University of Antwerp
Faculty of Applied Engineering

University of Antwerp
Faculty of Applied Engineering

Distributed Systems

Lab Report - Final

TADIWOS ANDARGIE
ID-no: 20249978

Group – 4: Bomboclat luchadores

Date: 26th May, 2025

Contents

LAB – 1

First Task: Git

1. Introduction

Git is an open-source version control system. It allows developers to track changes, collaborate efficiently, and manage different versions of their code.

This lab exercise involved setting up a local Git repository, creating a remote repository on GitHub, performing various Git operations, and resolving conflicts.

2. Objectives

- Create a local repository
- Create a public repository on Git Hub
- Create feature branches and test switching between branches
- To push and pull code from a remote repository
- Roll back to previous commits
- Merge the conflicts manually and in IDE

3. Methodology and Results

1 Setting Up a Git Repository

1. **Initialize a Git repository** in the project directory:

```
git init
```

2. Check the repository status:

```
git status
```

3. Add files to tracking:

```
git add HelloWorld
```

4. Commit the changes:

```
git commit -m "Initial commit"
```

2

```
tadiwos@tadiwos-Legion-Slim:~$ cd /home/tadiwos/6th Semester/Distributed Systems/Labs/Lab_
bash: cd: too many arguments
tadiwos@tadiwos-Legion-Slim:~$ cd "/home/tadiwos/6th Semester/Distributed Systems/Labs/Lab_1"
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ ls
HelloWorld
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ git init
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /home/tadiwos/6th Semester/Distributed Systems/Labs/Lab_1/.git/
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ ls
HelloWorld
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ git add HelloWorld
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ git commit -m "Initial Commit"
Author identity unknown

*** Please tell me who you are.

Run

  git config --global user.email "you@example.com"
  git config --global user.name "Your Name"

to set your account's default identity.
Omit --global to set the identity only in this repository.

fatal: unable to auto-detect email address (got 'tadiwos@tadiwos-Legion-Slim.(none)')
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ git config user.name "Tadiwos"
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ git config user.email "tadiwosadanefa@gmail.com"
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ git config --global --list
fatal: unable to read config file '/home/tadiwos/.gitconfig': No such file or directory
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ git commit -m "Initial Commit"
[master (root-commit) 9d45304] Initial Commit
 5 files changed, 85 insertions(+)
 create mode 100644 HelloWorld/.gitignore
 create mode 100644 HelloWorld/.idea/.gitignore
 create mode 100644 HelloWorld/pom.xml
 create mode 100644 HelloWorld/src/main/java/org/example/HelloWorld.java
 create mode 100644 HelloWorld/src/main/java/org/example/Main.java
```

Creating and Linking a Remote Repository

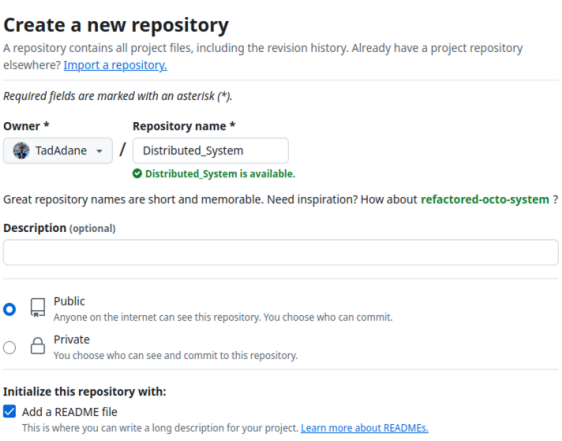
Create a new repository on GitHub.

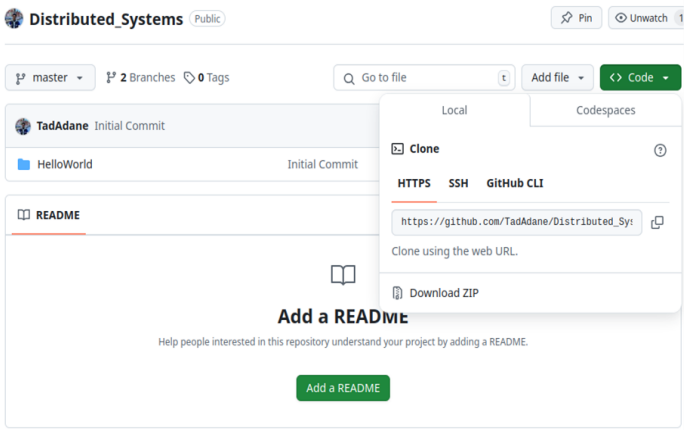
1. Link the local repository to the remote GitHub repository:

git remote add origin https://github.com/TadAdane/Distributed_Systems.git

2. Verify the remote repository:

git remote -v





```
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   .idea/misc.xml

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        Screenshot From 2025-03-06 11-58-03.png
        Screenshot From 2025-03-06 11-58-21.png
        Screenshot From 2025-03-06 11-58-38.png

no changes added to commit (use "git add" and/or "git commit -a")
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ git remote add origin https://github.com/TadAdane/Distributed_Systems.git
error: remote origin already exists.
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ git remote -v
origin  https://github.com/TadAdane/Distributed_Systems.git (fetch)
origin  https://github.com/TadAdane/Distributed_Systems.git (push)
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$
```

3 Creating and Switching Between Branches

1. Create a new feature branch:

git branch feature-branch

2. Switch to the feature branch:

git checkout feature-branch

3. Modify a file and commit changes:

git add .
git commit -m "Update feature-branch"

```
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ git checkout -b feature-branch
Switched to a new branch 'feature-branch'
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ git status
On branch feature-branch
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   HelloWorld/src/main/java/org/example/HelloWorld.java

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .idea/

no changes added to commit (use "git add" and/or "git commit -a")
```

```
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ git add .
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ git commit -m "Update feature branch"
[feature-branch 4eaa775] Update feature branch
 6 files changed, 39 insertions(+), 1 deletion(-)
 create mode 100644 .idea/.gitignore
 create mode 100644 .idea/Lab_1.iml
 create mode 100644 .idea/misc.xml
 create mode 100644 .idea/modules.xml
 create mode 100644 .idea/vcs.xml
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ git checkout main
error: pathspec 'main' did not match any file(s) known to git
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ git status
On branch master
Your branch is up to date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .idea/

nothing added to commit but untracked files present (use "git add" to track)
```

4 Pushing Code to Remote Repository

1. Push the main branch to GitHub:

git push -u origin main

2. Push the feature branch:

git push origin feature-branch

```
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ git push -u origin main
error: src refspec main does not match any
error: failed to push some refs to 'https://github.com/TadAdane/Distributed_Systems.git'
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .idea/

nothing added to commit but untracked files present (use "git add" to track)
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ git push -u origin master
Username for 'https://github.com': TadAdane
Password for 'https://TadAdane@github.com':
remote: Support for password authentication was removed on August 13, 2021.
remote: Please see https://docs.github.com/get-started/getting-started-with-git/about-remote-repositories#cloning-with-https-urls for information on currently recommended modes of authentication.
fatal: Authentication failed for 'https://github.com/TadAdane/Distributed_Systems.git/'
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ ^C
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ git push -u origin master
Username for 'https://github.com': TadAdane
Password for 'https://TadAdane@github.com':
remote: Support for password authentication was removed on August 13, 2021.
remote: Please see https://docs.github.com/get-started/getting-started-with-git/about-remote-repositories#cloning-with-https-urls for information on currently recommended modes of authentication.
```

```
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ ^C
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ git push -u origin master
Username for 'https://github.com': TadAdane
Password for 'https://TadAdane@github.com':
remote: Support for password authentication was removed on August 13, 2021.
remote: Please see https://docs.github.com/get-started/getting-started-with-git/about-remote-repositories#cloning-with-https-urls for information on currently recommended modes of authentication.
fatal: Authentication failed for 'https://github.com/TadAdane/Distributed_Systems.git/'
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ git push -u origin master
Username for 'https://github.com': TadAdane
Password for 'https://TadAdane@github.com':
remote: Support for password authentication was removed on August 13, 2021.
remote: Please see https://docs.github.com/get-started/getting-started-with-git/about-remote-repositories#cloning-with-https-urls for information on currently recommended modes of authentication.
fatal: Authentication failed for 'https://github.com/TadAdane/Distributed_Systems.git/'
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ git push -u origin master
Username for 'https://github.com': TadAdane
Password for 'https://TadAdane@github.com':
remote: Support for password authentication was removed on August 13, 2021.
remote: Please see https://docs.github.com/get-started/getting-started-with-git/about-remote-repositories#cloning-with-https-urls for information on currently recommended modes of authentication.
fatal: Authentication failed for 'https://github.com/TadAdane/Distributed_Systems.git/'
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ git push -u origin master
Username for 'https://github.com': TadAdane
Password for 'https://TadAdane@github.com':
Enumerating objects: 14, done.
Counting objects: 100% (14/14), done.
Delta compression using up to 16 threads
Compressing objects: 100% (8/8), done.
Writing objects: 100% (14/14), 1.87 KiB | 956.00 KiB/s, done.
Total 14 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://github.com/TadAdane/Distributed_Systems.git
 * [new branch]      master -> master
branch 'master' set up to track 'origin/master'.
```

5 Rolling Back to a Previous Commit

1. View commit history:

git log --oneline

2. Roll back to a previous commit:

git reset --hard 9d45304

```
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ git log --oneline
9d45304 (HEAD -> master, origin/master) Initial Commit
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ git reset --hard <commit-hash>
bash: syntax error near unexpected token `newline'
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ git reset --hard 9d45304
HEAD is now at 9d45304 Initial Commit
```

6 Simulating and Resolving Merge Conflicts

1. Modify the same file in both main (on GitHub) and Master (locally) with different changes.
2. Attempt to push the local changes:

```
git push origin main
```

- Git rejects the push due to conflicting remote changes.

3. Pull the latest changes from GitHub, triggering a merge conflict:

```
git pull origin main
```

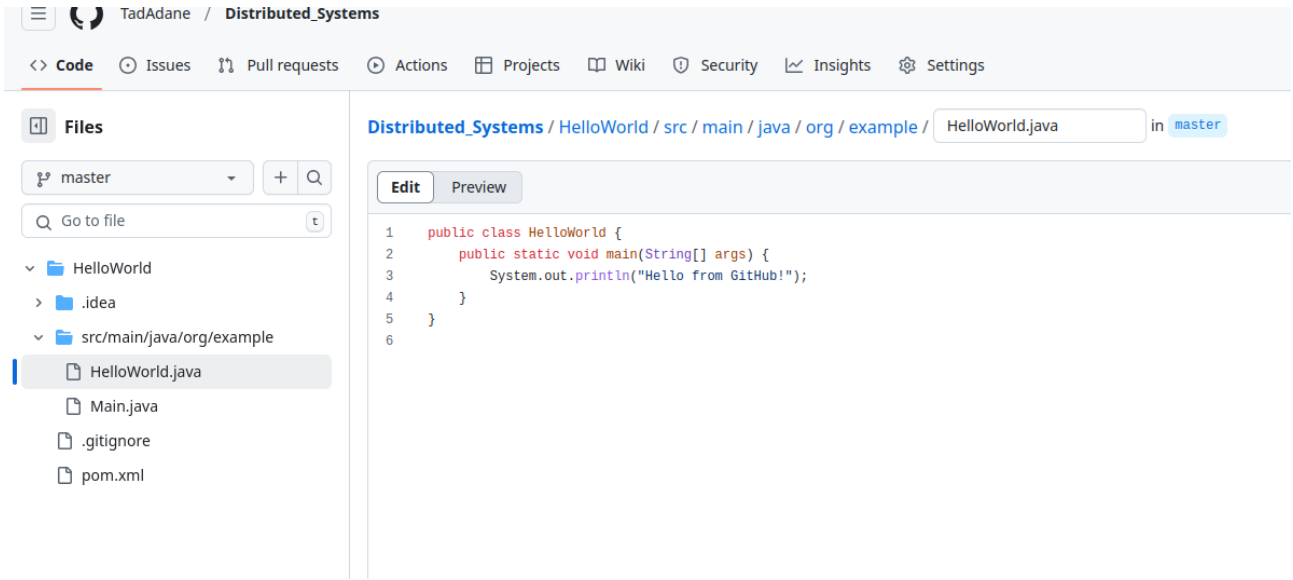
4. Open the conflicted file in **IntelliJ IDEA**. The merge tool displays the conflicting changes from both branches.
5. Manually resolve the conflict by merging the changes in IntelliJ IDEA and selecting "Apply Changes and Mark Resolved."
6. Add and commit the resolved file using Git on IntelliJ IDEA

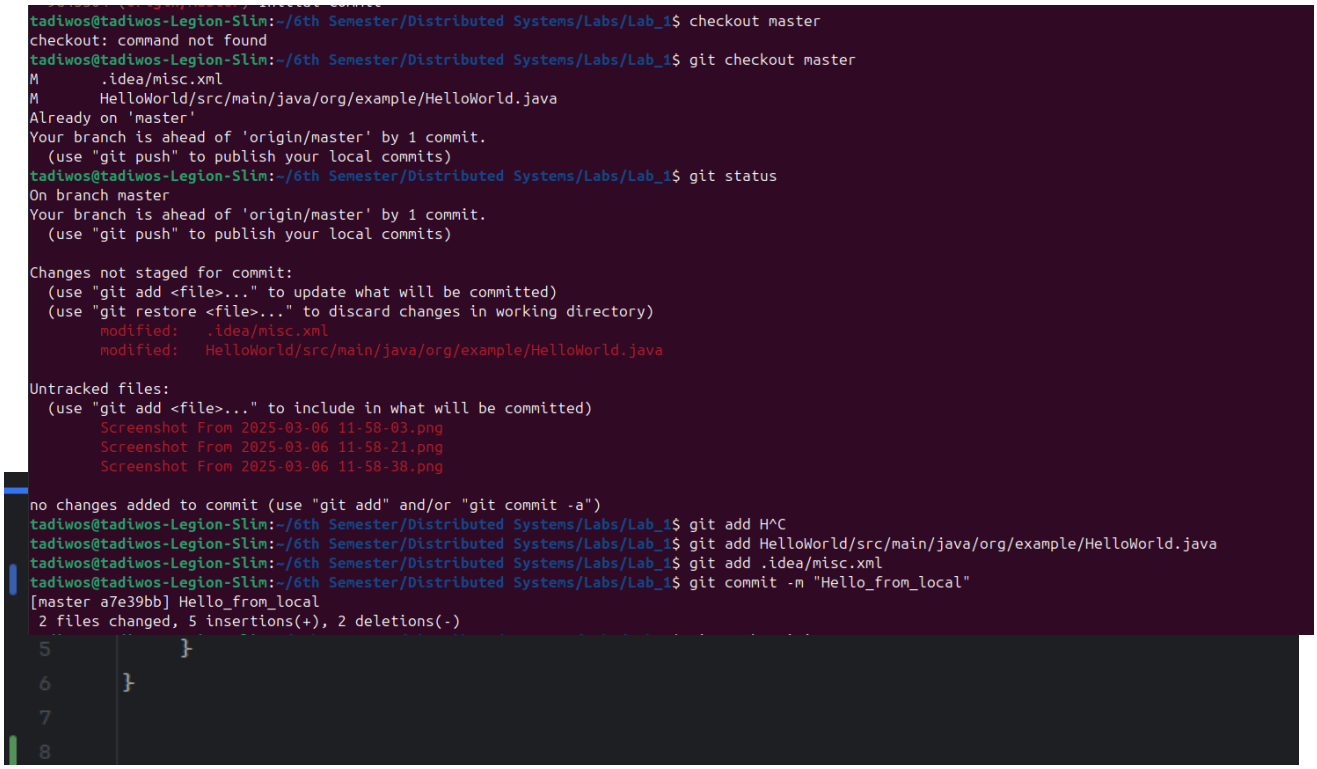
```
git add .
git rebase --continue
```

7. Push the resolved changes to GitHub:

```
git push origin main
```

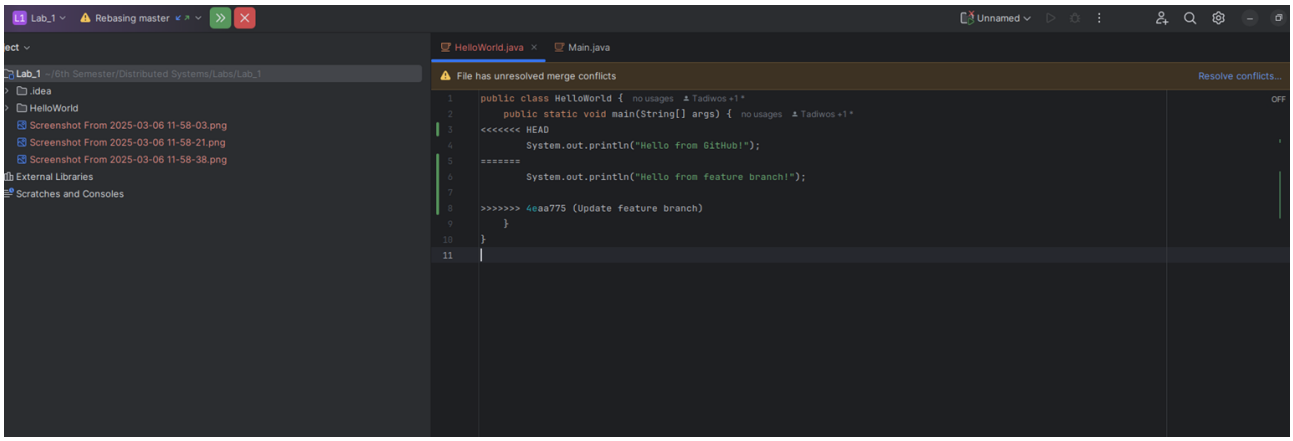
CREATING CONFLICTS

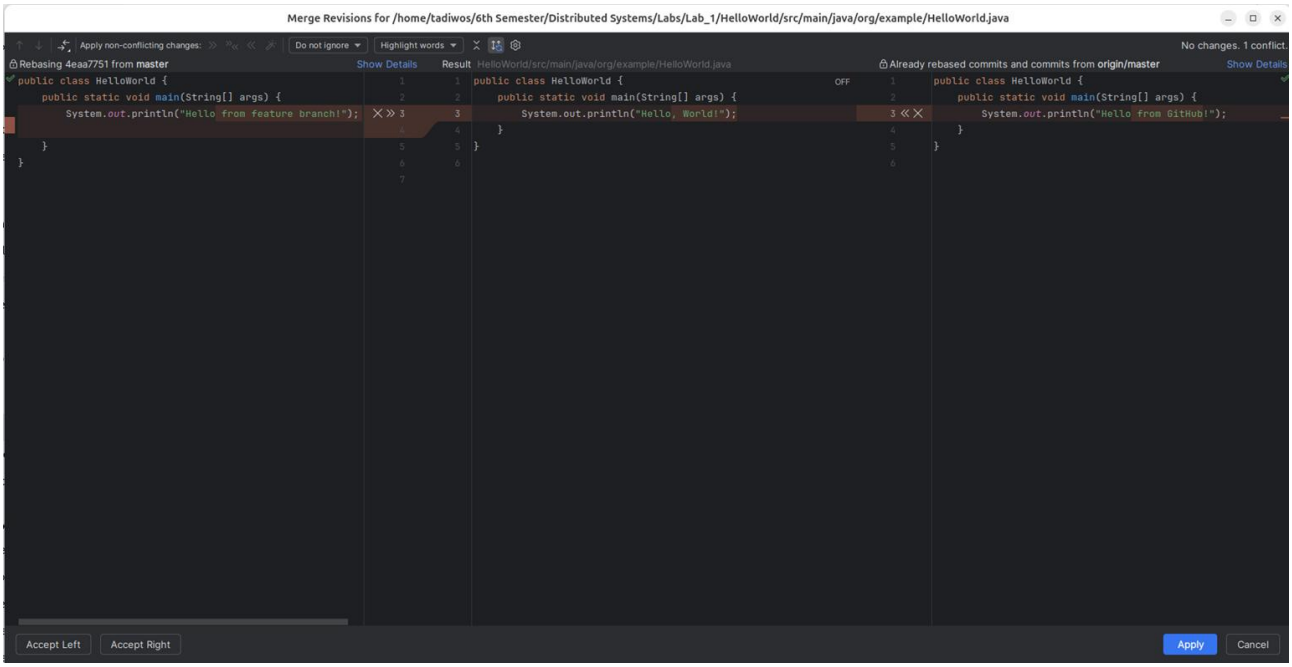




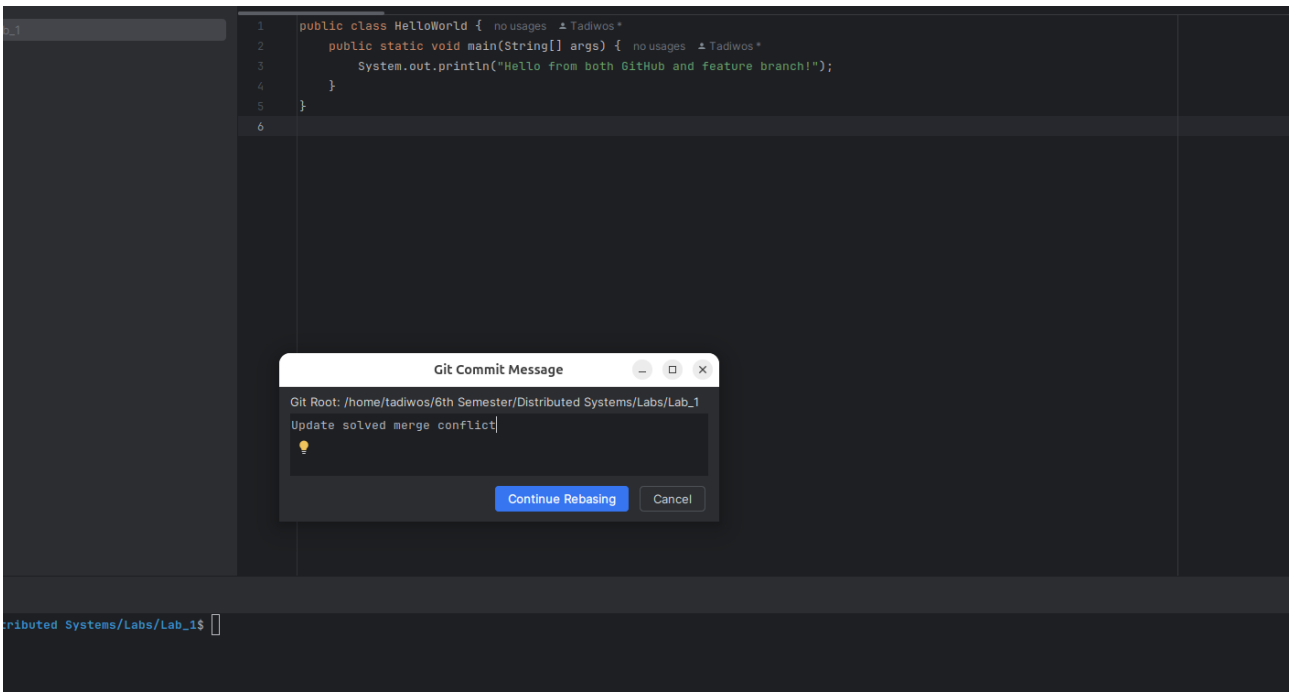
TRYING TO PUSH TO REMOTE REPOSITORY WITHOUT RESOLVING CONFLICT

```
and the repository exists.
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ git push origin master
Username for 'https://github.com': TadAdane
Password for 'https://TadAdane@github.com':
To https://github.com/TadAdane/Distributed_Systems.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://github.com/TadAdane/Distributed_Systems.git'
hint: Updates were rejected because the remote contains work that you do not
hint: have locally. This is usually caused by another repository pushing to
hint: the same ref. If you want to integrate the remote changes, use
hint: 'git pull' before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ git pull origin master
remote: Enumerating objects: 17, done.
remote: Counting objects: 100% (17/17), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 9 (delta 2), reused 0 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (9/9), 1.23 KiB | 629.00 KiB/s, done.
From https://github.com/TadAdane/Distributed_Systems
 * branch            master     -> FETCH_HEAD
   9d45304..6bd8d34  master     -> origin/master
hint: You have divergent branches and need to specify how to reconcile them.
hint: You can do so by running one of the following commands sometime before
hint: your next pull:
hint:
hint:   git config pull.rebase false  # merge
hint:   git config pull.rebase true   # rebase
hint:   git config pull.ff only       # fast-forward only
hint:
hint: You can replace "git config" with "git config --global" to set a default
hint: preference for all repositories. You can also pass --rebase, --no-rebase,
hint: or --ff-only on the command line to override the configured default per
hint: invocation.
fatal: Need to specify how to reconcile divergent branches.
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ git status
On branch master
Your branch and 'origin/master' have diverged,
and have 2 and 1 different commits each, respectively.
  (use "git pull" if you want to integrate the remote branch with yours)

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        Screenshot From 2025-03-06 11-58-03.png
        Screenshot From 2025-03-06 11-58-21.png
        Screenshot From 2025-03-06 11-58-38.png

nothing added to commit but untracked files present (use "git add" to track)
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/Lab_1$ git pull --rebase origin master
From https://github.com/TadAdane/Distributed_Systems
 * branch            master     -> FETCH_HEAD
Auto-merging HelloWorld/src/main/java/org/example/HelloWorld.java
CONFLICT (content): Merge conflict in HelloWorld/src/main/java/org/example/HelloWorld.java
error: could not apply 4eaa775... Update feature branch
hint: Resolve all conflicts manually, mark them as resolved with
hint: "git add/rm <conflicted_files>", then run "git rebase --continue".
hint: You can instead skip this commit: run "git rebase --skip".
hint: To abort and get back to the state before "git rebase", run "git rebase --abort".
hint: Disable this message with "git config advice.mergeConflict false"
Could not apply 4eaa775... Update feature branch
```





RESOLVING CONFLICT



TRYING TO PUSH TO REMOTE REPOSITORY AFTER RESOLVING CONFLICT

Results and Observations

- Successfully initialized a Git repository and linked it to GitHub.
- Feature branches were created, modified.
- Push and pull operations were performed without errors.
- Merge conflicts were successfully resolved manually in IntelliJ IDEA.
- Rolling back to a previous commit restored an older version of the project.

