Student File System

Anant Patel Gayatri Priyadarsini Hari Hara Sudhan S

File storage systems like OneDrive, the one drive mountable to the Windows and MACOS systems. Similar file systems can be developed for Linux-based OS. With this in mind, the focus of our project would be to provide access to IITGn students to a **linux-based distributed** file system.

User Guide

Platform Requirements

1. Linux Operating System (Ubuntu >=20.04)
2. GoLang >= 1.18.1
3. Go-Fuse = 2.3.0
4. MongoDB
5. Linux libfuse

Deployment

Client

```
# downloading packages
cd client
go mod download github.com/hanwen/go-fuse/v2
go get github.com/hanwen/go-fuse/v2/
go build main.go rpc.go

# running the client
# ./main <master-addr> <mount-dir> <root-parent> <root-name>
./main 10.7.50.133 /mnt/sfs /home/test1 test
```

Master

Install MongoDB from the given link.

```
# building master servers
cd master
go build mserver.go master.go rpc.go structs.go mongo.go
client-rpc.go datanode-rpc.go
```

Change the `.ini` with the update server addresses and credentials for MongoDB:

```
[rpc]
host = 0.0.0.0
port = 9000

[mongodb]
uri = mongodb://0.0.0.0:27017
username = filedb
password = supersecret
db = filedb
mech = SCRAM-SHA-1

# master server
./mserver
```

Data Node

```
cd datanode
go build datanode-rpc.go datanode.go main.go rpc.go
# ./datanode-rpc <master-ip> <master-port> <data-node-ip>
<data-node-port>
./datanode-rpc 10.7.50.133 9000 10.0.62.251 9000
```

Introduction

A Distributed File system is a file system that is distributed across a network that allows users on the network to access the files from anywhere on the network. The proposed distributed file system would support basic file system operations like CRUD. In addition to that, it provides fault tolerance and recovery features to ensure that the data is always available to the users.

Our architecture would have data nodes running a Linux server and a master node maintaining the file and user metadata. Furthermore, we intend to provide mounting facilities for the file system to be accessible from the Linux system as is.

Features

- CRUD Operations
    - The Distributed File system will support Create, Read, Update, and Delete operations on the files.
- Availability
    - The data stored in the file system would be available to the users even in an event of failure.
- Fault Tolerance and Recovery
    - The files system will be able to recover from hardware failure, and permanent loss of the files would be avoided by replication of the files.
- Access Control and Authentication (Feature not available in the current version)
    - The system will support authentication and also support file access control based on flags.
- Mounting file system
    - Support for accessing file via the existing file system (Ext4)
- Horizontal Scaling
    - Support for the increase in the number of Data Nodes at any point of time. This is needed to accommodate the increasing number of students.
- File Sharing
    - Support file sharing across users.

Technical Details

The File System will have a Master/ Slave architecture. The master node would host all the metadata (file name, replication factor, physical storage path, etc.) related to the files stored in the system. The slave nodes (hereafter referred to as data nodes) would store the files using the existing file system (Linux - Debian).

In order to avoid a single point of failure, there we propose a shadow master node that would replicate the updates done to the master node. In case of a failure of the master node, the shadow master node takes up the position of the master node.

To indicate that the data nodes are alive, they send a heartbeat signal to the master node at a fixed interval. A similar heartbeat signal would be sent by the master node to the shadow master node. All the communications between the data nodes, master node and shadow master node would take place via RPC calls.

For mounting the file system from the data node to the user system, we had multiple options to choose from, we narrowed it down to FUSE. We intend to use the FUSE library to establish the connection between the master node and the user. We intend to show only the current directory and the files/folders in the same to the user. This is all available with the master in the form of metadata. For opening the file from the user system, we may need to download the file from the data nodes instead of the master node.

