single node and achieve a fairer distribution.
- The master returns, for each chunk, the IP address of the selected slave node and a short-lived token.
- The client downloads each chunk directly from the corresponding slave node using the provided token.
- The client reassembles the original file from the downloaded chunks.
- The client is notified of the download result.

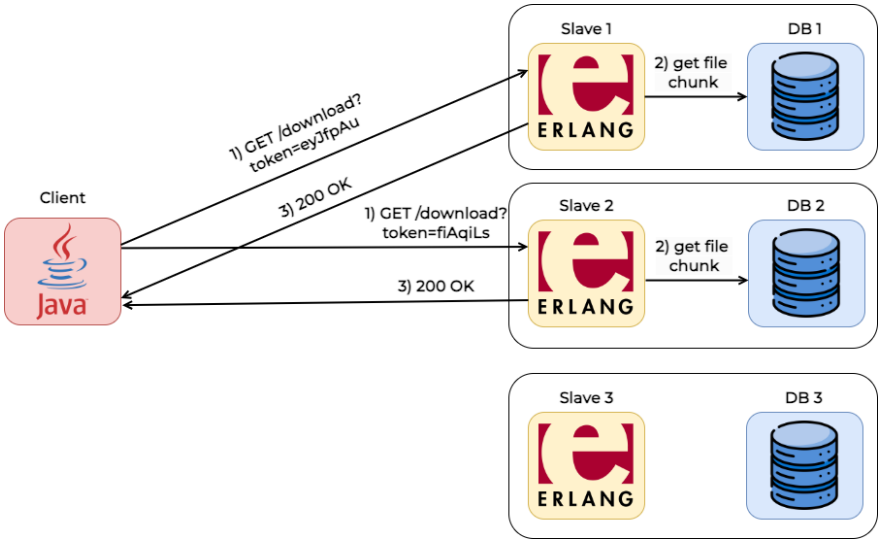

Figure 2: Anatomy of the Download of a file

3.3 File Sharing

- The user selects a file to share with another user through the client interface.
- The client sends the sharing request to the backend, authenticated with a JWT token.
- The master node updates the file metadata to grant the recipient full access to the file.
- The file is immediately visible in the recipient's file list, as if it were their own.

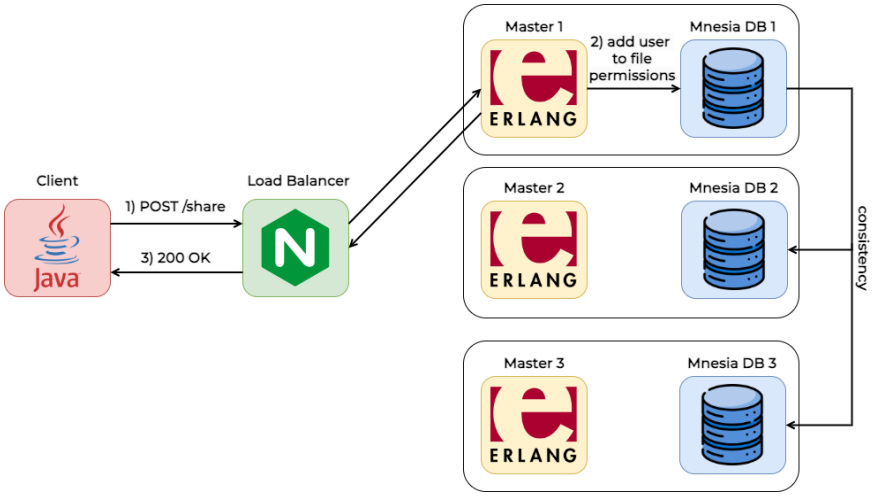

Figure 3: Anatomy of the Share of a file

3.4 File Deletion

- The user selects a file to delete from the file management interface.
- The client sends the deletion request to the backend, authenticated with a JWT token.
- The master node checks user permissions and retrieves the metadata for the selected file.
- If the user is the last owner of the file, the master instructs all slave nodes storing the file's chunks to delete them and removes the corresponding metadata.
- If the file is still shared with other users, only the metadata related to the requesting user's access is removed.
- The file becomes inaccessible to the user who requested the deletion.