4. Challenges and Solutions

Challenges Faced:

1. Authentication issues when pushing to GitHub (GitHub removed password authentication in 2021).

Solutions Implemented:

1. Used **GitHub Personal Access Tokens (PAT)** instead of passwords for authentication.

5. Conclusion

This lab provided hands-on experience with Git commands for local and remote repository management. It provided experience on to create, manage, and merge branches. Resolving merge conflicts and rolling back to previous versions showcased realistic scenarios one might face and this lab helped to see how to deal with it. Overall, the tasks completed reinforced version control concepts crucial for collaborative development.

Second Task: TCP and UDP connections

1. Introduction

TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) are essential networking protocols, each for different applications. TCP is connection-based, ensuring reliable, ordered, and error-free data transfer, making it ideal for applications like file transfers.

In this lab, the focus was on implementing network communication between a **client** and **server** using both **TCP (Transmission Control Protocol)** and **UDP (User Datagram Protocol)**. The task involved creating two types of servers (TCP and UDP), handling client requests, and utilizing multithreading for efficient resource management in the server-side applications.

This report will describe the processes followed to develop and test a **TCP/UDP server-client communication application**, including setting up the necessary environment and tools, implementing the solution, and troubleshooting conflicts during Git operations.

2. Objectives

The objectives of this lab were:

- To develop a **TCP/UDP client-server application**.
- To implement **multithreading** in the TCP server to handle multiple client connections.
- To understand and use **Git** for version control, including pushing code to **GitHub** and resolving **merge conflicts**.

3. Methodology

3.1 Setting Up the TCP/UDP Server and Client

For the TCP server-client communication:

- TCP server listens for incoming connections on a predefined port.
- TCP client sends requests to the server, and the server sends back the requested files.

For the UDP server-client communication:

- UDP server listens for client requests and sends files in response.
- UDP client sends a file request, and the server sends the requested file in small packets.

3.2 Git Setup and Branching

We used Git for version control and created a GitHub repository to push the code. The following Git commands were used:

- Initialized a local Git repository.
- Created a remote repository on GitHub and linked it with the local repository.
- Created a feature branch to develop the server-client communication.
- Implemented the code and committed the changes.
- Pushed the changes to the master branch on GitHub.

3.3 Simulating and Resolving Merge Conflicts

The following steps were taken to simulate and resolve merge conflicts:

- Modified the same file in both the main branch and feature branch.
- Attempted to merge the feature branch into main using:
    - git checkout main
    - git merge feature-branch

Conflicts were triggered in the HelloWorld.java file because both branches had modifications in the same lines.

Manually resolved the conflicts using IntelliJ IDEA's merge tool by combining the changes from both branches:

- System.out.println("Hello from both GitHub and feature branch!");

Marked the conflict as resolved in IntelliJ IDEA, committed the changes, and pushed the updated code to GitHub.

3.4 TCP Server and Client Development

TCP Server:

- The TCP server was implemented using Java's built-in networking library (java.net package).
- It uses a ServerSocket to listen for incoming connections, and a Socket is created for each client to communicate.

TCP Client:

- The TCP client connects to the server and sends a file request.

Single-threaded TCP server:

- The server could only handle one client at a time, blocking all other requests until the current client connection was processed.

```java
// Project lab1
public class TCPServer {  Edward
    public static void main(String[] args) {  Edward
        int port = 5000;

        try (ServerSocket serverSocket = new ServerSocket(port)) {
            System.out.println("[*] Single-threaded server running on port " + port);
            System.out.println("[*] Waiting for a client...");

            Socket clientSocket = serverSocket.accept();    // Accept one client
            System.out.println("[*] Client connected.");

            handleClient(clientSocket);                     // Process client requests
            System.out.println("[*] Client disconnected. Server shutting down.");

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

```java
    private static void handleClient(Socket clientSocket) throws IOException {  1 usage  Edward
        try (
                DataInputStream dis = new DataInputStream(clientSocket.getInputStream());
                DataOutputStream dos = new DataOutputStream(clientSocket.getOutputStream())
        ) {
            String fileName;
            while (!(fileName = dis.readUTF()).equalsIgnoreCase( anotherString: "exit")) {
                File file = new File(fileName);
                if (file.exists()) {
                    byte[] buffer = Files.readAllBytes(file.toPath()); // Read file directly
                    dos.writeInt(buffer.length);
                    dos.write(buffer);
                    System.out.println("[*] Sent file: " + fileName);
                } else {
                    dos.writeInt( v: 0);
                    System.out.println("[!] File not found: " + fileName);
                }
                dos.flush();
            }
            System.out.println("[*] Client requested to exit.");
        } catch (EOFException e) {
            System.out.println("[!] Client disconnected unexpectedly.");
        } finally {
            clientSocket.close();
        }
    }

}
```

The TCP server will keep checking when a client connects and will process the file requests of clients:

```
[*] Connected to server. Type 'exit' to quit.
[*] Enter file name to request (or 'exit' to quit): test.txt
[*] File 'test.txt' received successfully.
[*] Enter file name to request (or 'exit' to quit): test.txt
[*] File 'test.txt' received successfully.
[*] Enter file name to request (or 'exit' to quit): test.txt
[*] File 'test.txt' received successfully.
[*] Enter file name to request (or 'exit' to quit): exit
[*] Closing connection...
```

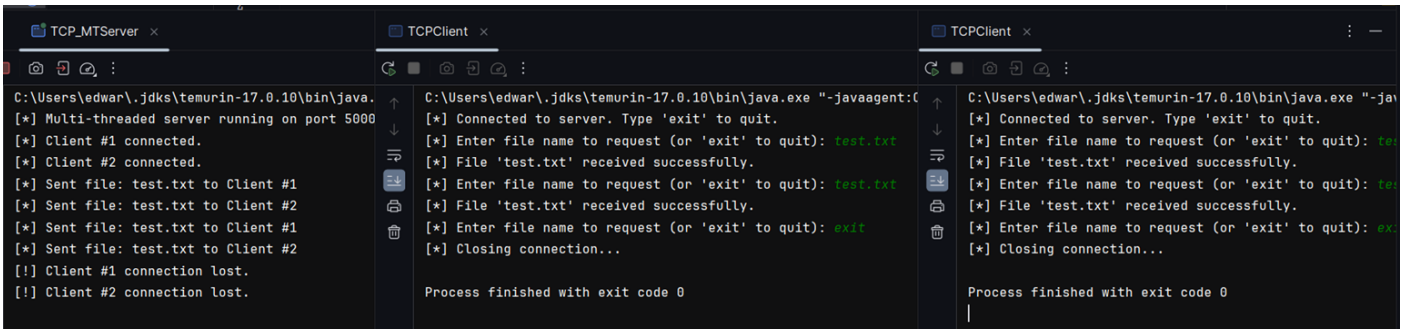
We can see that the TCP client received the requested file successfully:

Multithreaded TCP server:

- The server was optimized to handle multiple client connections simultaneously by assigning a separate thread for each client using Runnable interface.
- The server was able to manage different client requests in parallel, improving efficiency.

```java
public class TCPMultiThreadedServer {  Edward
    private static final AtomicInteger clientCount = new AtomicInteger( initialValue: 0);  1 usage

    public static void main(String[] args) {  Edward
        int port = 5000;
        try (ServerSocket serverSocket = new ServerSocket(port)) {
            System.out.println("[*] Multi-threaded server running on port " + port);
            while (true)
                new Thread(new ClientHandler(serverSocket.accept(), clientCount.incrementAndGet())).start();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

class ClientHandler implements Runnable {  1 usage  Edward
    private final Socket clientSocket;  4 usages
    private final int clientNumber;  7 usages

    public ClientHandler(Socket socket, int clientNumber) {  1 usage  Edward
        this.clientSocket = socket;
        this.clientNumber = clientNumber;
    }

    @Override  no usages  Edward
    public void run() {
        System.out.println("[*] Client #" + clientNumber + " connected.");
        try (DataInputStream dis = new DataInputStream(clientSocket.getInputStream());
             DataOutputStream dos = new DataOutputStream(clientSocket.getOutputStream())) {

            String fileName;
            while (!(fileName = dis.readUTF()).equalsIgnoreCase( anotherString: "exit")) {
                File file = new File(fileName);
                if (file.exists()) {
                    byte[] buffer = Files.readAllBytes(file.toPath());
                    dos.writeInt(buffer.length);
                    dos.write(buffer);
                    System.out.println("[*] Sent file: " + fileName + " to Client #" + clientNumber);
                } else {
                    dos.writeInt( v: 0);
```

difference with the single threaded server is that we explicitly log when a client connects, requests a file, or disconnects. Unlike the second version, it also sends the clientNumber back to the client as part of the response.



We can see here that different clients that joined at different times requested files from the server and they were able to receive the files:

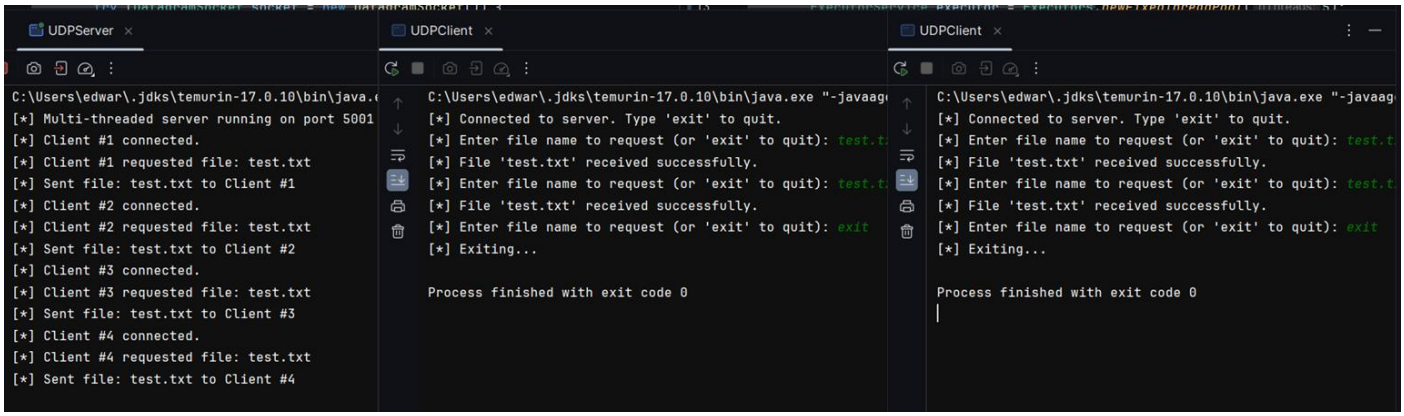
```
C:\Users\edwar>netstat -ano | findstr :5000
  TCP    0.0.0.0:5000           0.0.0.0:0              LISTENING       26792
  TCP    127.0.0.1:5000         127.0.0.1:60425        ESTABLISHED     26792
  TCP    127.0.0.1:5000         127.0.0.1:60429        ESTABLISHED     26792
  TCP    127.0.0.1:60425        127.0.0.1:5000         ESTABLISHED     31836
  TCP    127.0.0.1:60429        127.0.0.1:5000         ESTABLISHED     22440
  TCP    [::]:5000              [::]:0                 LISTENING       26792
```

3.5 UDP Server and Client Development

UDP Server:

- The UDP server listens on a specific port and processes incoming client requests concurrently using a fixed thread pool of 5 threads.
- The server uses DatagramSocket to receive and send packets.

```java
import ...

public class UDPServer {  Edward
    private static final int PORT = 5001;  2 usages
    private static final String EOF_MARKER = "EOF_SIGNAL";  1 usage
    private static int clientCount = 0;  1 usage

    public static void main(String[] args) {  Edward
        ExecutorService executor = Executors.newFixedThreadPool( nThreads: 5);

        try (DatagramSocket serverSocket = new DatagramSocket(PORT)) {
            System.out.println("[*] UDP Server running on port " + PORT);
            byte[] buffer = new byte[1024];

            while (true) {
                DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
                serverSocket.receive(packet);
                executor.execute(new ClientHandler(serverSocket, packet, ++clientCount));
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    static class ClientHandler implements Runnable {  1 usage  Edward
        private final DatagramSocket serverSocket;  3 usages
        private final DatagramPacket packet;  5 usages
        private final int clientId;  3 usages

        public ClientHandler(DatagramSocket socket, DatagramPacket packet, int clientId) {  1 usage  Edward
            this.serverSocket = socket;
            this.packet = packet;
            this.clientId = clientId;
        }

        @Override  no usages  Edward
        public void run() {
            try {
                String fileName = new String(packet.getData(),  offset: 0, packet.getLength());
                InetAddress clientAddress = packet.getAddress();
```



UDP Client:

- The UDP client sends a file request to the server and processes the incoming file in small chunks.

```java
public class UDPClient {  Edward

    public static void main(String[] args) {  Edward
        try (DatagramSocket socket = new DatagramSocket()) {
            BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
            System.out.println("[*] Connected to server. Type 'exit' to quit.");

            while (true) {
                System.out.print("[*] Enter file name to request (or 'exit' to quit): ");
                String fileName = reader.readLine();

                if (fileName.equalsIgnoreCase( anotherString: "exit")) {
                    System.out.println("[*] Exiting...");
                    break;
                }

                byte[] buffer = fileName.getBytes();
                DatagramPacket requestPacket = new DatagramPacket(buffer, buffer.length, InetAddress.getByName(SERVER_IP), SERVER_PORT);
                socket.send(requestPacket);

                FileOutputStream fos = new FileOutputStream(fileName);
                buffer = new byte[4096];
                DatagramPacket responsePacket = new DatagramPacket(buffer, buffer.length);

                while (true) {
                    socket.receive(responsePacket);
                    String response = new String(responsePacket.getData(),  offset: 0, responsePacket.getLength());

                    if ("FILE_NOT_FOUND".equals(response)) {
                        System.out.println("[!] File not found on server.");
                        break;
                    }

                    if (EOF_MARKER.equals(response)) {
                        System.out.println("[*] File '" + fileName + "' received successfully.");
                        break;
                    }

                    fos.write(responsePacket.getData(),  off: 0, responsePacket.getLength());
                }
```

Since UDP is connectionless, there were no "established" connections, but the server could still process multiple requests without a persistent connection.

4. Results and Observations

- The TCP server was able to handle multiple clients in the multithreaded version, allowing them to request files simultaneously.
- The UDP server worked efficiently by processing up to five client requests at a time.
- Git operations, including pushing code to GitHub, were successfully executed. A merge conflict was simulated by modifying the same file in both branches, and it was resolved manually in IntelliJ IDEA.
- The network traffic was tested using commands like `netstat` to monitor active connections, confirming that both TCP and UDP servers were functioning as expected.

5. Challenges and Solutions

Challenges Faced:

1. Merge conflicts: Occurred when pushing changes to GitHub after modifying the same lines in different branches.
2. Client connection limitations: The single-threaded TCP server could only handle one client at a time.
3. UDP client and server synchronization: Ensuring that files were correctly transmitted in small UDP packets.

Solutions Implemented:

1. Manually resolved Git merge conflicts using IntelliJ IDEA's merge tool.
2. Optimized the TCP server to handle multiple clients using multithreading.
3. Implemented a thread pool in the UDP server to handle multiple clients efficiently.

6. Conclusion

This lab provided hands-on experience with TCP/UDP communication and multithreading in Java. We successfully developed a client-server application that handled file requests using both protocols. Additionally, Git operations helped manage code versions, and merge conflicts were resolved to maintain a smooth workflow. This exercise reinforced key networking and version control concepts, essential for collaborative software development.

LAB - 2

1. Introduction

REST (Representational State Transfer) is a software architectural style used for designing networked applications. RESTful web services use HTTP methods for communication, where:

- GET retrieves data from the server.
- POST creates new data on the server.
- PUT updates existing data on the server.
- DELETE removes data from the server.

The goal of this lab was to develop a REST web service that simulates a bank account system using Spring Boot. By understanding Representational State Transfer (REST), the lab focused on handling standard HTTP methods (GET, POST, PUT, DELETE) to interact with a bank system. The project also included the implementation of thread safety to allow multiple clients to interact with the same bank account concurrently without causing data inconsistencies. The application was then deployed on a remote cloud server, providing a real-world scenario for full-stack development and cloud deployment.

2. Objectives

1. Develop REST-based server-client application in Java
    - The server provides a set of methods that are are accessible for the client
    - The server is designed to work as a bank
    - The client should be able to:
        - Get balance from account
        - Add money on the account
        - Remove money form the account
        - Get money from the account
2. If the bank account is joint, and two family members share it, extend exercise 1 with 2 clients who can do the same operation on the same account at the same time
3. Try to run the applications on top of the designated cloud resources, with one node action as a server, while another one or two act as clients

3. Methodologies and Results

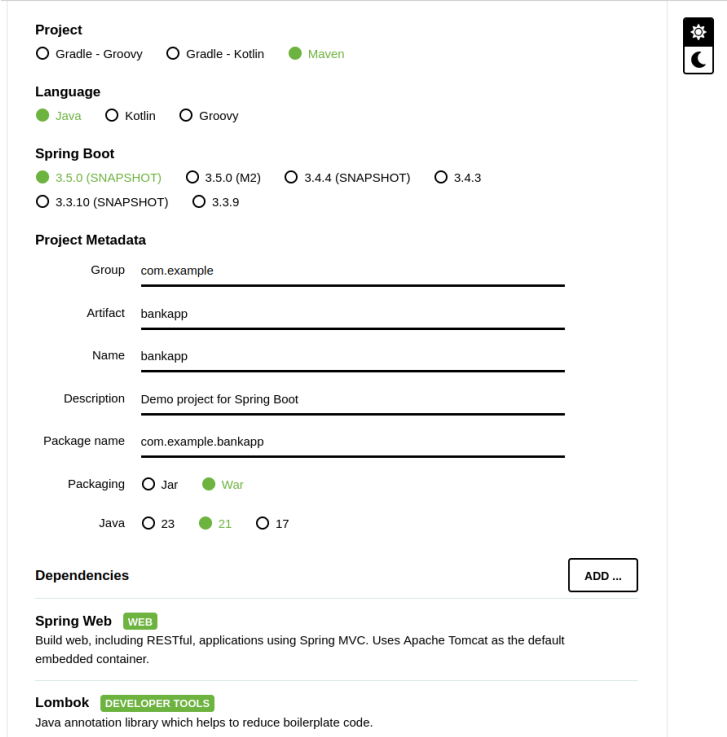
Implementation (Lab Procedure)

1. Set up the Development Environment:

- Spring Boot Framework
- Postman
- IDE (IntelliJ)
- Maven
    - Java 21 and Maven were installed on both local and remote servers.

sudo apt update
sudo apt install openjdk-21-jdk -y

- The project was created using Spring Boot to enable rapid development of RESTful



APIs.

2. Developing the Bank Account System:

- A BankAccount java model was created using IntelliJ.
- RESTful API methods were implemented to allow:
  - Account creation (**POST**): Create a new account.
  - Account retrieval (**GET**): Retrieve the account balance.
  - Deposit (**PUT**): Add money to an account.
  - Withdraw (**PUT**): Remove money from an account.
  - Account deletion (**DELETE**): Delete an account.

- Building the Spring Boot Application Locally:

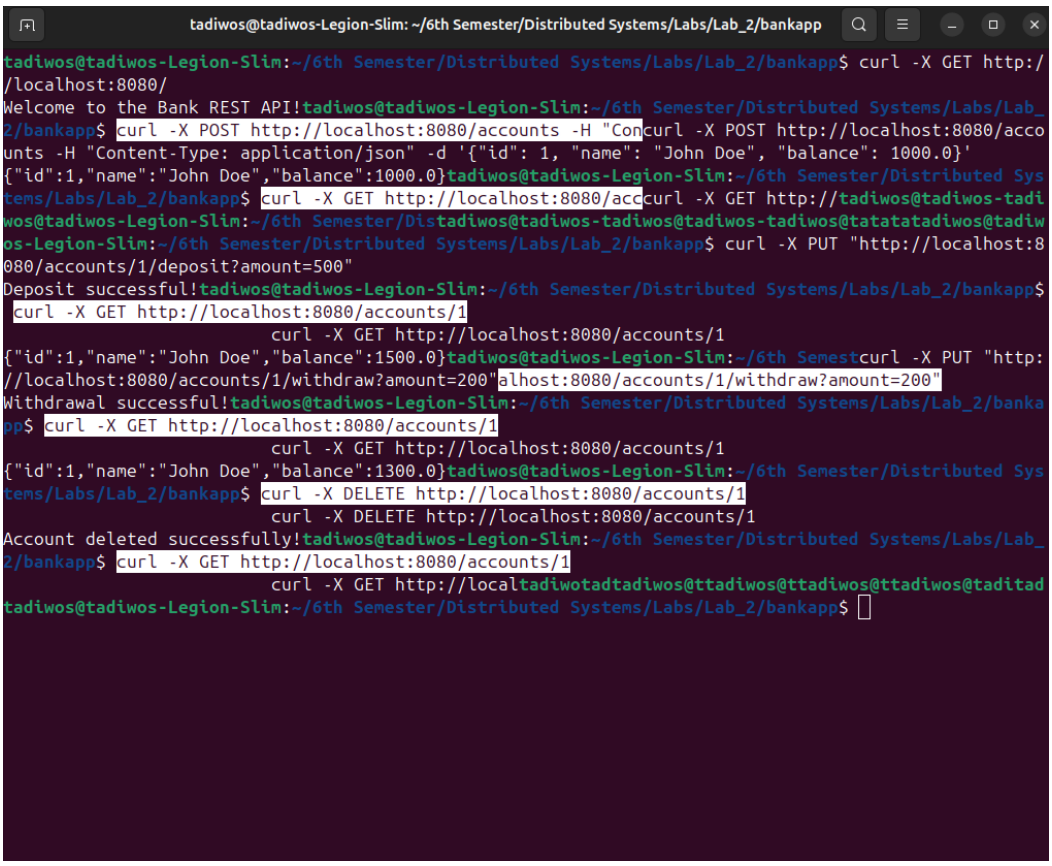
Before deploying to the cloud, I built and tested the Spring Boot application locally using Maven:

- mvn clean install

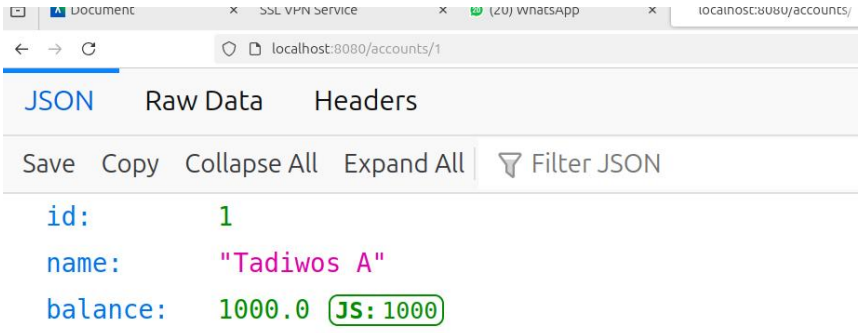
To run the Spring Boot application locally:

- mvn spring-boot:run

Running Curl commands for using REST

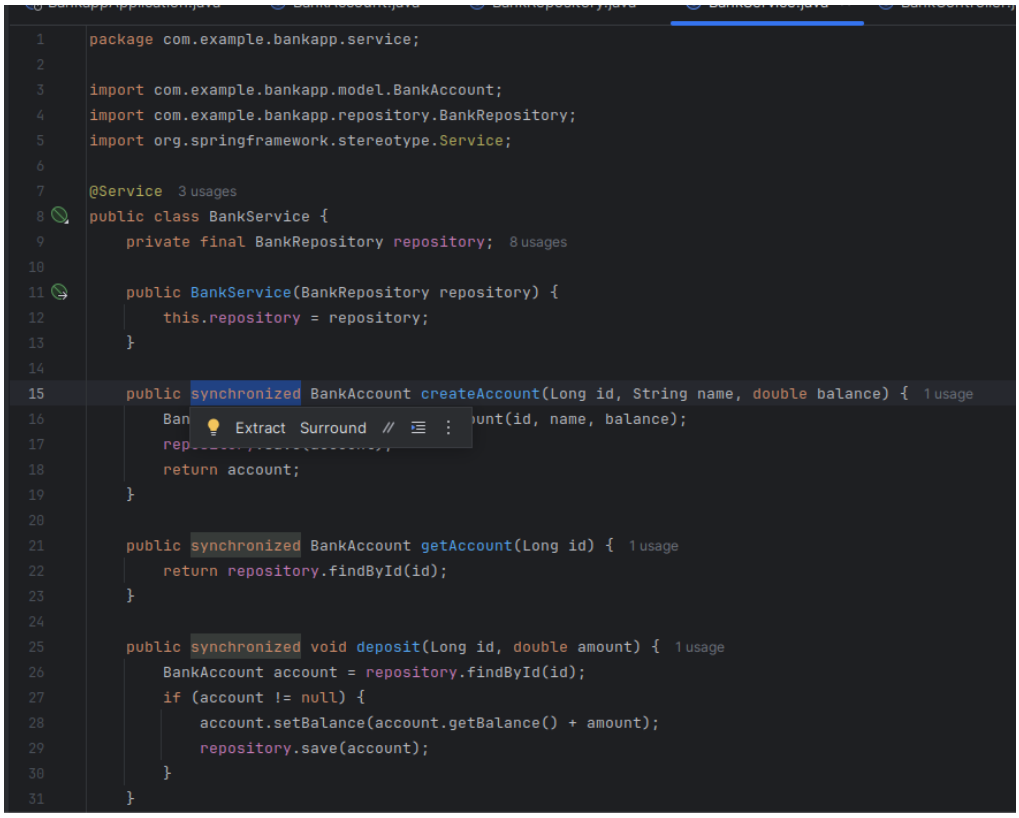


Web output for local



3. **Concurrency Handling**:

- **Thread safety** was implemented using **synchronized methods** in the BankService class to ensure that multiple clients can access and modify the same account concurrently without causing data inconsistencies.