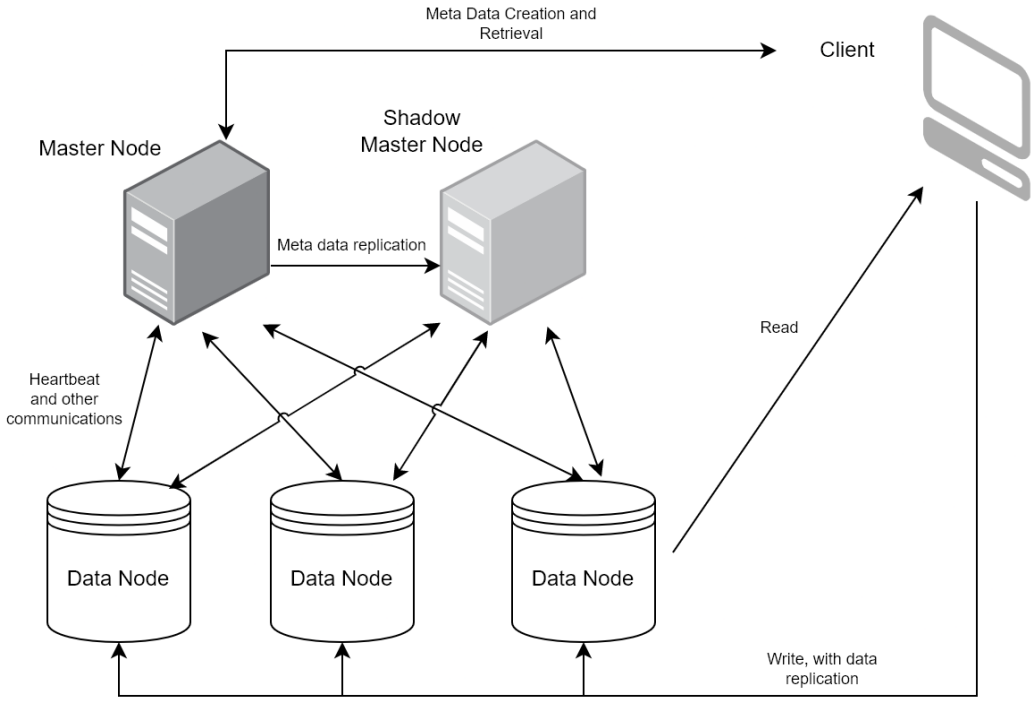


Fig 1: Architecture of proposed file system

1. Broker Architecture

In the Broker Architecture, there is a broker service that acts as an intermediary between different components of the architecture. There are two ways the broker can behave, one is to identify where to forward each request, whereas the other is that components pass messages to the broker with the address specifications of where the request is to be forwarded. We use a combination of both ways to create an **intermediary master node**.

While in traditional broker architectures, there is no connection between the different components (client and data nodes) directly, but here we plan to allow the connection between a data node and client when the client makes a write request. This is to prevent a continuous connection between the master-client and master-data node for when the file is being uploaded.

We allow the user to make write requests through the broker only. This way, we avoid inconsistency issues that may arise from multiple write requests. Furthermore, to prevent the broker from being a single point of failure, we plan to deploy multiple brokers having a replicated MongoDB in consensus.

2. System Components

2.1 Master Node

The master node acts as an intermediary between the actual client and the storage location (data node). The master maintains the log of all the datanodes which are alive using heartbeats. Master additionally ensures that the update operations initiated by the client are all completed by the datanodes. It maintains the logs for the same. In case some datanode is inactive for some time and comes up again, the master sends it the latest changes to be made.

- Functionality
    - RPC
    - Database operations and synchronization with shadow
    - Logging
    - Replication
- Connections Allowed/Required:
    - RPC with client
    - RPC with Shadows
    - MongoDB
    - RPC with Data Nodes

2.2 Data Node

- Functionality
    - Request/Response with client
    - Replication
- Connections
    - RPC with client
    - RPC with master for metadata and operation completion updates.
    - RPC with other datanodes for replication

Replication is done between the datanodes using the information given by the master. The primary datanode in an operation will update the master after the operation is done. Once the master is informed, the primary again sends a request to the master to fetch the replication nodes. The primary node then forwards the requests for replication to these nodes. These nodes, in turn, just inform the master node about the operation being done/completed.

Fault tolerance for the datanode is done using the logs maintained by the master node. Every time a new operation is done, the master node is updated about it, which is then stored as logs with timestamps. When a datanode has been down for a while and gets back up after, it pings the master node as usual. The master then sends it the operations it has to replicate from the other nodes in order to update itself to the latest changes.