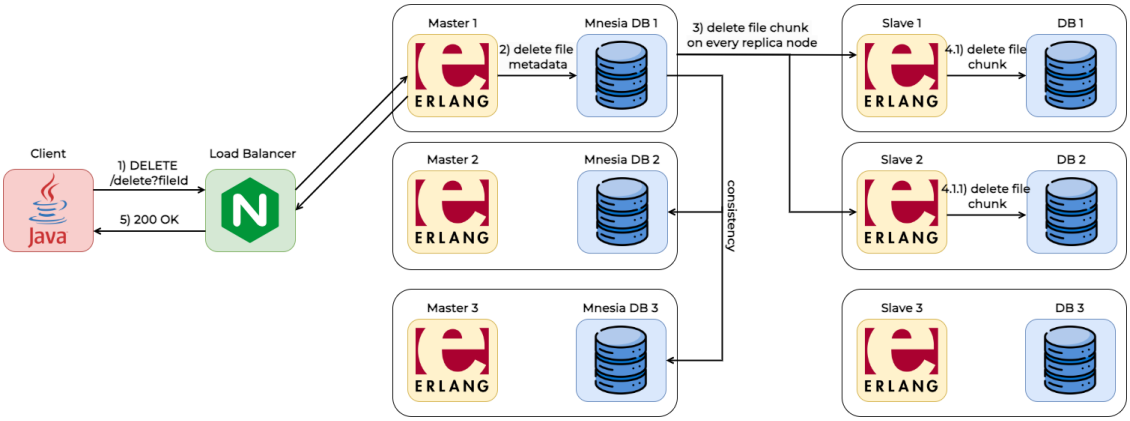

Figure 4: Anatomy of the Delete of a file

4 Requirements

4.1 Functional Requirements

The functional requirements are:

1. **FR01 – File Upload Handling**: The client must be able to send a file upload request to the load balancer node. The corresponding master will then process the request and prepare the file for distributed storage.
2. **FR02 – File Chunking and Distribution**: When a file is uploaded, the master node must split it into fixed-size chunks and determine the best slave nodes to store them, based on their current load.
3. **FR03 – Load-Aware Chunk Assignment**: The master node must track the workload of each slave node and ensure that chunks are assigned to the least-loaded nodes to achieve efficient resource distribution.
4. **FR04 – Chunk Replication Management**: After assigning a chunk to a primary slave node, the master must also specify a secondary slave node for replication. The primary slave is responsible for forwarding the chunk to its designated replication node, ensuring redundancy.

5. **FR05 – Metadata Management**: The master must maintain a metadata table tracking which slave nodes store each chunk of a file and the replica. This table is used to manage file retrieval and system consistency.

6. **FR06 – File Download Handling**: When a user wants to download a file, the client first sends a request to the master node asking for the chunks of the specified file. For each chunk, the master responds with the IP address of the selected slave node, a short-lived token, and the `chunkPosition`, indicating which part of the file the token refers to. The client then sends a request to the corresponding slave node including only the token. The slave node must validate the token (e.g., check expiration) and extract from it the necessary information such as the file name and the chunk position, in order to return the correct chunk to the client.

7. **FR08 – Master-Initiated Load Monitoring of Slaves**: When the master node receives a download request, it must query each slave node that stores a replica of the requested chunk to retrieve its current load status. In response, each slave reports two metrics: `pendingRequests`, representing the number of active requests the slave is currently handling or will certainly handle, and `possibleRequests`, representing potential requests that might be assigned based on ongoing selection processes. After evaluating the load of all candidate slaves, the master selects the least loaded one for each chunk and informs all contacted slaves of the outcome by sending either a `chosen` or `not chosen` message. Upon receiving this feedback, slaves must update their `pendingRequests` and `possibleRequests` accordingly.

8. **FR9 – Users Registration and Login**: Users must first register with a username and password, and upon successful login, they are issued a JWT token that is valid for a limited time (e.g., a few hours). The master uses this token to authenticate client upload and download requests.

9. **FR10 – Load Balancer for Master Node Distribution**: The system must include a load balancer component responsible for distributing incoming client requests across multiple master nodes. The load balancer must implement a policy to ensure an even distribution of the workload. It must also detect master node failures and stop forwarding requests to unreachable nodes.

10. **FR11 – File Sharing Between Users**: The system must allow a user to share a file with other users by specifying a list of authorized usernames. This access control list must be stored alongside the file's metadata. When a user requests to download a shared file, the master must verify whether the requester is the owner of the file or is included in the list of authorized users. Only if the user is authorized, the master will proceed to generate and return the download tokens for the requested chunks.

11. **FR12 - File Deletion Management**: The system must allow a client to request the deletion of a file. Upon receiving the request, the master must remove the file's metadata from its records and instruct all relevant slave nodes to delete the associated chunks from their local storage. After the deletion, neither the original uploader nor any other clients with whom the file was shared should be able to access or retrieve the file from the distributed storage system.

4.2 Non-Functional Requirements

The non-functional requirements are:

1. **NFR01 – Communication Protocols**: The system must use HTTP for all communications between clients and slave nodes. However, communication between the master and slave nodes must rely on Erlang's inter-process communication (IPC).
2. **NFR02 – System Availability Despite Slave Failures**: The system must remain available and operational even if one or multiple slave nodes go offline. The failure of a single slave must not prevent file retrieval as long as redundant copies of the necessary file chunks exist in other nodes.
3. **NFR03 – Chunk-Based File Storage**: Each slave node must store and manage only parts (chunks) of a file, rather than the entire file.
4. **NFR04 – Concurrency and Communication Handling**: The master and slave nodes, must manage concurrent requests and inter-node communication. Java-based clients must be able to handle multiple simultaneous connections to different slave nodes for file retrieval.