```java
package com.example.bankapp.service;

import com.example.bankapp.model.BankAccount;
import com.example.bankapp.repository.BankRepository;
import org.springframework.stereotype.Service;

@Service  3 usages
public class BankService {
    private final BankRepository repository;  8 usages

    public BankService(BankRepository repository) {
        this.repository = repository;
    }

    public synchronized BankAccount createAccount(Long id, String name, double balance) {  1 usage
        Ban  Extract  Surround  //  ⋮ unt(id, name, balance);
        rep
        return account;
    }

    public synchronized BankAccount getAccount(Long id) {  1 usage
        return repository.findById(id);
    }

    public synchronized void deposit(Long id, double amount) {  1 usage
        BankAccount account = repository.findById(id);
        if (account != null) {
            account.setBalance(account.getBalance() + amount);
            repository.save(account);
        }
    }
```

4. **Deploying the Application**:

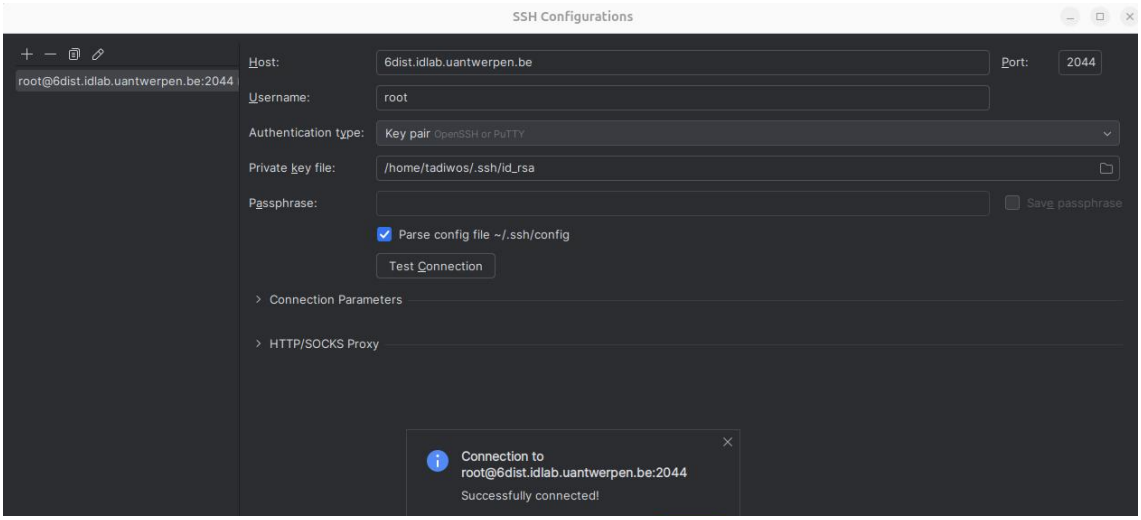
Two methods:

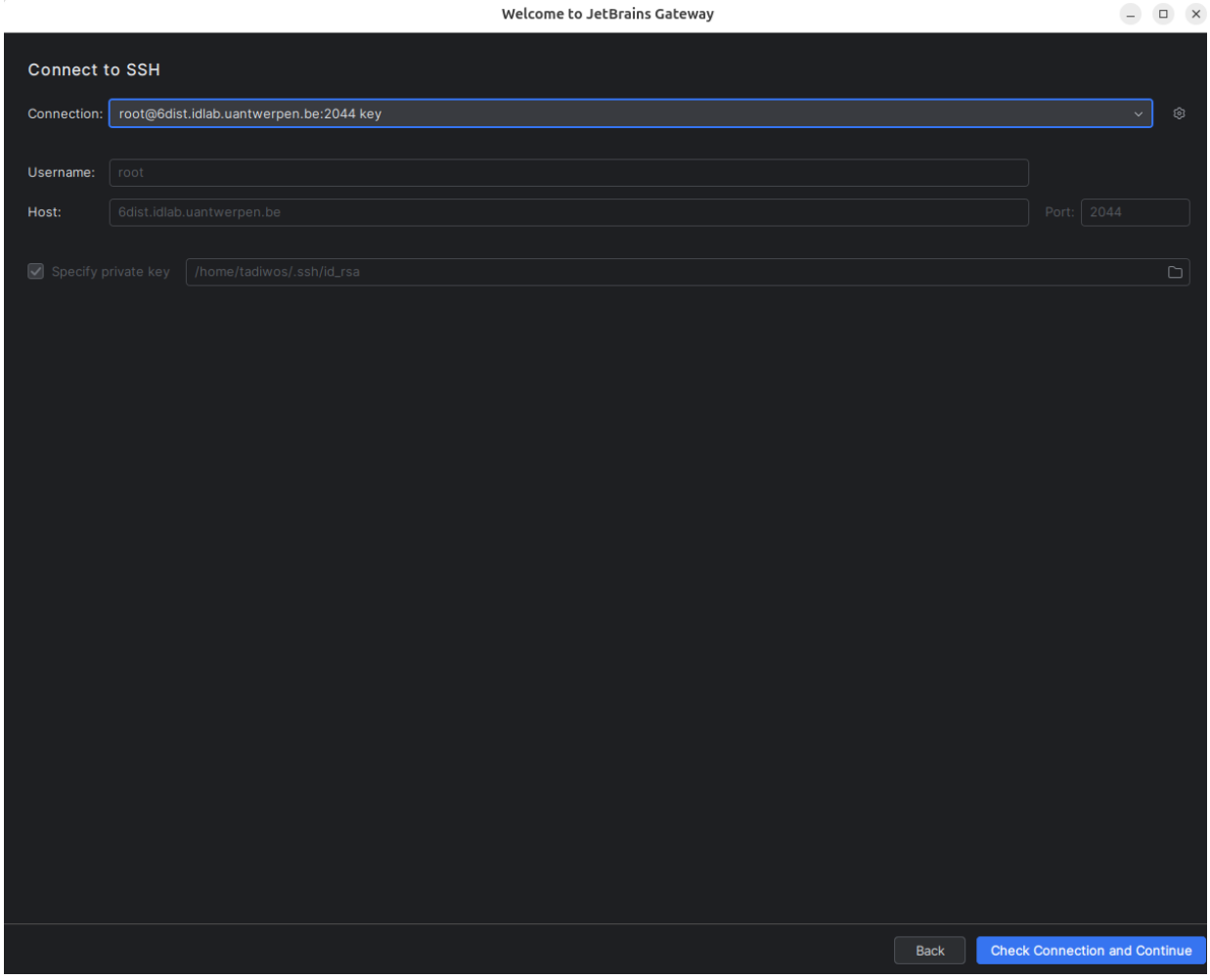
1) Using IntelliJ IDE

2) Using GitBash and Github

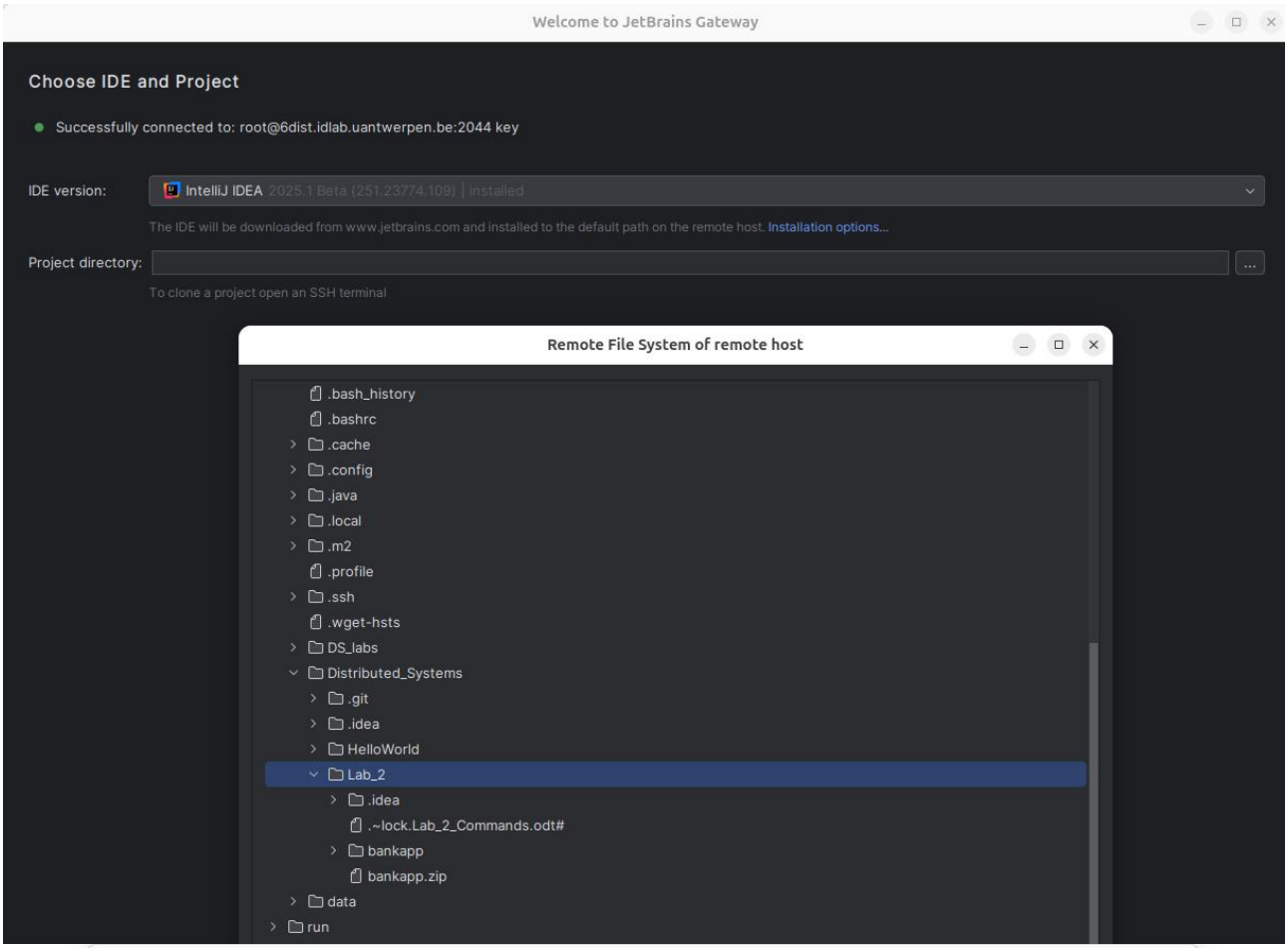
Using IntelliJ IDE:

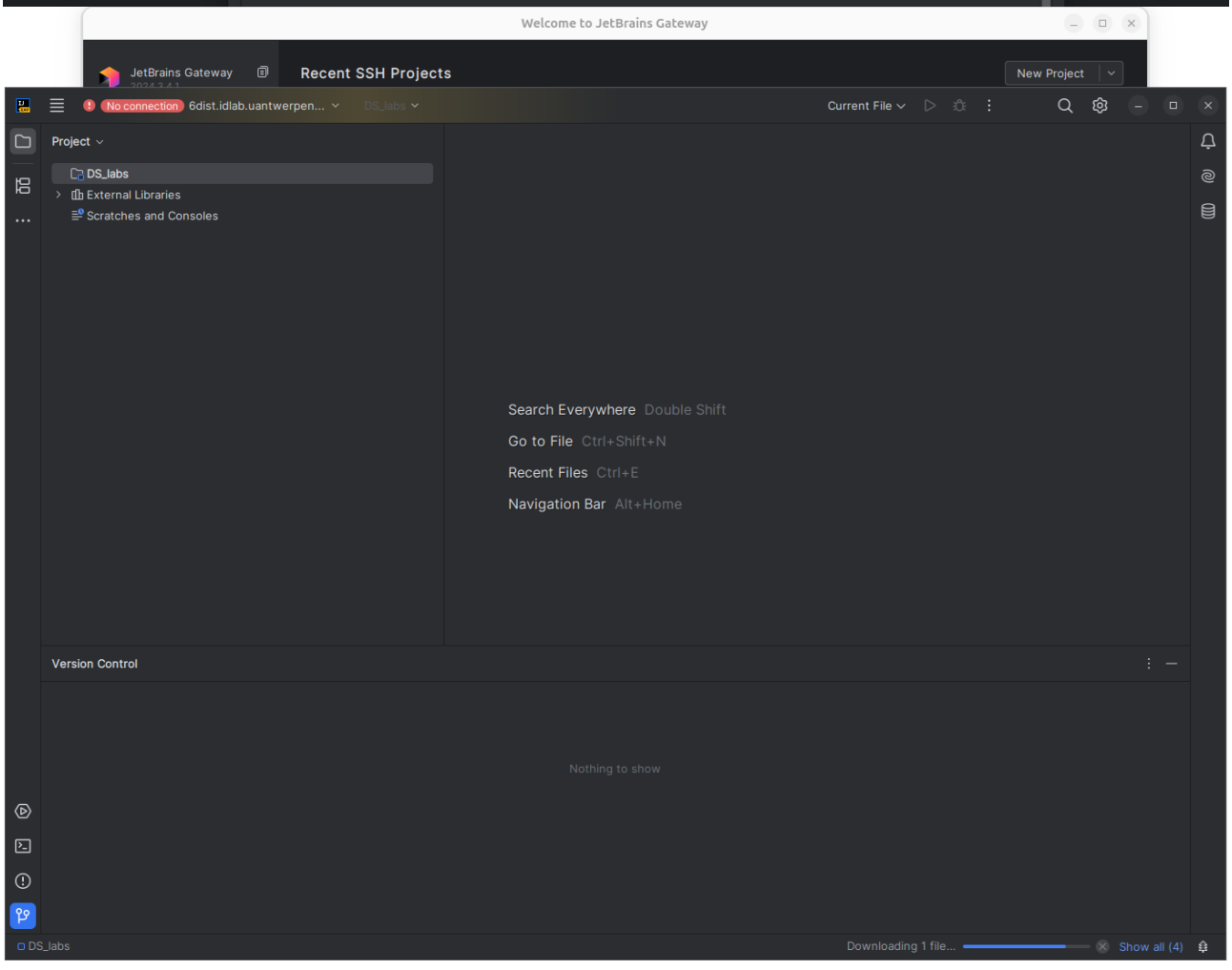
- Using SSH Configuration
Host: 6dist.idlab.uantwerpen.be
Username: root
Private key file: Key for node access (Private)





Create IntelliJ project using the IDE on the container once connection was established


Welcome to JetBrains Gateway
Choose IDE and Project
Successfully connected to: root@6dist.idlab.uantwerpen.be:2044 key
IDE version:
IntelliJ IDEA 2025.1 Beta (251.23774.109) | installed
The IDE will be downloaded from www.jetbrains.com and installed to the default path on the remote host. Installation options...
Project directory:
To clone a project open an SSH terminal
Remote File System of remote host
.bash_history
.bashrc
.cache
.config
.java
.local
.m2
.profile
.ssh
.wget-hsts
DS_labs
Distributed_Systems
.git
.idea
HelloWorld
Lab_2
.idea
.~lock.Lab_2_Commands.odt#
bankapp
bankapp.zip
data
run

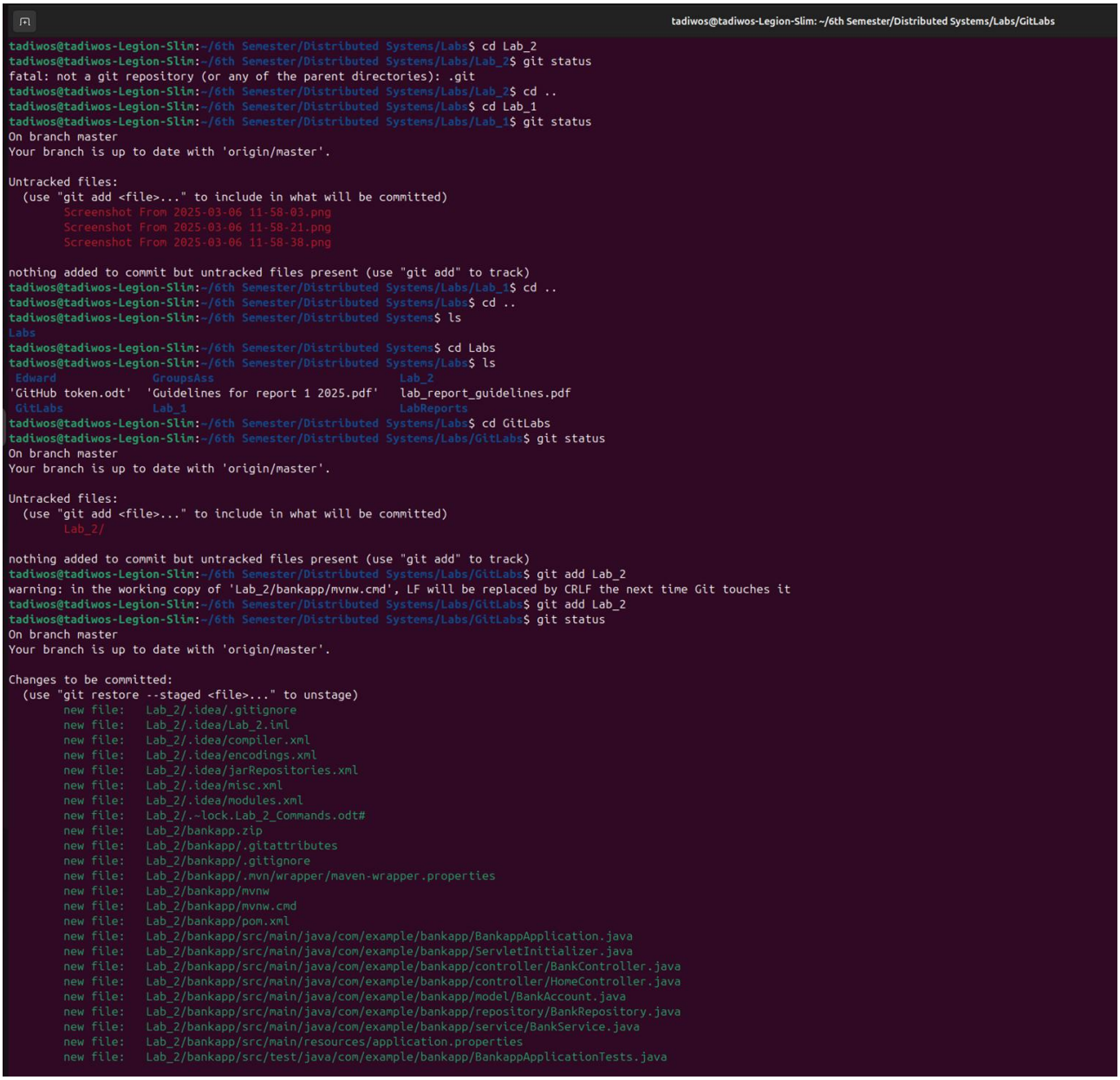

Welcome to JetBrains Gateway
JetBrains Gateway
Recent SSH Projects
New Project
No connection
6dist.idlab.uantwerpen...
DS_labs
Current File
Project
DS_labs
External Libraries
Scratches and Consoles
Search Everywhere Double Shift
Go to File Ctrl+Shift+N
Recent Files Ctrl+E
Navigation Bar Alt+Home
Version Control
Nothing to show
DS_labs
Downloading 1 file...
Show all (4)

This method tend to take a lot of time and some times even would lose connection hence didn't continue further in using to deploy with it.

2) Using GitBash and Github

- The project was pushed to GitHub and then pulled onto a remote Ubuntu server.
- Java 21 was set up on the server, and the Spring Boot application was deployed and tested using curl and Postman.

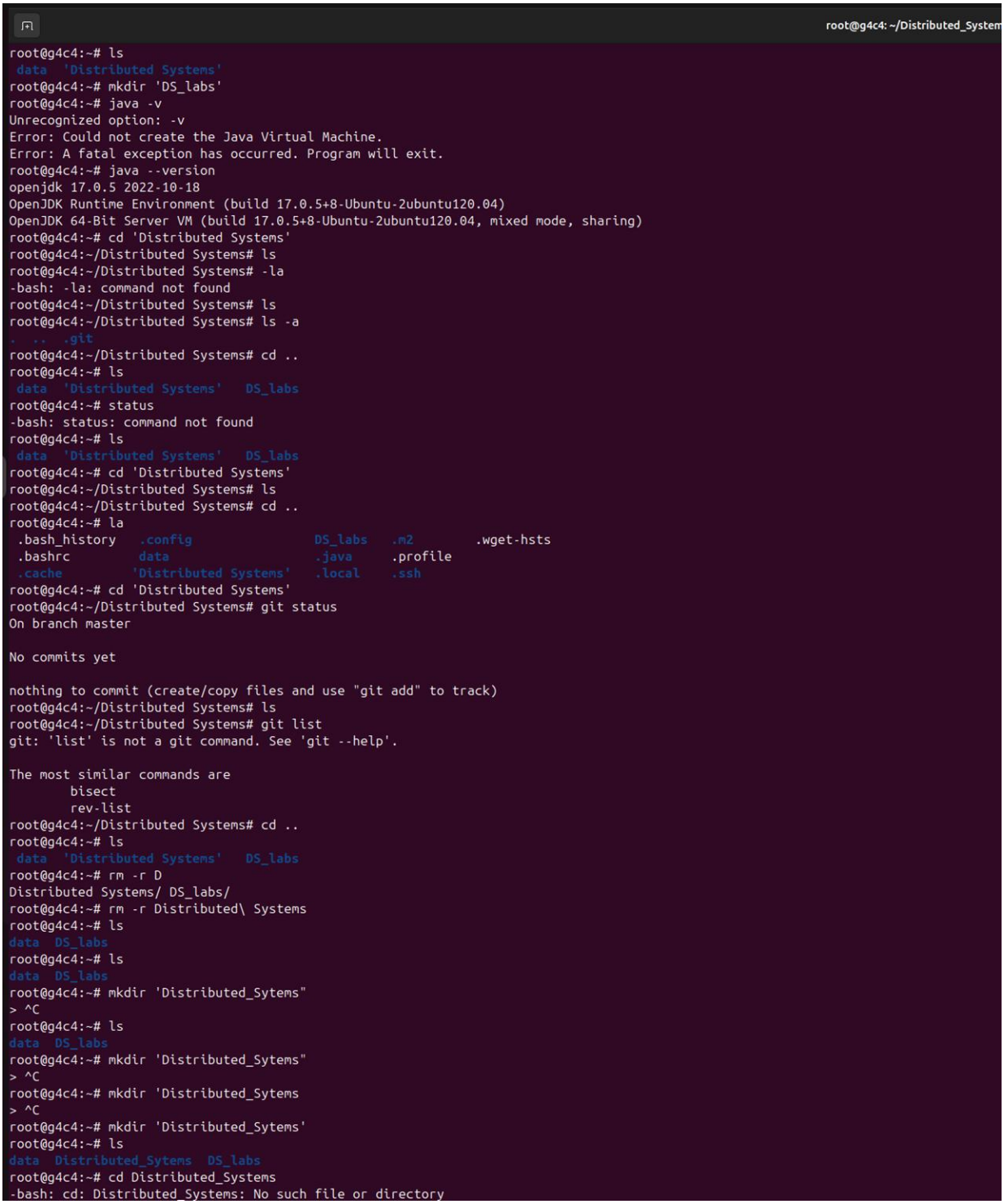
Create git in local project and push to Github remote repository

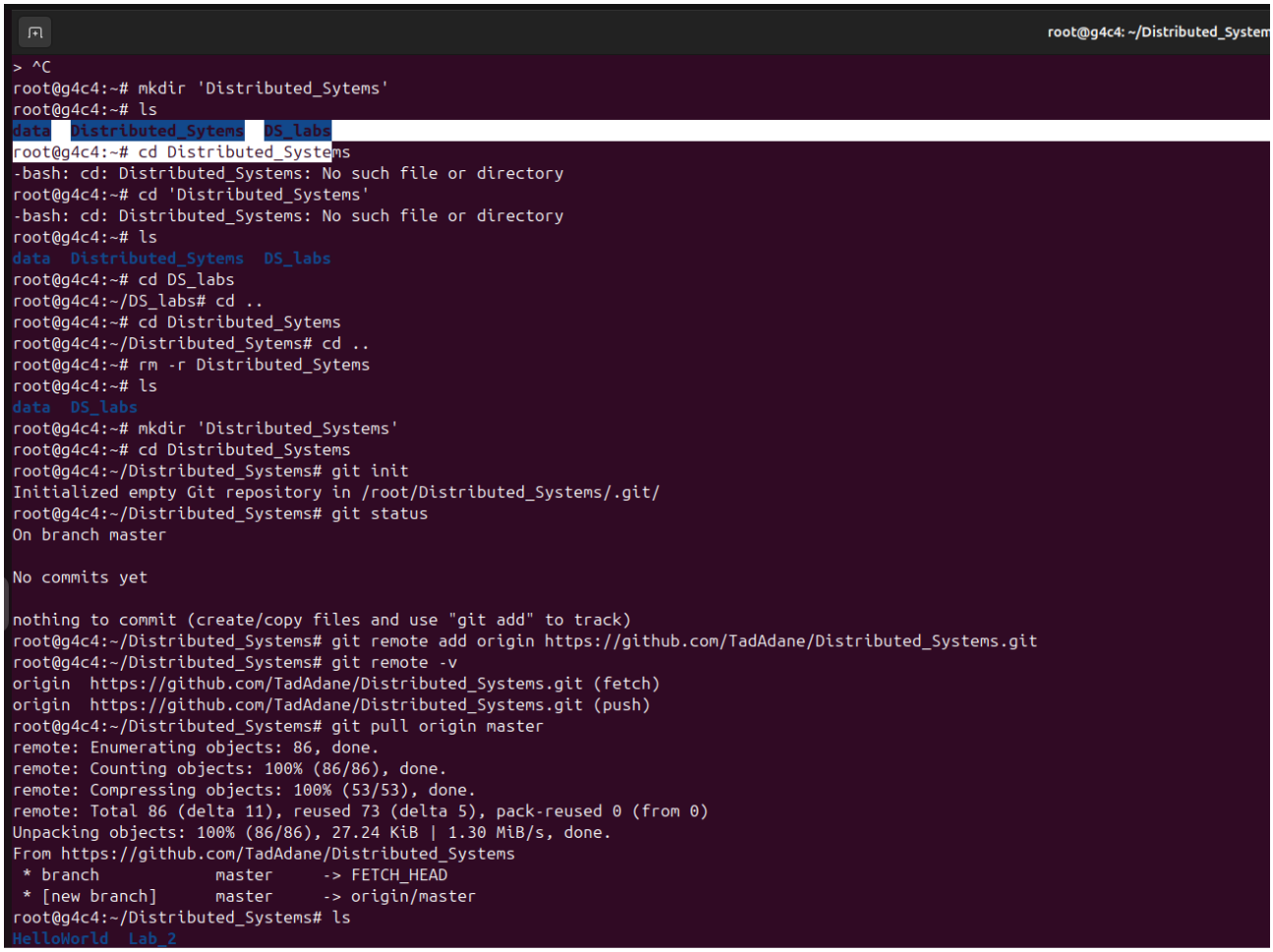