2.3 Database

The database used is MongoDB. There exists a MongoDB ReplicaSet for fault tolerance.
The following are the collections in the same:

User collection

- User ID
- Username: string Primary Key
- Password: string
- Last login: Time

File collection

- File ID: ObjectId
- Parent Folder: String
- File Name: string
- Last Modified: Date Time
- Done: bool

Folder collection

- Folder ID: Primary Key
- Parent Folder ID: Foreign Key
- Folder name: string
- Last Modified: Date Time

Data-Server collection

- Server ID
- Server Address

Permission collection Feature not available

- File ID
- User ID

File-access collection

- File ID
- Server ID
- Last Updated

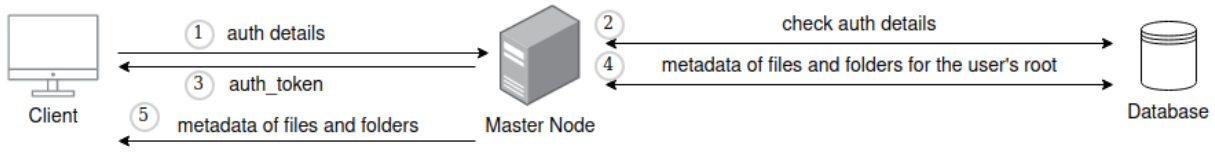
2.4 Client Node

The client nodes are the user interfaces between the distributed file system and the user. The following operations are available at the client side. The associated diagrams depict the required communications between the different components of the system.

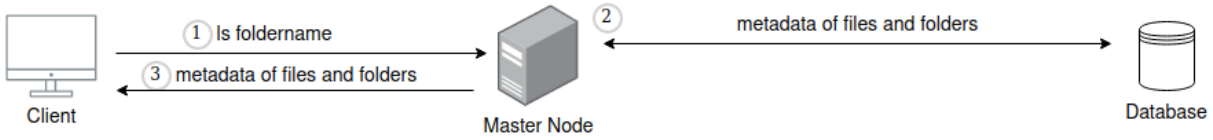
File System in Userspace (FUSE) is a software interface for Unix systems that allows non-privileged users to create their own file system without modifying the kernel code. It works by creating a bridge to the kernel interfaces. Go-Fuse (by hanwen) is an adaptation of the libfuse in Go Language. The go-fuse provides interfaces for ReadDir, Lookup, Read, Write, Open, Getattr, Mkdir, Unlink, Rmdir, Rename, etc.