```
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/GitLabs$ git commit -m "Added Lab_2"
[master 4853bde] Added Lab_2
 24 files changed, 880 insertions(+)
 create mode 100644 Lab_2/.idea/.gitignore
 create mode 100644 Lab_2/.idea/Lab_2.iml
 create mode 100644 Lab_2/.idea/compiler.xml
 create mode 100644 Lab_2/.idea/encodings.xml
 create mode 100644 Lab_2/.idea/jarRepositories.xml
 create mode 100644 Lab_2/.idea/misc.xml
 create mode 100644 Lab_2/.idea/modules.xml
 create mode 100644 Lab_2/.~lock.Lab_2_Commands.odt#
 create mode 100644 Lab_2/bankapp.zip
 create mode 100644 Lab_2/bankapp/.gitattributes
 create mode 100644 Lab_2/bankapp/.gitignore
 create mode 100644 Lab_2/bankapp/.mvn/wrapper/maven-wrapper.properties
 create mode 100755 Lab_2/bankapp/mvnw
 create mode 100644 Lab_2/bankapp/mvnw.cmd
 create mode 100644 Lab_2/bankapp/pom.xml
 create mode 100644 Lab_2/bankapp/src/main/java/com/example/bankapp/BankappApplication.java
 create mode 100644 Lab_2/bankapp/src/main/java/com/example/bankapp/ServletInitializer.java
 create mode 100644 Lab_2/bankapp/src/main/java/com/example/bankapp/controller/BankController.java
 create mode 100644 Lab_2/bankapp/src/main/java/com/example/bankapp/controller/HomeController.java
 create mode 100644 Lab_2/bankapp/src/main/java/com/example/bankapp/model/BankAccount.java
 create mode 100644 Lab_2/bankapp/src/main/java/com/example/bankapp/repository/BankRepository.java
 create mode 100644 Lab_2/bankapp/src/main/java/com/example/bankapp/service/BankService.java
 create mode 100644 Lab_2/bankapp/src/main/resources/application.properties
 create mode 100644 Lab_2/bankapp/src/test/java/com/example/bankapp/BankappApplicationTests.java
```

```
tadiwos@tadiwos-Legion-Slim: ~/6th Semester/Distributed Systems/Labs/GitLabs
 create mode 100644 Lab_2/bankapp/src/main/java/com/example/bankapp/controller/HomeController.java
 create mode 100644 Lab_2/bankapp/src/main/java/com/example/bankapp/model/BankAccount.java
 create mode 100644 Lab_2/bankapp/src/main/java/com/example/bankapp/repository/BankRepository.java
 create mode 100644 Lab_2/bankapp/src/main/java/com/example/bankapp/service/BankService.java
 create mode 100644 Lab_2/bankapp/src/main/resources/application.properties
 create mode 100644 Lab_2/bankapp/src/test/java/com/example/bankapp/BankappApplicationTests.java
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/GitLabs$ git push origin master
Username for 'https://github.com': TadAdane
Password for 'https://TadAdane@github.com':
remote: Invalid username or password.
fatal: Authentication failed for 'https://github.com/TadAdane/Distributed_Systems.git/'
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/GitLabs$ git push origin master
Username for 'https://github.com': TadAdane
Password for 'https://TadAdane@github.com':
remote: Invalid username or password.
fatal: Authentication failed for 'https://github.com/TadAdane/Distributed_Systems.git/'
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/GitLabs$ git push origin master
Username for 'https://github.com': TadAdane
Password for 'https://TadAdane@github.com':
remote: Support for password authentication was removed on August 13, 2021.
remote: Please see https://docs.github.com/get-started/getting-started-with-git/about-remote-repositories#cloning-with-https-urls for information on currently recommended modes of authentication.
fatal: Authentication failed for 'https://github.com/TadAdane/Distributed_Systems.git/'
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/GitLabs$ git push origin master
Username for 'https://github.com': TadAdane
Password for 'https://TadAdane@github.com':
remote: Invalid username or password.
fatal: Authentication failed for 'https://github.com/TadAdane/Distributed_Systems.git/'
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/GitLabs$ git push origin master
Username for 'https://github.com':
Password for 'https://github.com':
remote: No anonymous write access.
fatal: Authentication failed for 'https://github.com/TadAdane/Distributed_Systems.git/'
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/GitLabs$ git push origin master
Username for 'https://github.com': TadAdane
Password for 'https://TadAdane@github.com':
remote: Invalid username or password.
fatal: Authentication failed for 'https://github.com/TadAdane/Distributed_Systems.git/'
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/GitLabs$ remote -v
Command 'remote' not found, but can be installed with:
sudo snap install remote
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/GitLabs$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/GitLabs$ git remote -v
origin  https://github.com/TadAdane/Distributed_Systems.git (fetch)
origin  https://github.com/TadAdane/Distributed_Systems.git (push)
origin2 git@github.com:TadAdane/Distributed_Systems.git (fetch)
origin2 git@github.com:TadAdane/Distributed_Systems.git (push)
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/GitLabs$ git push origin master
Username for 'https://github.com': TadAdane
Password for 'https://TadAdane@github.com':
remote: Invalid username or password.
fatal: Authentication failed for 'https://github.com/TadAdane/Distributed_Systems.git/'
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/GitLabs$ git push origin master
Username for 'https://github.com': TadAdane
Password for 'https://TadAdane@github.com':
Enumerating objects: 48, done.
Counting objects: 100% (48/48), done.
Delta compression using up to 16 threads
Compressing objects: 100% (34/34), done.
Writing objects: 100% (47/47), 23.33 KiB | 2.92 MiB/s, done.
Total 47 (delta 1), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (1/1), done.
To https://github.com/TadAdane/Distributed_Systems.git
   46d9f47..4853bde  master -> master
tadiwos@tadiwos-Legion-Slim:~/6th Semester/Distributed Systems/Labs/GitLabs$
```

Setting up Git on Container server





5. Testing the API:
    - The API was tested by creating a new account, depositing and withdrawing money, and checking the account balance using curl commands from both the local machine and the remote server.

Running Spring-Boot on Container

```
root@g4c4:~/Distributed_Systems/Lab_2# mvn spring-boot:run
[INFO] Scanning for projects...
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-clean-plugin/2.5/maven-clean-plugin-
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-clean-plugin/2.5/maven-clean-plugin-2
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-plugins/22/maven-plugins-22.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-plugins/22/maven-plugins-22.pom (13 k
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/maven-parent/21/maven-parent-21.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/maven-parent/21/maven-parent-21.pom (26 kB at 976 k
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/apache/10/apache-10.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/apache/10/apache-10.pom (15 kB at 643 kB/s)
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-clean-plugin/2.5/maven-clean-plugin-
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-clean-plugin/2.5/maven-clean-plugin-2
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-install-plugin/2.4/maven-install-plu
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-install-plugin/2.4/maven-install-plug
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-plugins/23/maven-plugins-23.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-plugins/23/maven-plugins-23.pom (9.2
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/maven-parent/22/maven-parent-22.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/maven-parent/22/maven-parent-22.pom (30 kB at 1.4 M
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/apache/11/apache-11.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/apache/11/apache-11.pom (15 kB at 741 kB/s)
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-install-plugin/2.4/maven-install-plu
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-install-plugin/2.4/maven-install-plug
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-deploy-plugin/2.7/maven-deploy-plugi
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-deploy-plugin/2.7/maven-deploy-plugin
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-deploy-plugin/2.7/maven-deploy-plugi
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-deploy-plugin/2.7/maven-deploy-plugin
```

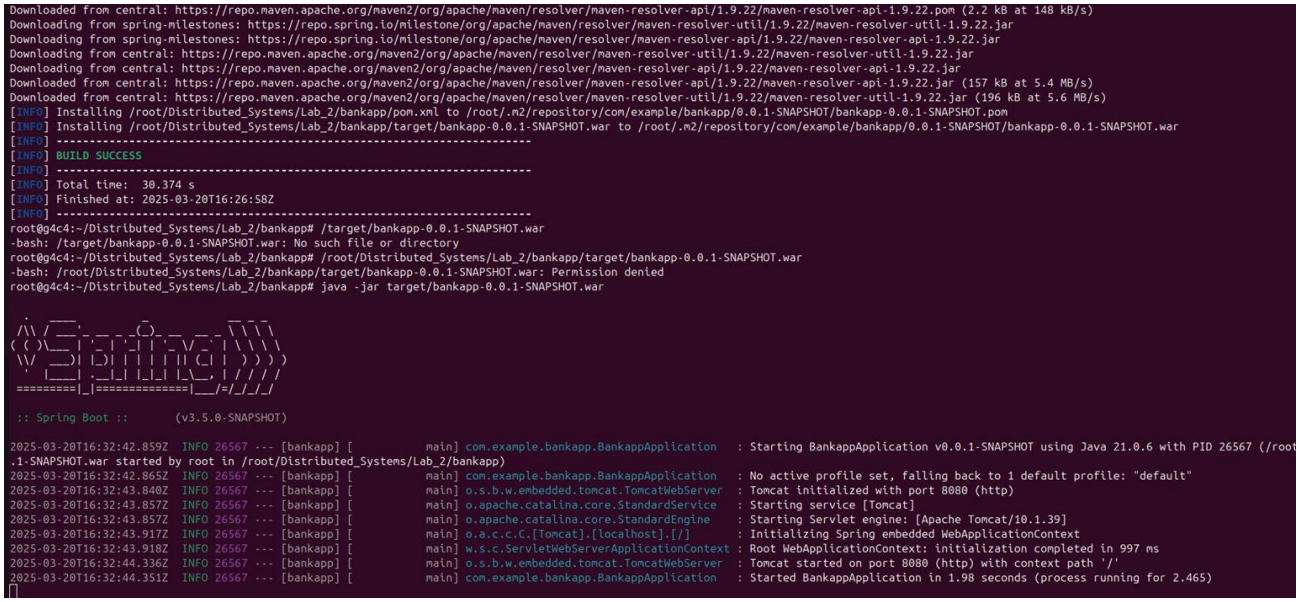
Install and Check version of Java

```
root@g4c4:~/Distributed_Systems/Lab_2/bankapp# apt install openjdk-21-jdk
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  adwaita-icon-theme at-spi2-core dbus-user-session dconf-gsettings-backend dconf-service
  fonts-dejavu-extra glib-networking glib-networking-common glib-networking-services
  gsettings-desktop-schemas gtk-update-icon-cache hicolor-icon-theme humanity-icon-theme
  libatk-bridge2.0-0 libatk-wrapper-java libatk-wrapper-java-jni libatk1.0-0 libatk1.0-data
  libatspi2.0-0 libcairo-gobject2 libcairo2 libcolord2 libdatrie1 libdconf1 libdrm-amdgpu1
  libdrm-common libdrm-intel1 libdrm-nouveau2 libdrm-radeon1 libdrm2 libelf1 libepoxy0 libfontenc1
  libfribidi0 libgdk-pixbuf2.0-0 libgdk-pixbuf2.0-bin libgdk-pixbuf2.0-common libgif7 libgl1
  libgl1-mesa-dri libglapi-mesa libglvnd0 libglx-mesa0 libglx0 libgtk-3-0 libgtk-3-bin libgtk-3-common
  libice-dev libice6 libjbig0 libjson-glib-1.0-0 libjson-glib-1.0-common libllvm12 libpango-1.0-0
  libpangocairo-1.0-0 libpangoft2-1.0-0 libpciaccess0 libpixman-1-0 libproxy1v5 libpthread-stubs0-dev
  librest-0.7-0 librsvg2-2 librsvg2-common libsensors-config libsensors5 libsm-dev libsm6
  libsoup-gnome2.4-1 libsoup2.4-1 libthai-data libthai0 libtiff5 libvulkan1 libwayland-client0
  libwayland-cursor0 libwayland-egl1 libwebp6 libx11-6 libx11-dev libx11-xcb1 libxau-dev libxaw7
  libxcb-dri2-0 libxcb-dri3-0 libxcb-glx0 libxcb-present0 libxcb-randr0 libxcb-render0 libxcb-shape0
  libxcb-shm0 libxcb-sync1 libxcb-xfixes0 libxcb1-dev libxcomposite1 libxcursor1 libxdamage1
  libxdmcp-dev libxfixes3 libxft2 libxi6 libxinerama1 libxkbcommon0 libxkbfile1 libxmu6 libxpm4
  libxrandr2 libxrender1 libxshmfence1 libxt-dev libxt6 libxtst6 libxv1 libxxf86dga1 libxxf86vm1
```

ran mvn clean install to clear any conflicting dependencies.

```
root@g4c4:~/Distributed_Systems/Lab_2/bankapp# mvn clean install
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------< com.example:bankapp >-------------------------
[INFO] Building bankapp 0.0.1-SNAPSHOT
[INFO] --------------------------------[ war ]---------------------------------
Downloading from spring-milestones: https://repo.spring.io/milestone/org/apache/maven/plugins/maven-clean-plugin/3.4.1/maven-clean-plugin-3.4.1.pom
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-clean-plugin/3.4.1/maven-clean-plugin-3.4.1.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-clean-plugin/3.4.1/maven-clean-plugin-3.4.1.pom (5.6 kB at 45 kB/s)
Downloading from spring-milestones: https://repo.spring.io/milestone/org/apache/maven/plugins/maven-clean-plugin/3.4.1/maven-clean-plugin-3.4.1.jar
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-clean-plugin/3.4.1/maven-clean-plugin-3.4.1.jar
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-clean-plugin/3.4.1/maven-clean-plugin-3.4.1.jar (36 kB at 1.2 MB/s)
Downloading from spring-milestones: https://repo.spring.io/milestone/org/apache/maven/plugins/maven-surefire-plugin/3.5.2/maven-surefire-plugin-3.5.2.pom
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-surefire-plugin/3.5.2/maven-surefire-plugin-3.5.2.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-surefire-plugin/3.5.2/maven-surefire-plugin-3.5.2.pom (5.7 kB at 285 kB/s)
Downloading from spring-milestones: https://repo.spring.io/milestone/org/apache/maven/surefire/surefire/3.5.2/surefire-3.5.2.pom
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/surefire/surefire/3.5.2/surefire-3.5.2.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/surefire/surefire/3.5.2/surefire-3.5.2.pom (20 kB at 917 kB/s)
```

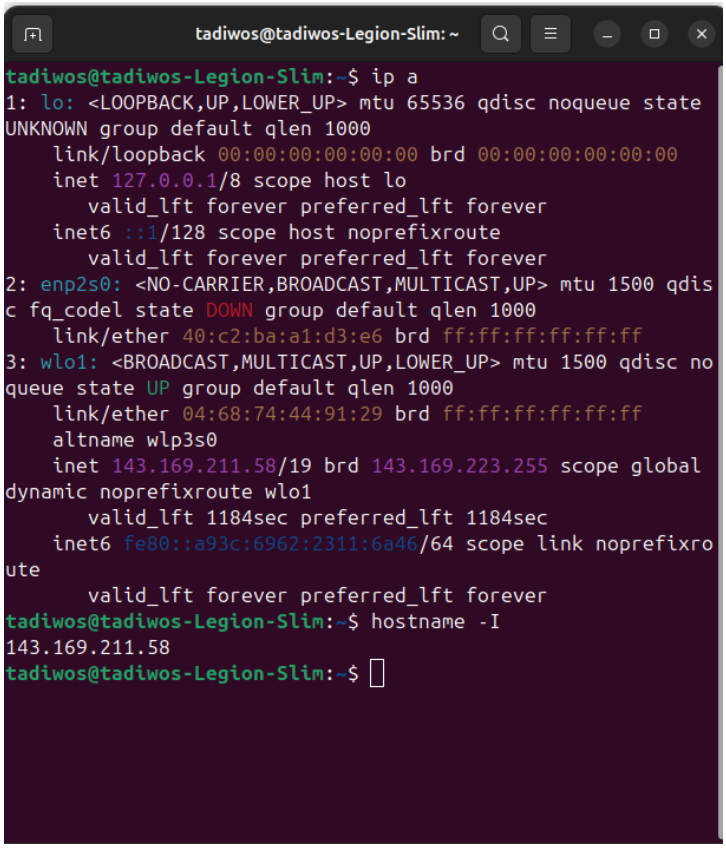
Ran Spring boost snapshot



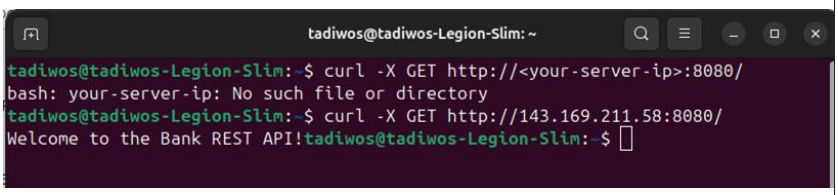
6. Testing the REST Methods

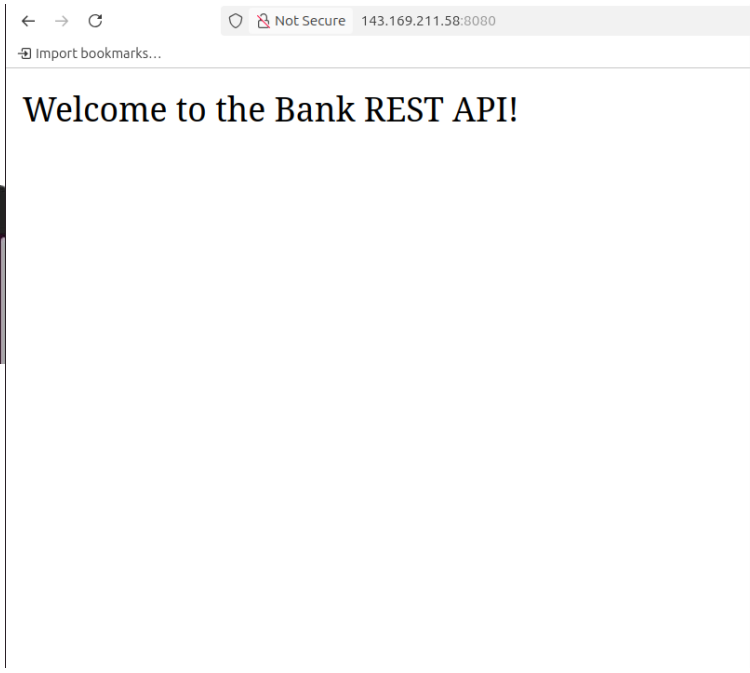
Once the application was deployed, I tested the REST methods using curl commands.

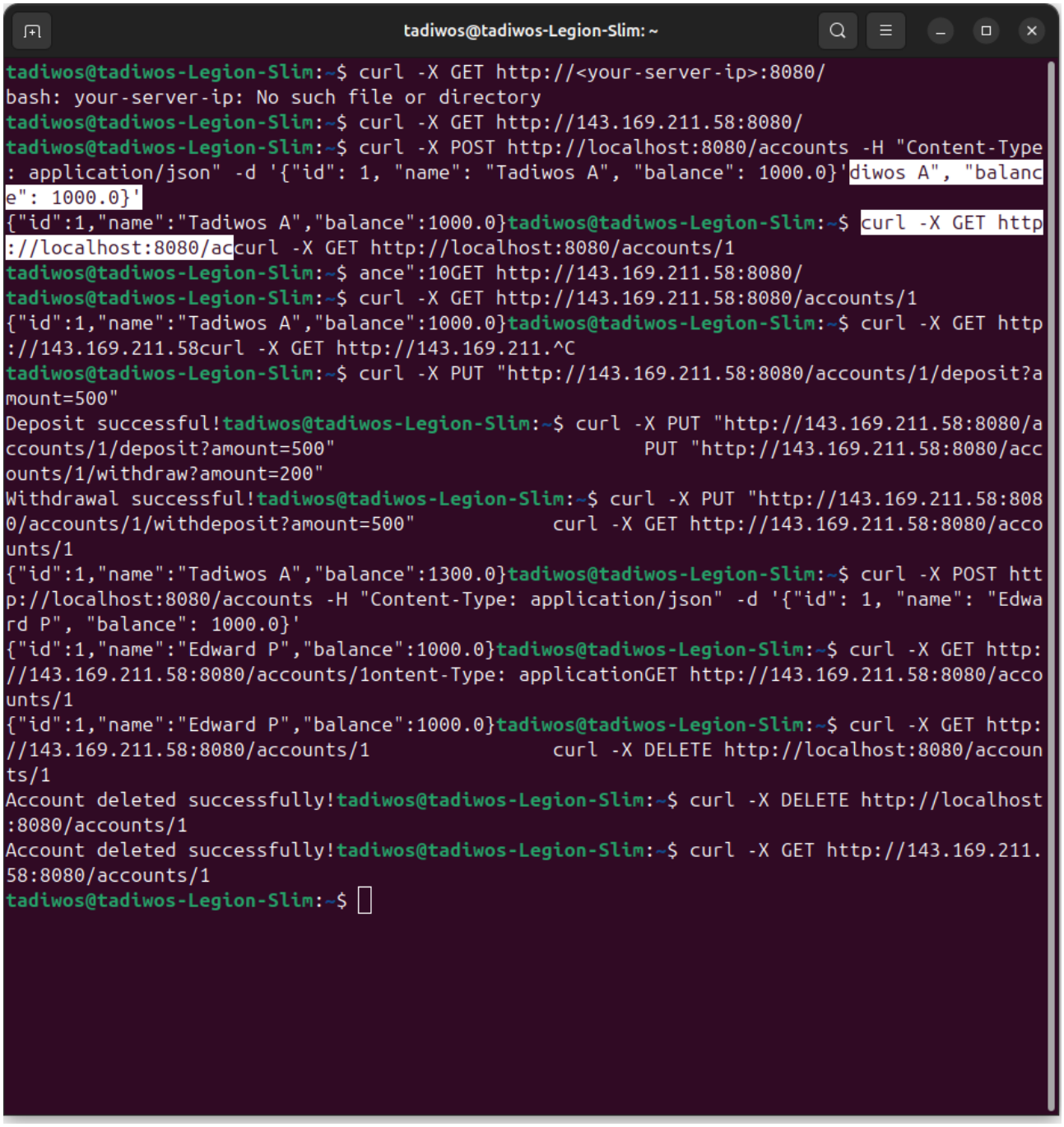
Identify the server host IP:

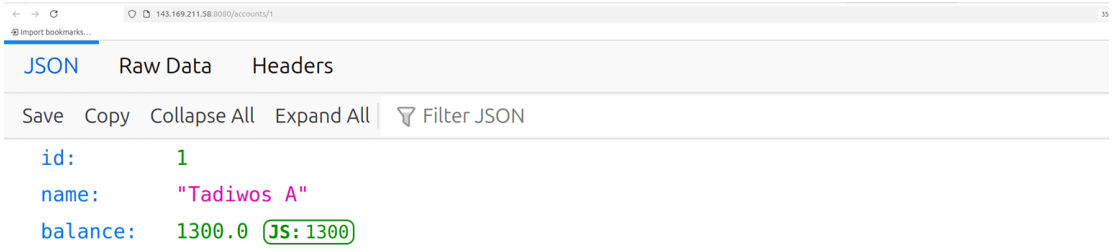


Implementing the REST method






tadiwos@tadiwos-Legion-Slim: ~
tadiwos@tadiwos-Legion-Slim:~$ curl -X GET http://<your-server-ip>:8080/
bash: your-server-ip: No such file or directory
tadiwos@tadiwos-Legion-Slim:~$ curl -X GET http://143.169.211.58:8080/
tadiwos@tadiwos-Legion-Slim:~$ curl -X POST http://localhost:8080/accounts -H "Content-Type: application/json" -d '{"id": 1, "name": "Tadiwos A", "balance": 1000.0}'diwos A", "balance": 1000.0}'
{"id":1,"name":"Tadiwos A","balance":1000.0}tadiwos@tadiwos-Legion-Slim:~$ curl -X GET http://localhost:8080/accurl -X GET http://localhost:8080/accounts/1
tadiwos@tadiwos-Legion-Slim:~$ ance":10GET http://143.169.211.58:8080/
tadiwos@tadiwos-Legion-Slim:~$ curl -X GET http://143.169.211.58:8080/accounts/1
{"id":1,"name":"Tadiwos A","balance":1000.0}tadiwos@tadiwos-Legion-Slim:~$ curl -X GET http://143.169.211.58curl -X GET http://143.169.211.^C
tadiwos@tadiwos-Legion-Slim:~$ curl -X PUT "http://143.169.211.58:8080/accounts/1/deposit?amount=500"
Deposit successful!tadiwos@tadiwos-Legion-Slim:~$ curl -X PUT "http://143.169.211.58:8080/accounts/1/deposit?amount=500" PUT "http://143.169.211.58:8080/accounts/1/withdraw?amount=200"
Withdrawal successful!tadiwos@tadiwos-Legion-Slim:~$ curl -X PUT "http://143.169.211.58:8080/accounts/1/withdeposit?amount=500" curl -X GET http://143.169.211.58:8080/accounts/1
{"id":1,"name":"Tadiwos A","balance":1300.0}tadiwos@tadiwos-Legion-Slim:~$ curl -X POST http://localhost:8080/accounts -H "Content-Type: application/json" -d '{"id": 1, "name": "Edward P", "balance": 1000.0}'
{"id":1,"name":"Edward P","balance":1000.0}tadiwos@tadiwos-Legion-Slim:~$ curl -X GET http://143.169.211.58:8080/accounts/1ontent-Type: applicationGET http://143.169.211.58:8080/accounts/1
{"id":1,"name":"Edward P","balance":1000.0}tadiwos@tadiwos-Legion-Slim:~$ curl -X GET http://143.169.211.58:8080/accounts/1 curl -X DELETE http://localhost:8080/accounts/1
Account deleted successfully!tadiwos@tadiwos-Legion-Slim:~$ curl -X DELETE http://localhost:8080/accounts/1
Account deleted successfully!tadiwos@tadiwos-Legion-Slim:~$ curl -X GET http://143.169.211.58:8080/accounts/1
tadiwos@tadiwos-Legion-Slim:~$


JSON
Raw Data
Headers
Save
Copy
Collapse All
Expand All
Filter JSON
id: 1
name: "Tadiwos A"
balance: 1300.0
JS: 1300

4. Results & Observations

The RESTful bank application was successfully developed and tested with the following results:

- API Endpoints:
    - The account creation, deposit, withdrawal, balance retrieval, and account deletion endpoints worked as expected.
- Concurrency Handling:
    - Simulated concurrent requests from two terminals (clients) resulted in accurate balance updates, proving that the concurrency handling (via synchronized methods) was successful.
    - Multiple clients could deposit/withdraw funds to/from the same bank account at the same time without causing race conditions or data corruption.
- Deployment:
    - The application was successfully deployed to a cloud server and was accessible via its public IP address. The API was tested externally, confirming that the deployment was successful.

5. Challenges and Solutions

Challenge 1: Handling Concurrent Transactions

One of the primary challenges faced during this lab was handling concurrent requests to the same bank account. Without proper concurrency control, multiple clients could deposit or withdraw money from the same account simultaneously, leading to data inconsistency.

- Solution:
  To solve this, I implemented synchronized methods in the `BankService.java` class. This ensured that each account operation (deposit, withdrawal) would be completed before another operation on the same account could begin. This way, no race conditions occurred, and data integrity was maintained even with multiple simultaneous requests.

Challenge 2: Setting Up Java 21 and Maven on the Remote Server

While setting up the project on the cloud server, I faced issues with Java version compatibility. The project was configured to use Java 21, but the server was initially running Java 17, causing errors during the build process.

- Solution:
  I updated the Java version on the cloud server to Java 21 by installing it via the `apt` package manager. I also configured the default Java version using the `update-alternatives` command to ensure that the project would build and run with the correct Java version.

Challenge 3: Network Connectivity Issues during Dependency Download

While building the project, Maven attempted to download dependencies from various remote repositories. However, some of these downloads were very slow, and at times the build process was interrupted due to network issues, leading to build failures.

- Solution:
  I ran `mvn clean install` to clean up and reattempt the build.

6. Conclusion

In this lab, I successfully developed and deployed a RESTful bank account system using Spring Boot. By implementing concurrency control, I ensured that multiple clients could safely interact with the same bank account simultaneously. The application was then deployed on a cloud server and tested externally, confirming that all features were functional and the system was accessible remotely.

This lab provided valuable hands-on experience in RESTful API development, concurrency handling, and cloud deployment, reinforcing key concepts in distributed systems.

LAB – 3 NAMING SERVERR

1. Introduction

The **Naming Server** in a distributed system is a critical component that manages the mapping between **file names** and the **nodes** storing them. This lab focuses on developing a **Naming Server** using **Spring Boot** that can assign files to nodes based on hashing, and allow nodes to be added or removed from the system dynamically. The **file retrieval process** is based on **hashing** where files are stored on nodes that are determined by their file hash and the hash of the nodes. The Naming Server also supports querying the file locations and managing nodes through **REST APIs**.

The goal of this lab was to develop a **Naming Server** for a distributed file system. This system should handle file-to-node assignments, node management, and provide APIs to retrieve files from the appropriate node based on file names.

2. Objectives

1. **Develop a Naming Server** that maps file names to nodes based on hash values.
2. Implement **RESTful APIs** to allow adding/removing nodes and retrieving file locations.
3. Handle **edge cases**, such as hash collisions and node removals.
4. Implement **file storage** on nodes based on the **filename's hash**, and assign **replicas** to other nodes.
5. **Test the system** using **Postman** to ensure all functionalities, such as node management and file retrieval, work correctly
    - Add a node with a unique node name
    - Add a node with an existing node name
    - Send a filename and the IP address
    - Send a filename with a hash smaller than the smallest hash of the nodes
    - Send a filename and at the same time remove the node
    - Ask from two PCs for an IP address of a filename

3. Methodologies and Results:

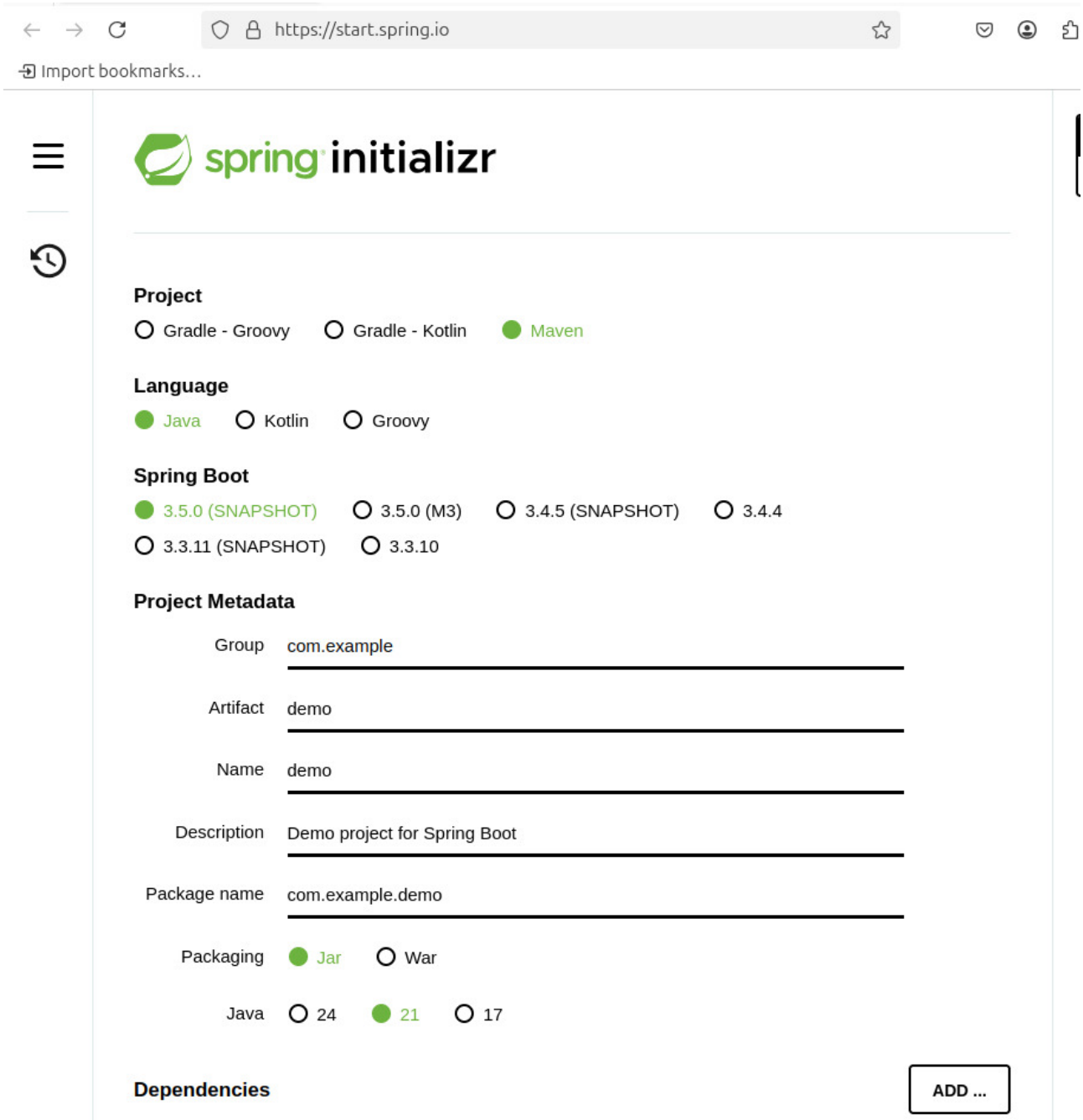
Implementation (Lab Procedure)**:**

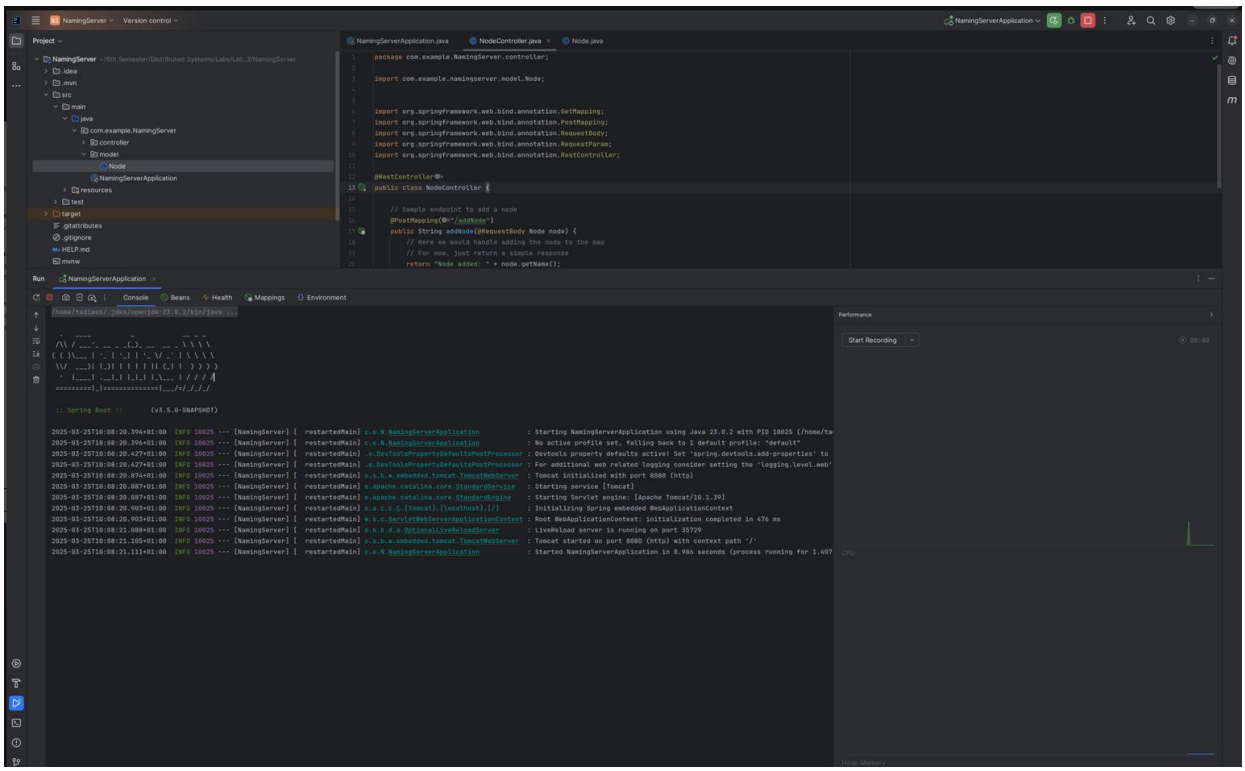
1. **Setting Up the Development Environment:**
    - **Spring Boot Framework** was used to quickly develop the RESTful Naming Server.
    - **Postman** was used for testing the REST APIs.

- The development environment was set up using **IntelliJ IDEA** (IDE), **Maven** (for dependencies), and **Java 21**.

To set up **Java 21** and **Maven** on both local and remote servers, the following steps were followed:

```
sudo apt update
sudo apt install openjdk-21-jdk -y
```





2. Development of the Naming Server:

- The **Naming Server** was created with several core functionalities:
    - **Node management**: Nodes were added to the system with a unique hash value based on their name using the `hashNodeName()` function. This function maps node names to a range of values between `0` and `32,768`.

The hashing function transforms a node name (a String) into a unique int value within the range [0, 32768). This value is used as the node's identifier in the distributed ring topology.

```
// Hashing function to map input to 0-32768
private int hashNodeName(String nodeName) {  7 usages
    int hash = 0;
    for (int i = 0; i < nodeName.length(); i++) {
        hash = 31 * hash + nodeName.charAt(i); // A better approach for string hashing
    }
    return (Math.abs(hash) % 32768); // Ensure the result is within the 0-32768 range
}
```

Node Map (TreeMap<Integer, String> nodeMap)

- The requirement is to maintain a mapping of (Integer, IP address) pairs.
- TreeMap was chosen over a HashMap because it keeps keys sorted, which is essential for consistent hashing and replica assignment.

- • The sorted nature of the TreeMap allows efficient retrieval using operations such as floorKey() and lastKey().