Client Operations

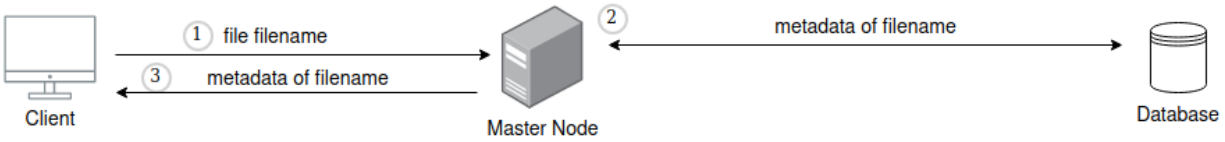
1. Login Feature not available



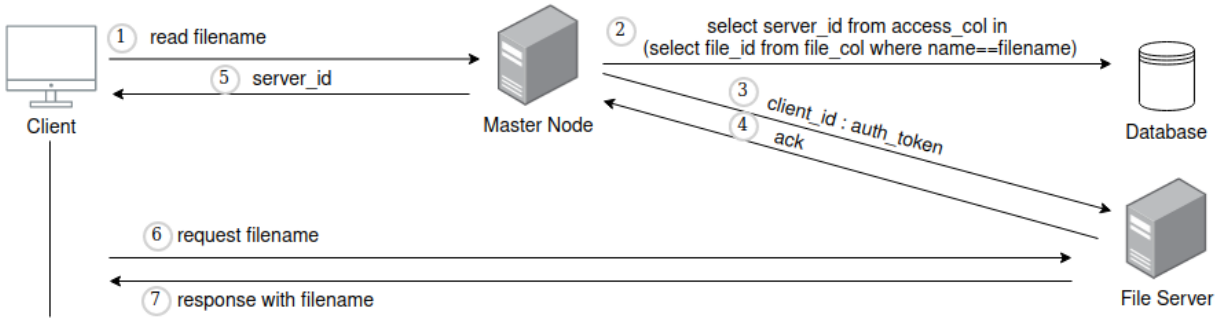
2. List entries of a Folder



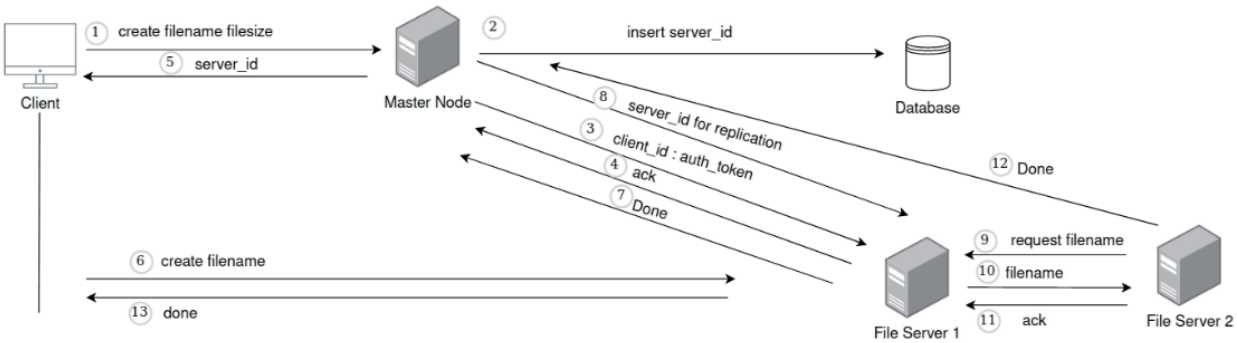
3. Get file properties(file command)