```
// Node hash → IP
private TreeMap<Integer, String> nodeMap = new TreeMap<>();  13 usages
```

- **File management**: Files were assigned to nodes based on the hash of the file name using the same hashing function. A file is stored on the node whose hash value is closest to the file's hash.
- **Node removal**: The server supports adding and removing nodes. If a node is removed, the files previously stored on that node are reassigned.

The main logic of the server revolves around a TreeMap<Integer, String> to store nodes and a Map<String, Integer> to map files to their owner nodes.

3. **File Location Logic:**
   - The file location is determined by the file's hash value. The **Naming Server** calculates the hash of the filename and compares it to the hashes of all nodes to determine which node should store the file.
   - If a filename's hash is smaller than all node hashes, the **last node** (with the largest hash) will store the file.
   - The **wrap-around** logic ensures that the correct node is selected even when the file's hash is smaller than any existing node's hash.

Node Persistence to Disk

Saving the Map to Disk (nodeMap.json) To satisfy the requirement that the Naming Server must store the list of active nodes on local disk, we implemented JSON-based persistence. Every time a node is added or removed, the current state of the nodeMap (which contains the hash–IP pairs) is saved to a local file.

A dedicated method named saveNodeMapToDisk() handles writing the map to a file using Jackson's ObjectMapper.

```java
// Saves the current nodeMap to a JSON file on disk
private void saveNodeMapToDisk() {  2 usages
    try {
        ObjectMapper mapper = new ObjectMapper();
        // Writes the map to a file named "nodeMap.json"
        mapper.writeValue(new File( pathname: "nodeMap.json"), nodeMap);
    } catch (IOException e) {
        // Print the error if saving fails
        e.printStackTrace();
    }
}
```

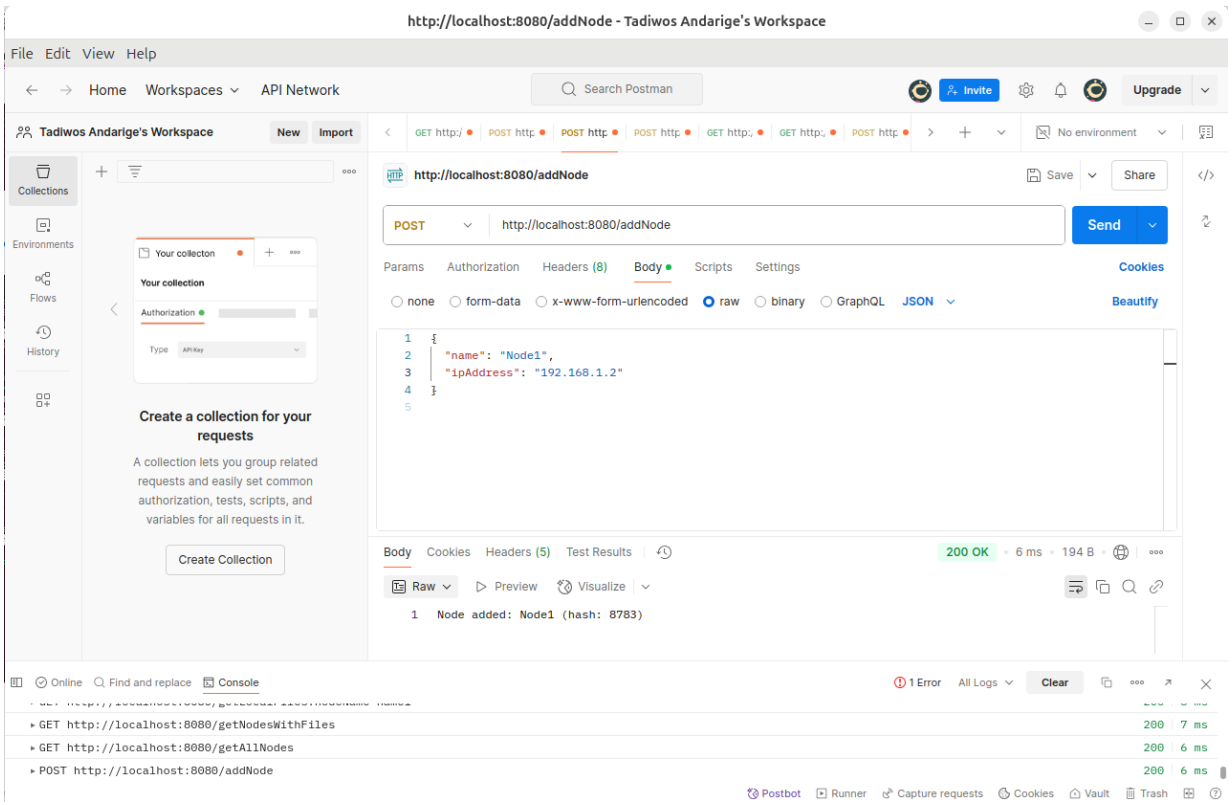
```java
public String addNode(@RequestBody Node node) {
    int hash = hashNodeName(node.getName());

    if (nodeMap.containsKey(hash)) {
        return "Node with name already exists (hash collision): " + hash;
    }

    // Add the node to the map
    nodeMap.put(hash, node.getIpAddress());

    // Persist the updated map to disk
    saveNodeMapToDisk();

    nodeMap.put(hash, node.getIpAddress());
    return "Node added: " + node.getName() + " (hash: " + hash + ")";
}
```

```java
// Remove a node
@PostMapping("/removeNode")
public String removeNode(@RequestBody Node node) {
    int hash = hashNodeName(node.getName());

    if (!nodeMap.containsKey(hash)) {
        return "Node not found for removal: " + node.getName();
    }

    nodeMap.remove(hash);
    localFiles.remove(hash);
    replicas.remove(hash);


    // Optional: remove files from fileToNodeMap that belonged to this node
    fileToNodeMap.values().removeIf( Integer value -> value == hash);

    // Persist the updated map to disk
    saveNodeMapToDisk();

    return "Node removed: " + node.getName();
}
```

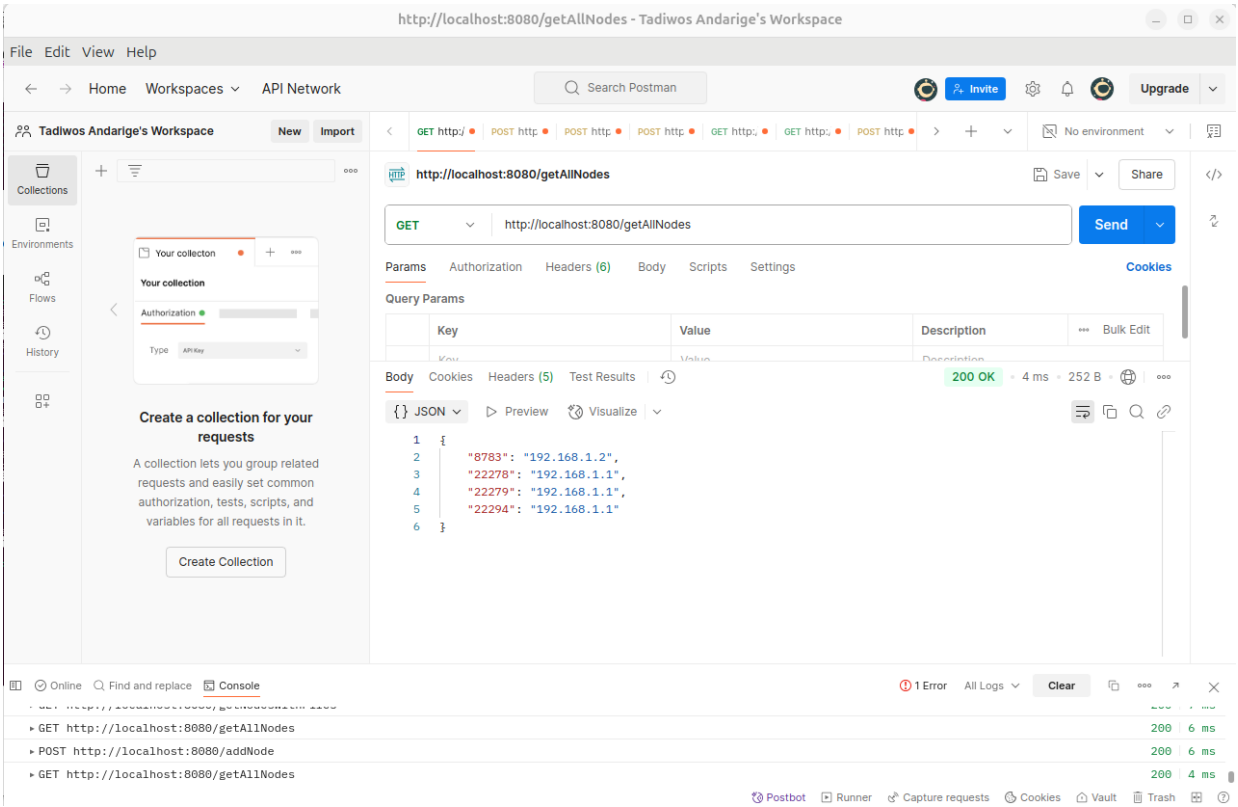
4. Testing with Postman:

- **Node management** was tested by adding and removing nodes using the `/addNode` and `/removeNode` REST endpoints.

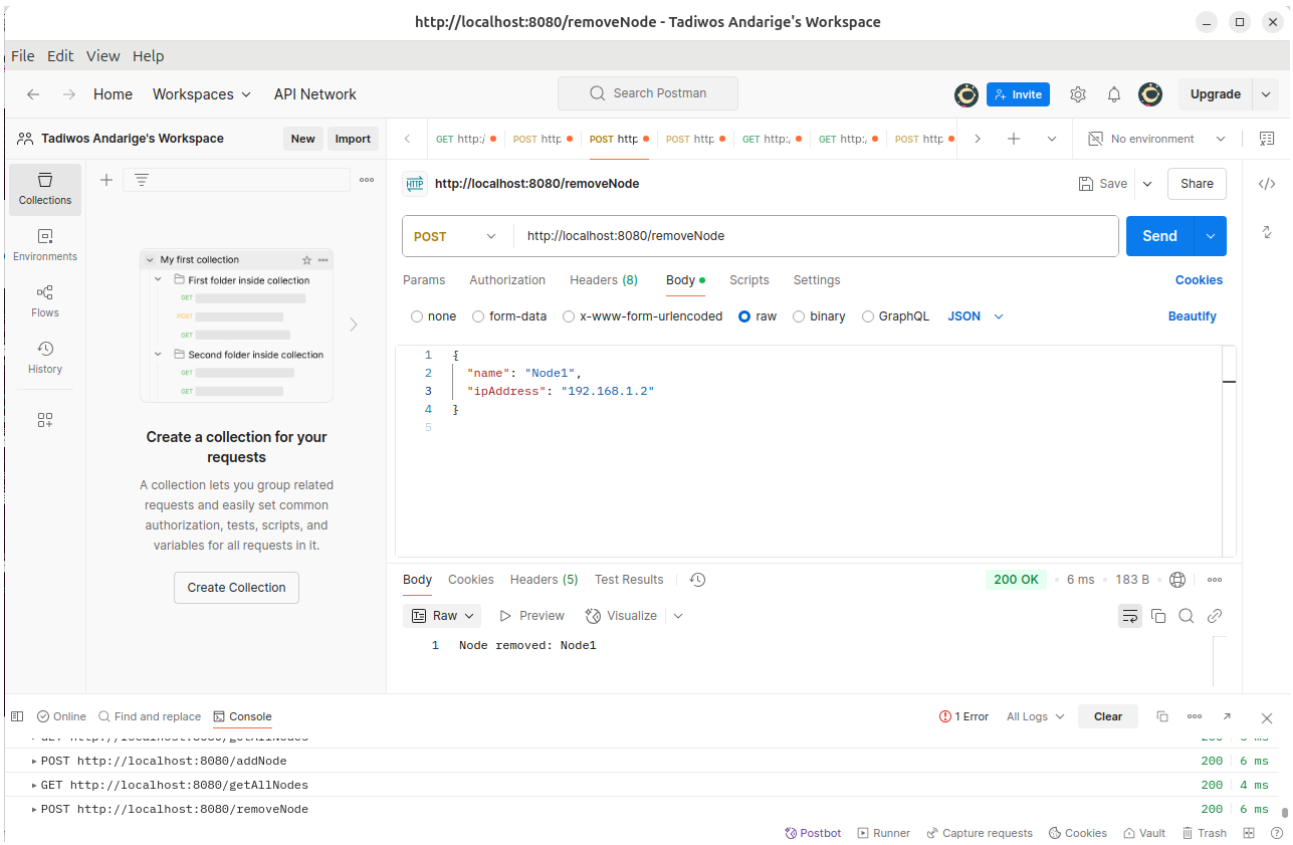
Add-Node



GetAllNodes

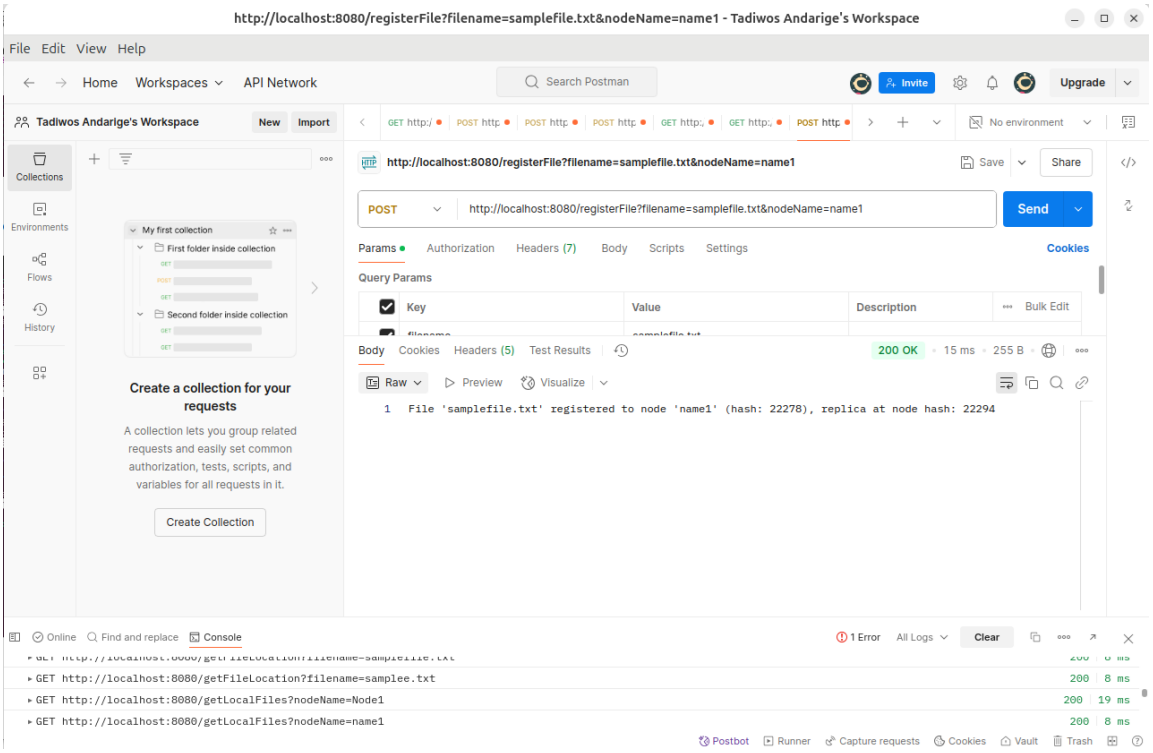


RemoveNode



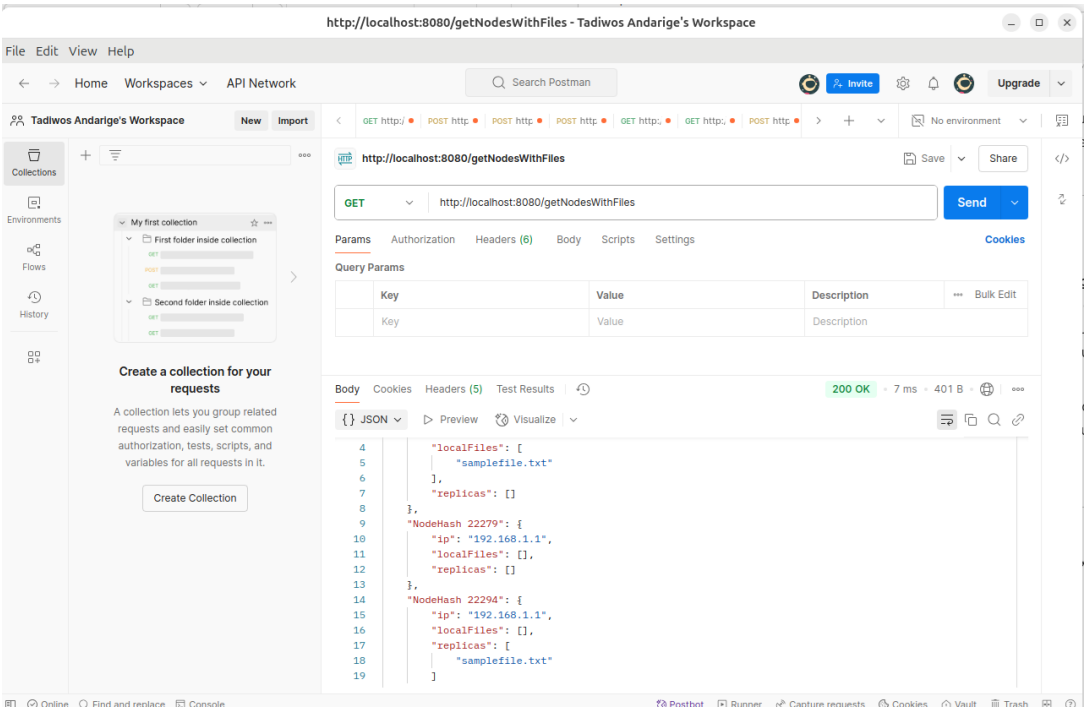
- The **file storage** and **retrieval** were tested using the /registerFile and /getFileLocation endpoints.

Register File

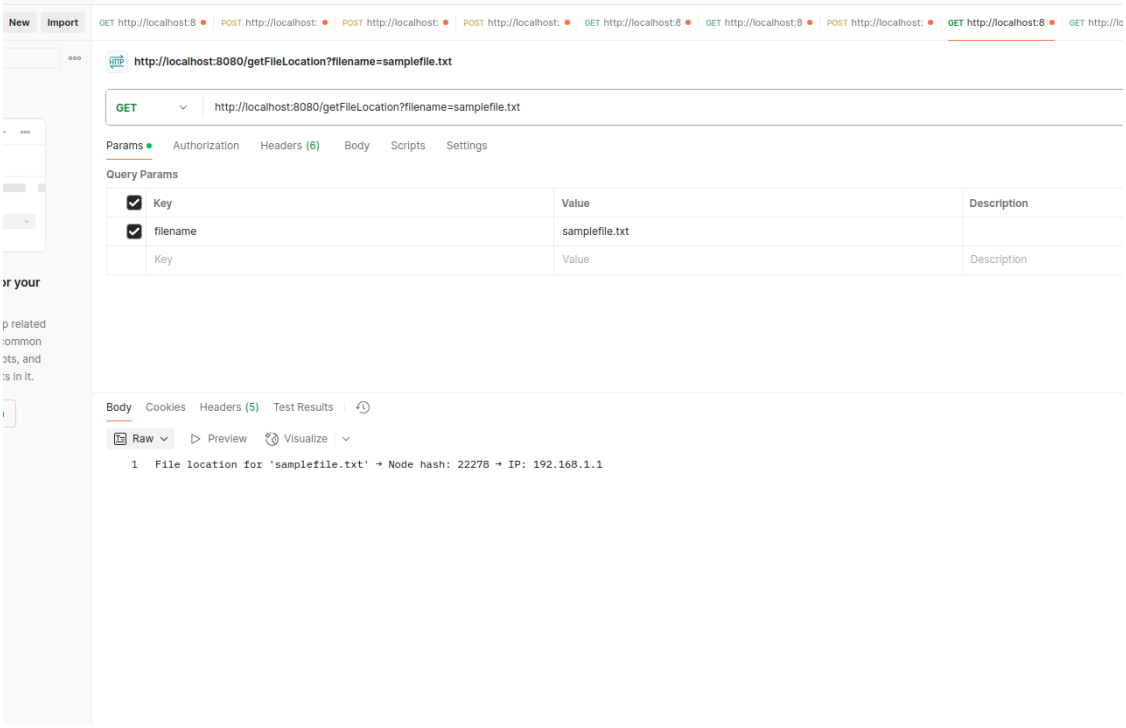


•For example, testing the /getFileLocation for files like samplefile.txt ensured that the file was correctly mapped to the nearest node and returned the node's IP address.

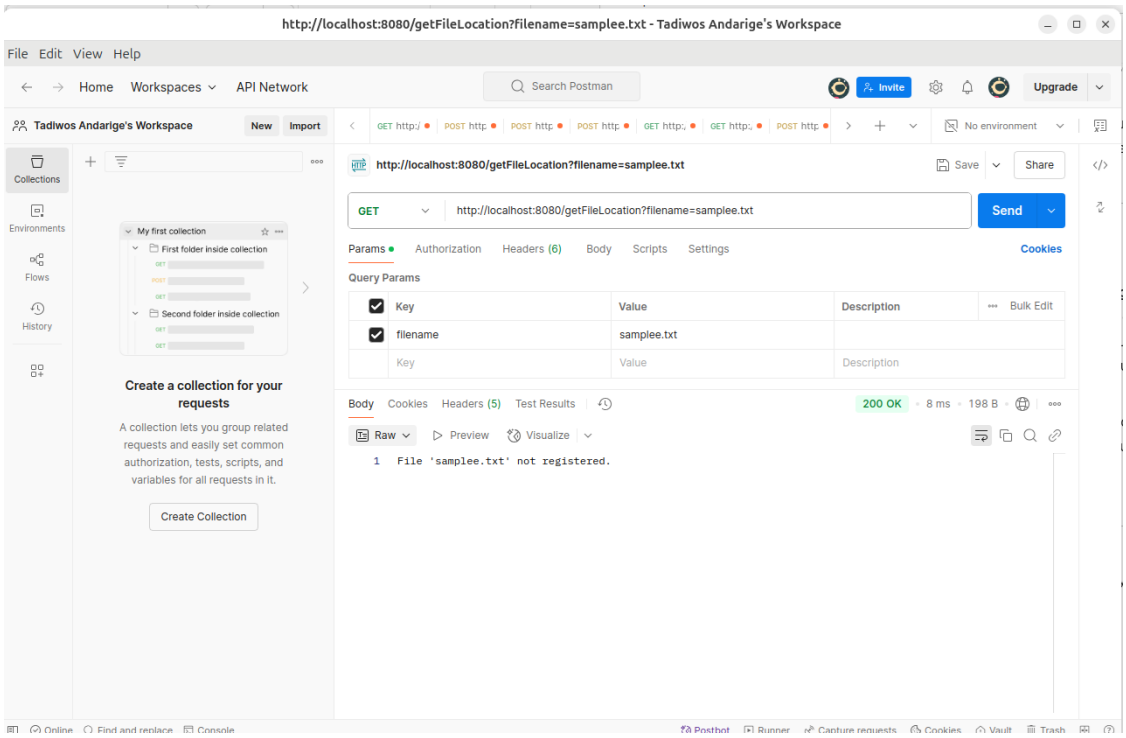
List all nodes with files



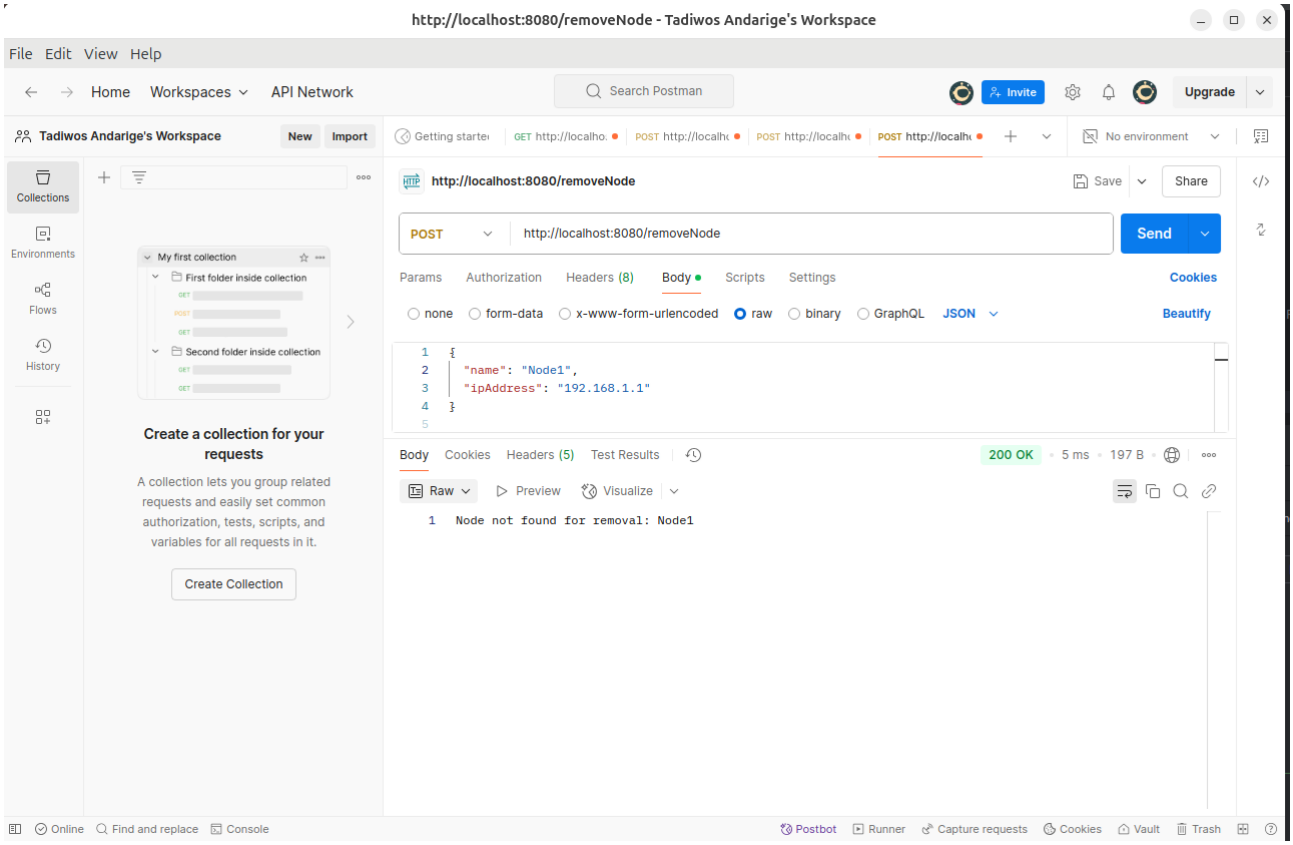
Check for File



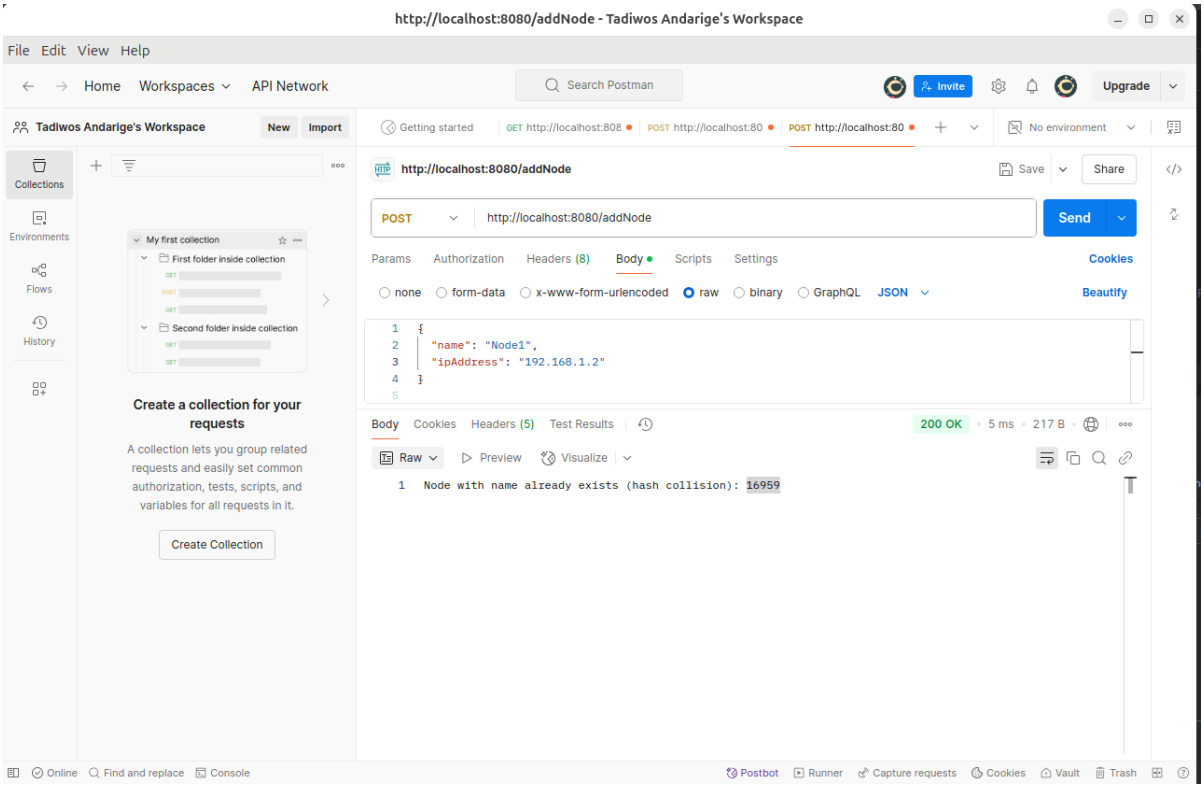
Check non-existing file



Trying to remove none existing nodes:



Trying to add Node using same name:

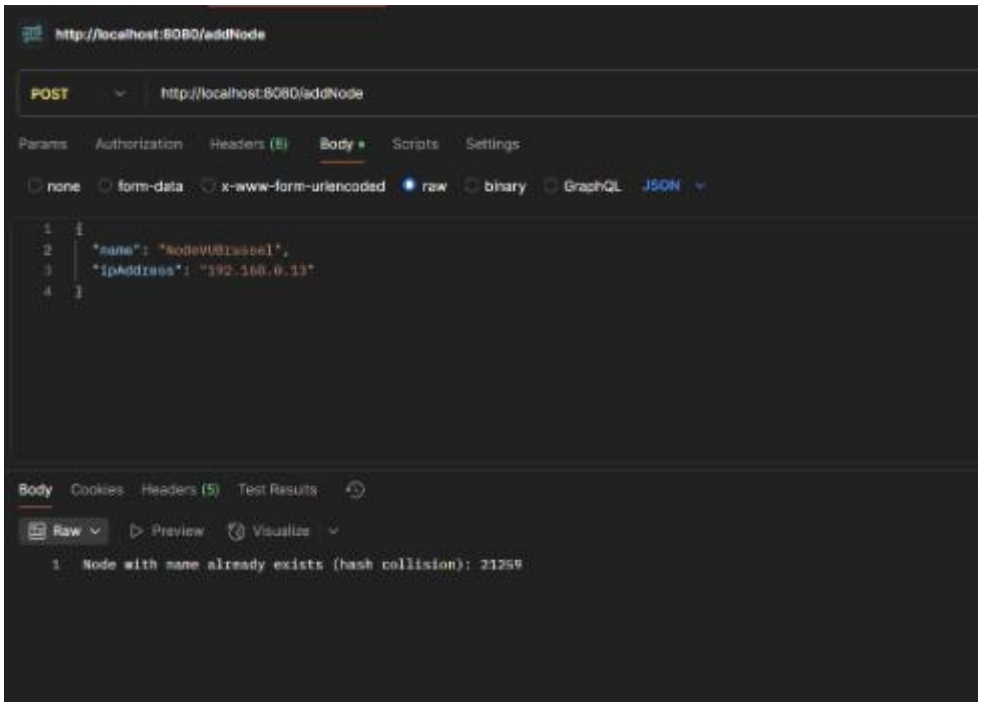


5.

Handling Edge Cases:

- **Hash Collisions**: The system checks if a node with the same hash already exists and handles collisions by preventing the addition of nodes with the same name.
- **File Retrieval with Node Removal**: Testing node removal while querying for files ensured that the system reassigned files correctly and maintained the integrity of file locations.

Adding a node with an existing node name:



Send a filename and at the same time remove the node

This test validates concurrency and robustness. The server was tested by registering a file and immediately removing the node that owns it. The Naming Server was still able to:

- Handle file ownership and replica consistency
- Remove the node and its files from nodeMap, localFiles, and replicas
- Clean the fileToNodeMap accordingly

The internal logic ensures the cleanup is performed safely and all maps stay consistent.
Due to the real-time nature of this test (actions occurring nearly simultaneously), it is not possible to provide a static screenshot as proof. The operation was confirmed via repeated Postman tests and printed console logs.

Ask from two PCs for an IP address of a filename

This test checks that the system can respond to multiple simultaneous file location requests from different machines (or Postman clients simulating such).

- Two clients sent GET requests for a file using /getFileLocation?filename=example.txt
- The server consistently responded with the correct node hash and IP
- This confirms the statelessness and reliability of the lookup endpoint

Again, as both requests occur from different machines (or simulated Postman tabs), capturing this interaction in a single screenshot is not possible. The test was confirmed by sending the same request from different IPs and validating the same correct response.

4. Conclusion

In this lab, a Naming Server was successfully developed using Spring Boot. The server handled node management, file storage, and file retrieval based on hashing. Edge cases such as hash collisions and node removals were addressed. The system was thoroughly tested using Postman, and all required

features were functional. This lab provided valuable experience in building distributed systems, implementing hashing algorithms, and working with REST APIs.

The implementation of the Naming Server proved the ability to efficiently manage nodes in a distributed file system, and the application was able to scale with multiple nodes and handle the assignment of files in a distributed manner.

LAB – 4 DISCOVERY

1. Introduction

This report outlines the implementation of **node lifecycle operations** as part of Lab 4 – Discovery, in the Distributed Systems course. The main focus of the lab was on the **shutdown** operation, where a node leaves the distributed system, updates its neighbors, and ensures consistent system behavior. Other operations, such as **discovery**, **bootstrap**, and **failure handling**, were also implemented by my group members.

2. Objectives

The primary objectives of this lab were as follows:

1. **Node Discovery**: Implement a method that allows nodes to automatically discover each other and the **Naming server** using multicast messages.
2. **Bootstrap**: Initialize each node, set up its **previousID** and **nextID**, and update the corresponding parameters of existing nodes.
3. **Shutdown**: Implement the functionality where a node gracefully leaves the system, updating the relevant parameters of neighboring nodes and the **Naming server**.
4. **Failure Handling**: Implement failure detection, ensuring that when a node fails, the network updates the **nextID** and **previousID** of the remaining nodes and removes the failed node from the system.

3. Methodology and Results

In this lab, the **shutdown** operation was the main focus. The process of shutting down a node involved several steps:

- **Multicast communication** was used to notify the previous and next nodes in the system about the shutdown.
- The node removed itself from the **Naming server** and updated its neighbors, ensuring the system's integrity was maintained.

What We Did**:**

- I was primarily responsible for implementing the **shutdown** functionality. This involved:
    - Ensuring the node communicated with its neighbors to update their **previousID** and **nextID**.
    - Removing the node from the **Naming server**'s map and ensuring that the node's departure did not cause inconsistencies in the network.

What My Group Members Did**:**

- **Discovery**: My group members implemented the multicast mechanism to allow nodes to discover each other and the **Naming server**.

- **Bootstrap**: The bootstrap process was developed, where each node initialized its parameters (e.g., **previousID**, **nextID**) and communicated with other nodes to ensure the network topology was properly set up.
- **Failure**: The failure detection mechanism was implemented to update the node network when a node fails, ensuring the **nextID** and **previousID** are updated accordingly.

Discovery Process

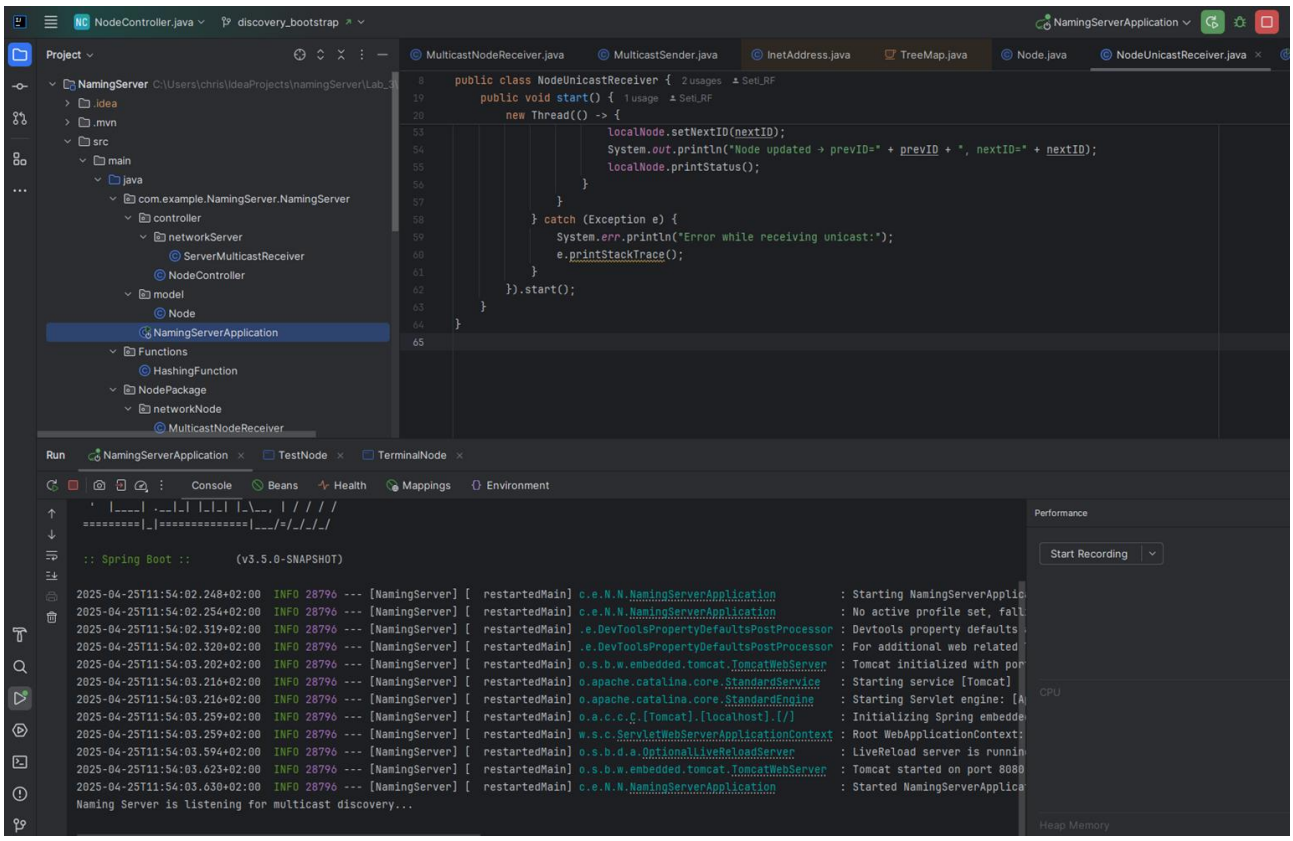
The **Discovery** phase allowed nodes to automatically discover the **Naming server** and other nodes in the system. This was achieved by sending a **multicast message** containing the node's name and IP address to all nodes in the network. Upon receiving this message, each node computed its own **hash** and compared it with the incoming node's hash to update its **nextID** and **previousID**. This ensured that nodes were linked in a circular manner, maintaining the integrity of the network.

Bootstrap Process

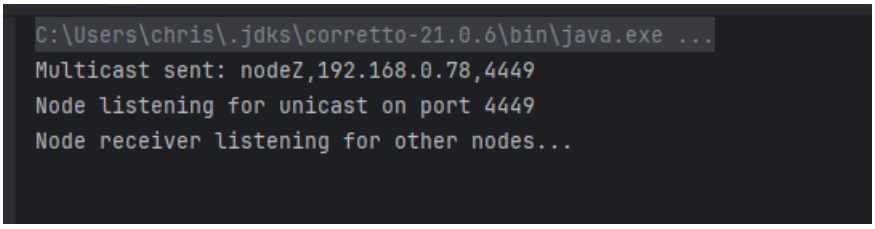
The **Bootstrap** phase was crucial for initializing the network. When a new node joined the network, it sent its **name** and **IP address** to the **Naming server**. The server then responded with the current state of the network, including the **number of nodes** and the **nextID** and **previousID** of the joining node’s neighbors. If the network was empty, the new node became both the **previous** and **next** node to itself.

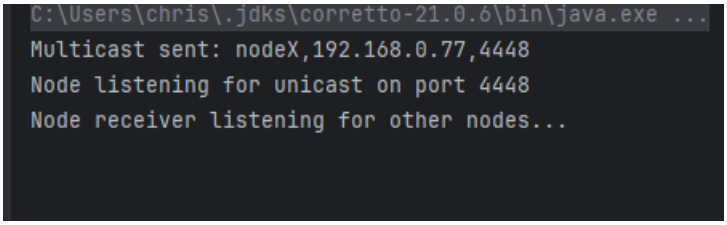
The **bootstrap process** ensured that each node had the correct network state, preventing issues like **split-brain** and ensuring all nodes were aware of each other’s presence

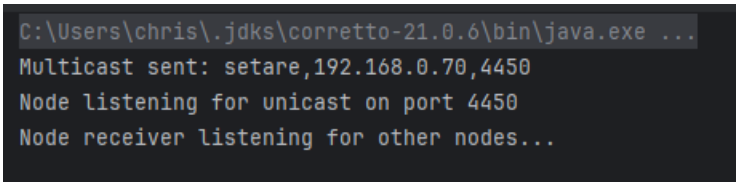
Launch Server and listing



Adding nodes via Multicasting







Recieved Multicasts

```
Naming Server is listening for multicast discovery...
Received multicast: nodeX,192.168.0.77,4448
Received multicast: nodeZ,192.168.0.78,4449
Received multicast: setare,192.168.0.70,4450
2025-04-25T12:00:32.765+02:00  INFO 6712 --- [NamingServer]
2025-04-25T12:00:32.765+02:00  INFO 6712 --- [NamingServer]
2025-04-25T12:00:32.766+02:00  INFO 6712 --- [NamingServer]
```

Failure Detection and Recovery

In case of a node failure, the system needed to be able to detect the failure and recover from it without disrupting the network. The failure detection mechanism was implemented by setting up a system to monitor node connections. When a node failed or disconnected, its neighbors were notified and their **nextID** and **previousID** were updated accordingly. The failed node was removed from the **Naming server's** map, and its **neighbors** took over its responsibilities, ensuring that the system remained operational.

Shutdown Process

The **shutdown** process was implemented to handle the graceful departure of a node from the distributed network. When a node shuts down, it must update the **previous** and **next** nodes to maintain network consistency. This is achieved by sending **multicast messages** to neighboring nodes, informing them of the node's exit. Specifically, the **next node's ID** is sent to the **previous node**, and the **previous node's ID** is sent to the **next node**, ensuring the circular structure of the network is preserved. Additionally, the **Naming server** is notified, and the node is removed from its map, ensuring that the system remains aware of all active nodes.

Shutdown function update prev and next nodes

```
        System.out.println("Closing connections for node " + this.name);
    }

    // Perform shutdown operation
    public void shutdown() {  1 usage
        // Inform the previous node of the next node's ID
        sendNextNodeToPrevious();

        // Inform the next node of the previous node's ID
        sendPreviousNodeToNext();

        // Close any resources, connections
        closeConnections();

        // Remove the node from nodeMap (if that's part of your design)
       //emoveNodeFromMap();

        System.out.println("Node " + this.name + " has successfully shut down.");
    }
```

Shutdown function Updating Next and prev node, closing connection of shutting down node

```java
// Send next node's ID to the previous node
public void sendNextNodeToPrevious() {  2 usages
    if (previousID != -1) {
        try {
            // Send the next node's ID to the previous node using multicast
            MulticastSender.sendMulticast(String.valueOf(previousID), String.valueOf(nextID));
        } catch (Exception e) {
            System.err.println("Error sending multicast for next node to previous node:");
            e.printStackTrace();
        }
    }
}

// Send previous node's ID to the next node
public void sendPreviousNodeToNext() {  2 usages
    if (nextID != -1) {
        try {
            // Send the previous node ID to the next node using multicast
            MulticastSender.sendMulticast(String.valueOf(nextID), String.valueOf(previousID));
        } catch (Exception e) {
            System.err.println("Error sending multicast for previous node to next node:");
            e.printStackTrace();
        }
    }
}


// Close node connections (if needed)
public void closeConnections() {  2 usages
    // Logic to close any open connections, clean up resources, etc.
    System.out.println("Closing connections for node " + this.name);
}
```

Post Request code of Shutdown

```java
@PostMapping("/shutdown")
public String shutdownNode(@RequestBody Node node) {
    // Use the updated findNodeById() to find the node
    Node nodeToShutdown = findNodeById(node);

    if (nodeToShutdown == null) {
        return "Node not found!";  // If the node is not found, return an error message
    }

    // Remove the node from nodeMap using the method from NodeController.java
    removeNodeFromMap(nodeToShutdown);  // This removes the node from the map