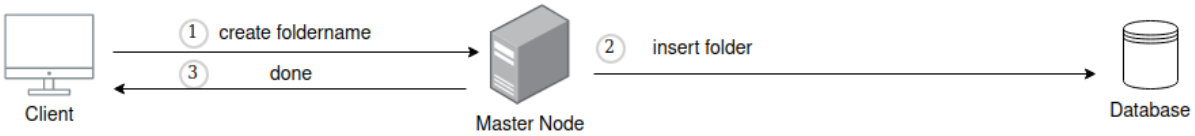
4. Read file



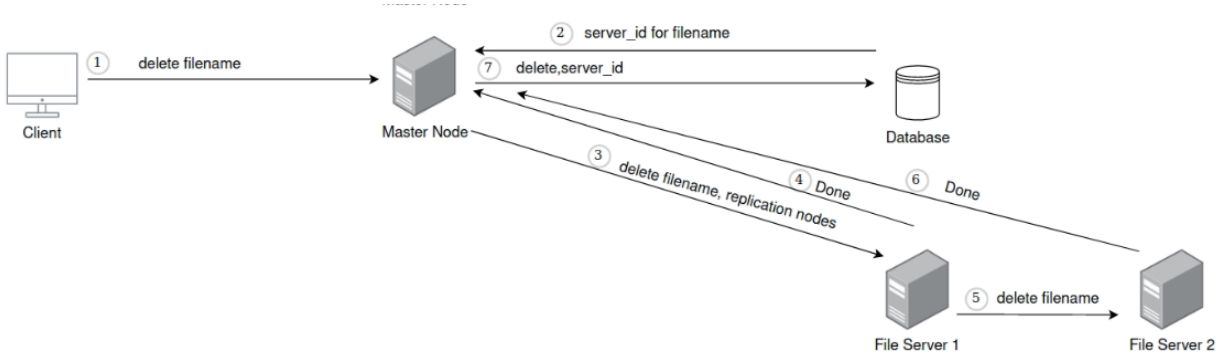
5. Create File



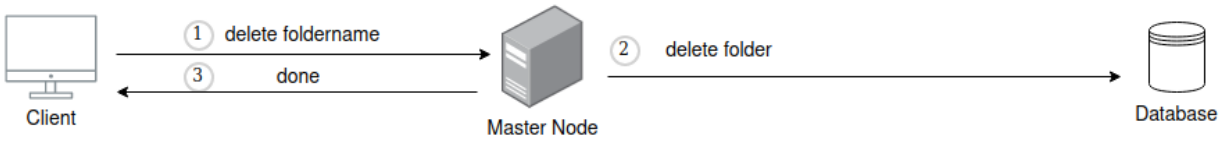
6. Create folder



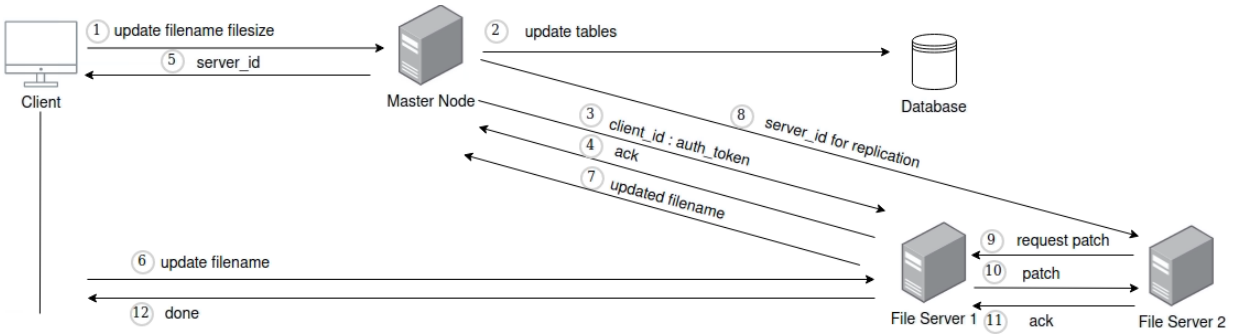
7. Delete file(s)



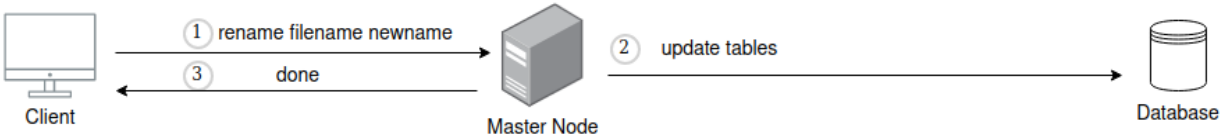
8. Delete folder(s)



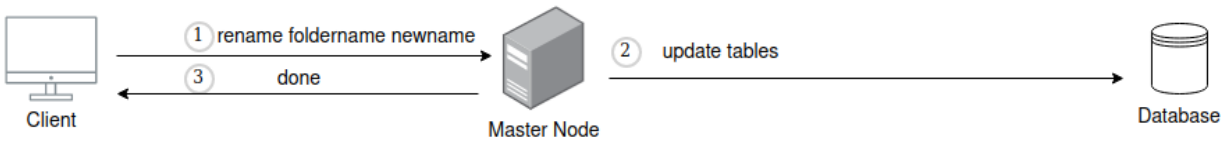
9. Update File



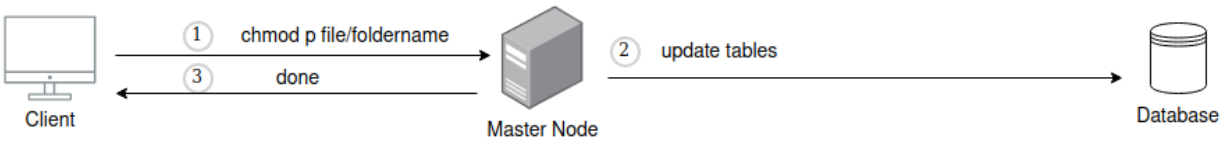
10. Rename file



11. Rename Folder Feature not available



12. Update Permissions Feature not available



References:

1. Go-FUSE Documentation
2. HDFS Documentation
3. Linux File System Documentation
4. MongoDB Documentation
5. Libfuse Documentation