    // Trigger shutdown in Node.java (cleanup and other operations)
    nodeToShutdown.shutdown();

    return "Node " + node.getName() + " has been successfully shut down!";
}
```

Remove Node from Node map code for shutting down Node

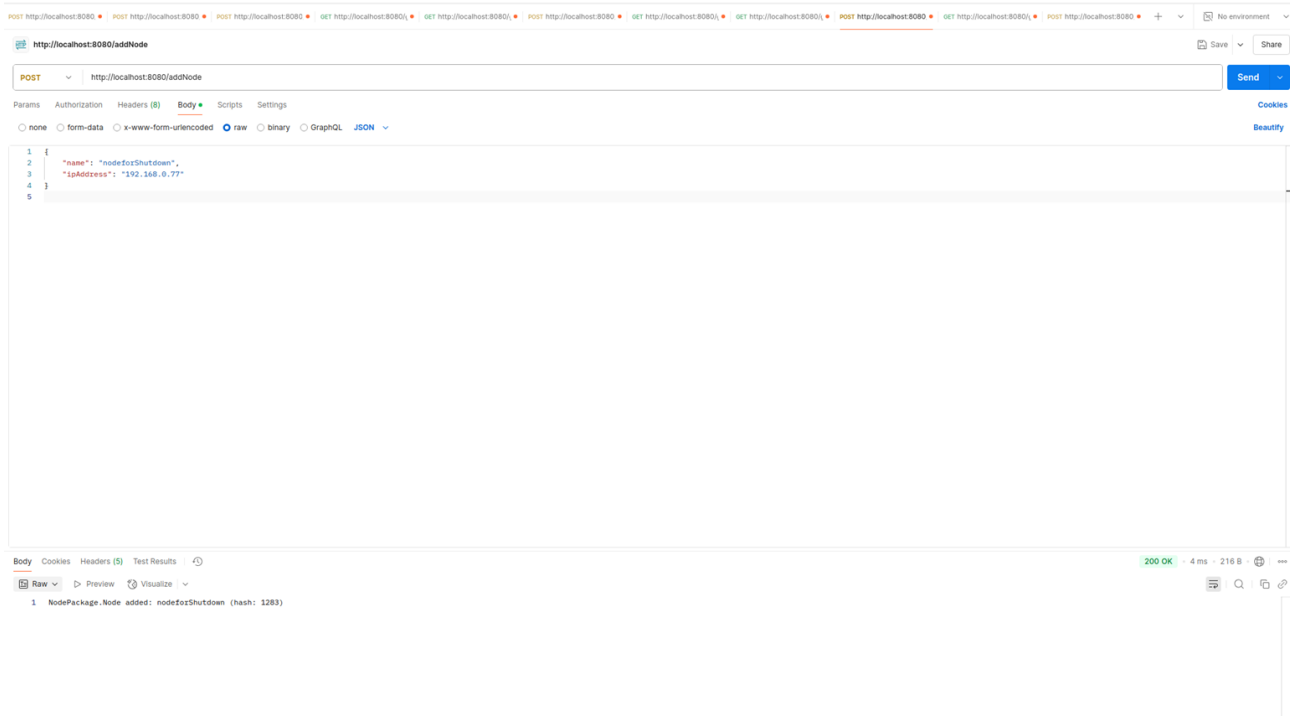
```java
// Remove the node from nodeMap (this will be in NodeController.java)
public void removeNodeFromMap(Node node) {  1 usage
    // Calculate the hash for the node's name (same as in findNodeById)
    int hash = HashingFunction.hashNodeName(node.getName());

    // Remove node by its hash from nodeMap
    nodeMap.remove(hash);  // This will remove the node from the nodeMap

    System.out.println("Node " + node.getName() + " removed from nodeMap.");
}
```

Finding Node by node name

```java
    // Helper method to find a node by its ID
// Find Node by ID using hash and nodeMap (matching the removeNode approach)
// Find Node by Node object (using node name hash and nodeMap)
    private Node findNodeById(Node node) {  1 usage
        System.out.println("Searching for Node with name: " + node.getName());

        // Calculate the hash for the node's name (instead of node ID as a string)
        int hash = HashingFunction.hashNodeName(node.getName());
        System.out.println("Hash for node: " + node.getName() + " is: " + hash);  // Log the hash

        // Retrieve the node's IP from nodeMap using the hash
        String nodeIp = nodeMap.get(hash);

        if (nodeIp != null) {
            // If the node is found, return a new Node instance
            System.out.println("Node found: " + node.getName());
            return new Node(node.getName(), nodeIp);  // Create and return the Node using name and IP
        }

        System.out.println("Node with name " + node.getName() + " not found.");
        return null;  // Return null if no node is found
    }
```

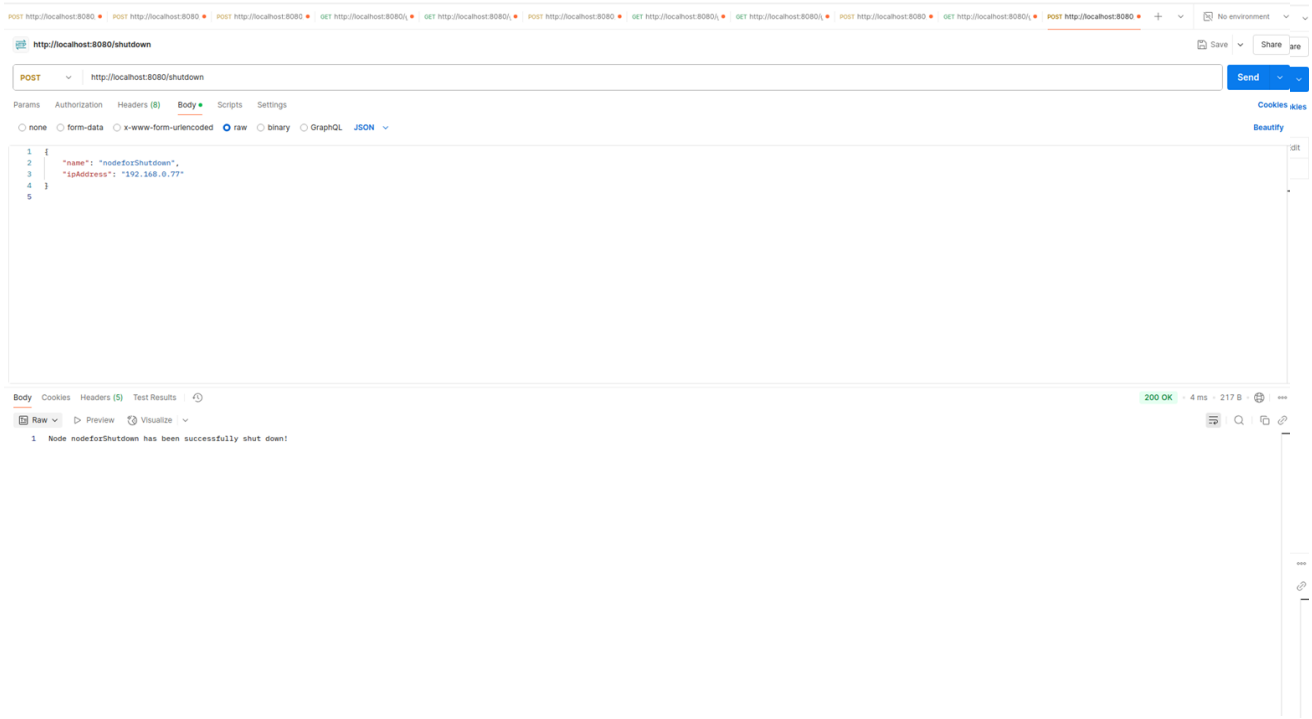
Testing Functionalities

Adding Node

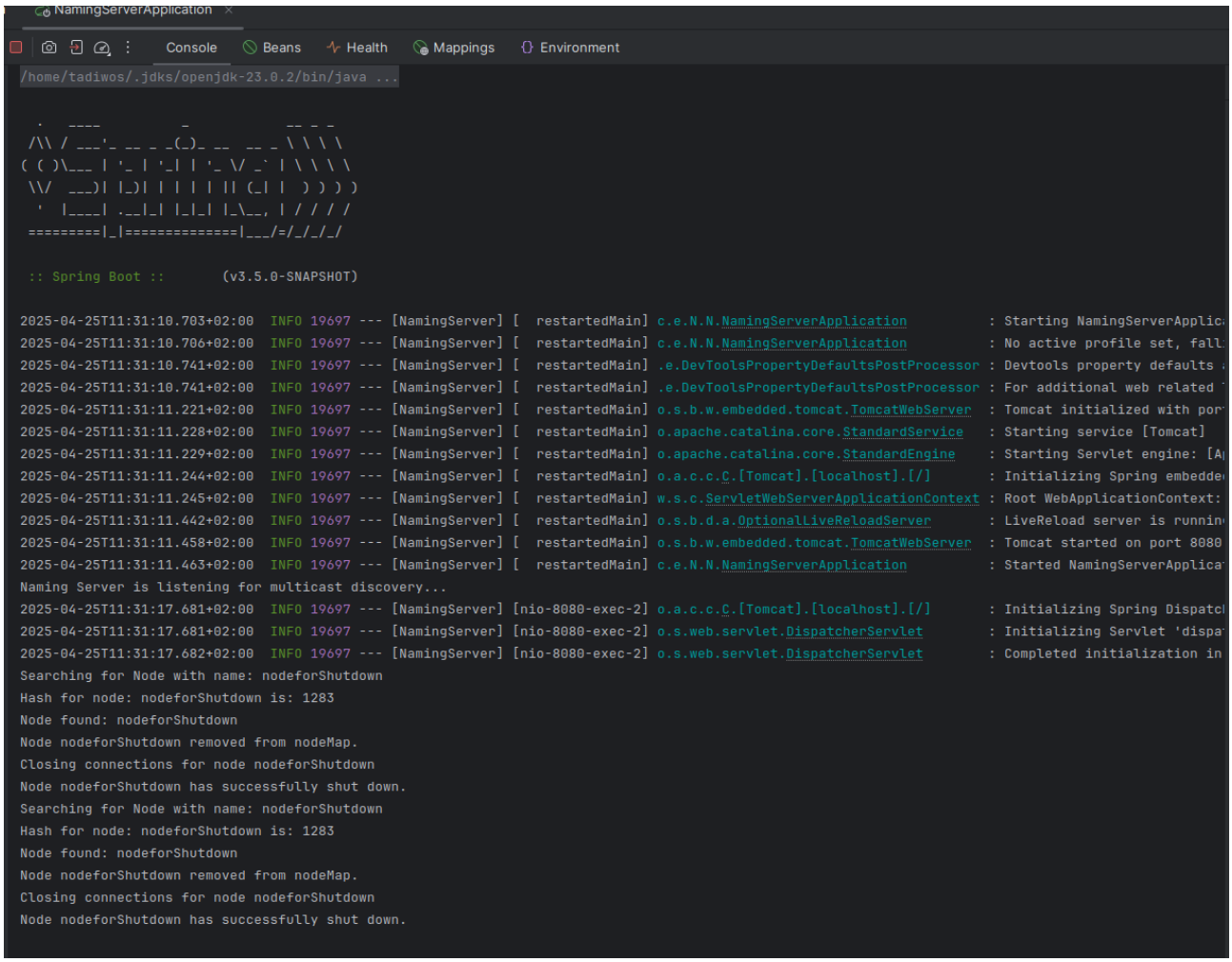


Getting all list of nodes (Before shutdown

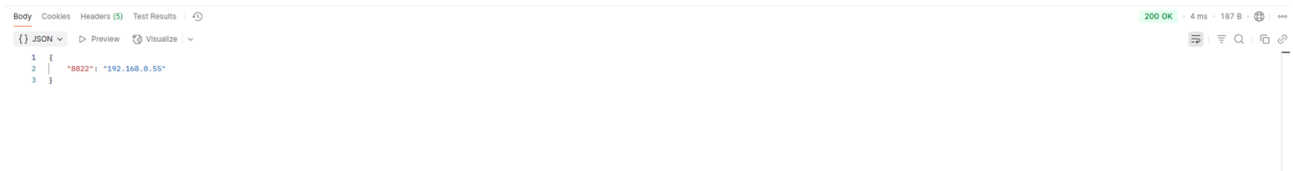
Request Shutdown of a node



The shutting down of a node



List of Nodes after Shutting down of a node

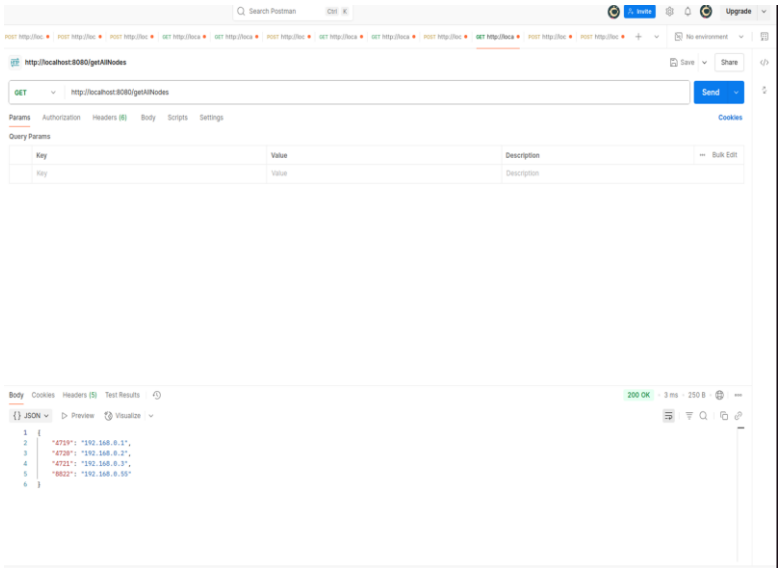


Testing Functionalities

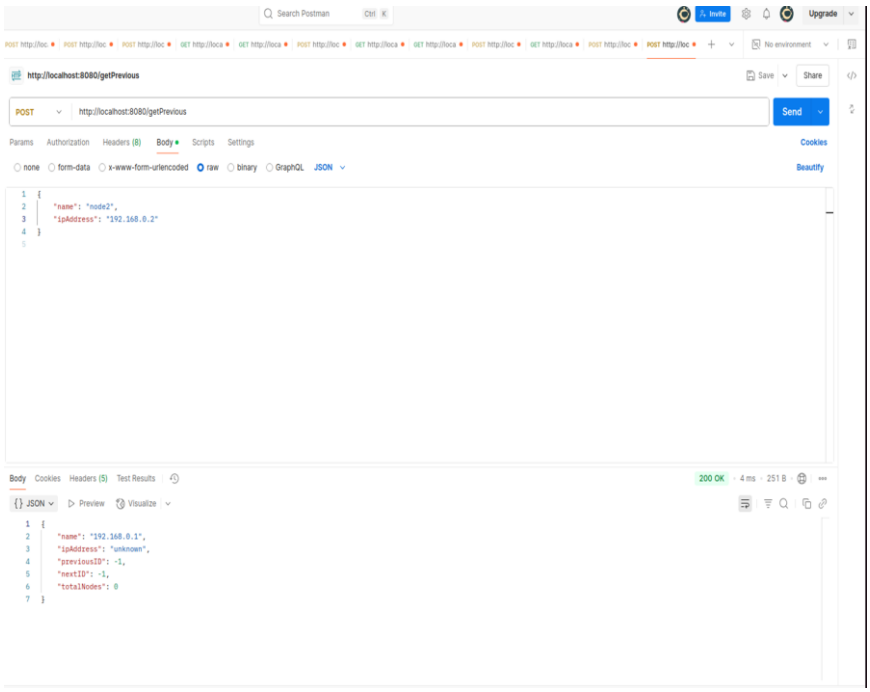
Updating of Next and previous node

All nodes available

Get Previous Node



Get Next Node



5 Challenges and Solutions

Challenge 1: Hash Mismatch

During the development of the **shutdown** functionality, I encountered an issue where the **hash of the node ID** was not consistent between node creation and searching. This was resolved by ensuring that the same **hashing function** was used for both adding and searching nodes in the **nodeMap**.

Challenge 2: Multicast Communication

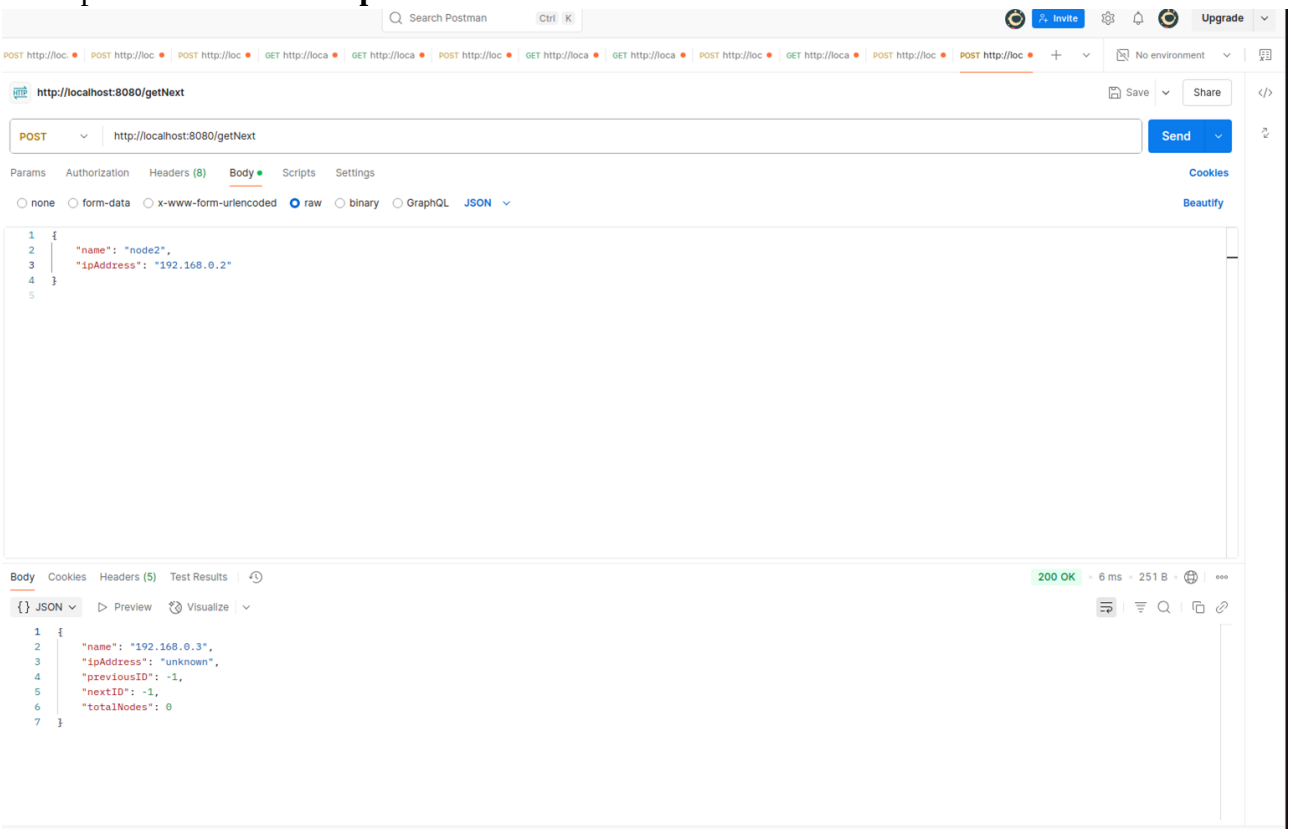
The multicast communication was initially challenging, especially ensuring that **multicast messages** were received correctly by all nodes.

Solution: We addressed this by reviewing the multicast address and port configuration, ensuring all nodes were listening on the same address.

Challenge 3: Synchronizing Neighbor Updates

One of the challenges we faced during the **shutdown** process was ensuring that the **previous** and **next nodes** were properly updated across all nodes. If one node failed to update its neighbor's **ID parameters**, the network would become inconsistent, which could lead to failures in subsequent operations.

Solution: To address this, we implemented a mechanism in the **shutdown** function to ensure that each node communicated with its neighbors to update their **previousID** and **nextID**. This was achieved using **multicast** communication, where the node being shut down sent its **next node's ID** to the previous node and its **previous node's ID** to the next node. This ensured the network remained



consistent even after a node left the system.

Challenge 4: Network Partitioning and Latency

When working with a distributed system, network partitioning and latency can pose significant challenges, especially during the **failure** phase. Nodes may not immediately detect a failure due to network issues or slow communication.

Solution: We incorporated a **retry mechanism** and periodic **ping checks** to ensure that node failures were detected promptly. When a failure was detected, the affected node's neighbors updated their **ID parameters**, and the system was able to recover without any disruption to the other nodes.

6 Conclusion

This lab provided an in-depth understanding of the node lifecycle within a distributed system, specifically focusing on shutdown and failure detection. By implementing multicast communication and bootstrap protocols, we ensured that nodes could dynamically join the network and interact with each other. Additionally, the shutdown process allowed us to gracefully remove nodes from the network, while the failure detection mechanism ensured the system remained operational even in the face of node failures.

The hands-on experience in implementing these features has significantly enhanced my understanding of distributed systems and their resilience. I have learned how critical it is to handle failures, node additions, and removals in a manner that preserves the consistency and availability of the system.

Lab – 5 REPLICATION

1. Introduction

This report documents the design and implementation of a file replication protocol within a distributed system. The primary objective of Lab 5 was to create an architecture that ensures reliable file storage through automatic replication, consistent file tracking, and safe node shutdown. This builds upon the foundation established in Lab 4, which introduced node discovery and faliure.

In this system, when a new file is added by a node—referred to as the *original node* or the *local file creator*—its hash value is computed and compared against those of active nodes arranged in a logical ring topology. The file is assigned to the first node with a hash equal to or greater than the file's hash value. This node is designated as the owner—the node appointed by the naming server to manage and track the file. To improve fault tolerance, the file is then replicated to a predefined number of neighboring nodes in the clockwise direction. These nodes, known as replicas, each maintain a copy of the file and its associated metadata.

Each replica stores log entries that include the file name, the hash of the file owner, and the replication timestamp. This metadata enables coordinated deletion and effective traceability. When a file is deleted, the owner uses its records to identify all replicas and issues deletion commands, ensuring that all redundant copies are consistently removed across the system.

In addition to handling file addition and deletion, the lab addresses the scenario of graceful shutdowns. When a node is removed, it must transfer all its locally owned files and stored replicas to neighboring nodes to prevent data loss. After successful transfer of all data and logs, the node is deregistered from the naming server and exits the system safely.

The naming server, while not central to the system's operation, assists in coordinating shutdown procedures. It updates ownership mappings, ensures that file transfers complete via TCP before node termination, and confirms that the system remains consistent post-exit.

2. Objectives

- Enable automatic file replication using consistent hashing: a node instantly sends out a multicast packet containing its name, the port it listens on, and the list of its local files.
- Track file ownership and replica locations via a Naming Server: Naming Server uses the data from the multicast to register each file and determine its owning node using a consistent file-hashing method.
- Ensure synchronization upon addition of new files: Naming Server maintains a complete overview of file ownership and replica assignments.
- Implement fault tolerance through safe node shutdown protocols.

3. Methodology

3.1 System Architecture

The system consists of distributed nodes and a central Naming Server. Communication takes place over:

- UDP Multicast for node discovery and announcement
- UDP Unicast for replication instructions
- TCP for reliable file transfers

Each node holds:

- Local files (original copies)
- Replicated files (backups from other nodes)

3.2 Startup Phase

1. **Node Startup and Self-Announcement**

When a new node is launched, it initiates its participation in the distributed system by invoking the createAndAnnounceNewNode(...) method from the NodeApp.java class. This method requires four critical inputs:

- A unique node identifier (e.g., "Christian")
- The UDP unicast port the node listens on (e.g., 3030)
- The absolute path to its local file directory
- The path to its replica directory, designated for storing backup copies of files received from other nodes

Upon receiving these parameters, the node creates a Node object, stores the configuration, and then scans its local directory through the loadLocalFilesFromDirectory(...) method. This scan generates two key structures:

- localFileObjects: a list of File objects used for actual data transmission
- localFileNames: a list of filenames used to inform the Naming Server about the node's file inventory

This upfront initialization ensures the node has complete awareness of its local data before broadcasting its existence.

File replication is carried out directly between nodes using TCP connections. Each node runs a persistent FileReceiver TCP server that listens for incoming file transfers. As a result, the system achieves efficient peer-to-peer file replication, while the Naming Server remains focused solely on control-plane duties.

2. **Constructing and Sending the Multicast Message**

Once initialized, the node constructs a multicast announcement using the MulticastSender.java class. This message contains the node’s name, unicast port, and local filenames, all delimited by the pipe (|) symbol to prevent parsing errors. The message is serialized into a byte array and transmitted to

the multicast group 230.0.0.0:4446 via UDP. This address is monitored by all active nodes and the Naming Server, ensuring network-wide delivery of the broadcast.

- Node name
- Unicast port
- Local filenames

3. 3. 3. **Naming Server Registration and Ownership Assignment:**
    - Registers the node
    - Uses consistent hashing to assign ownership
    - Selects replica nodes based on the ring structure
    - Sends replication instructions to original owners

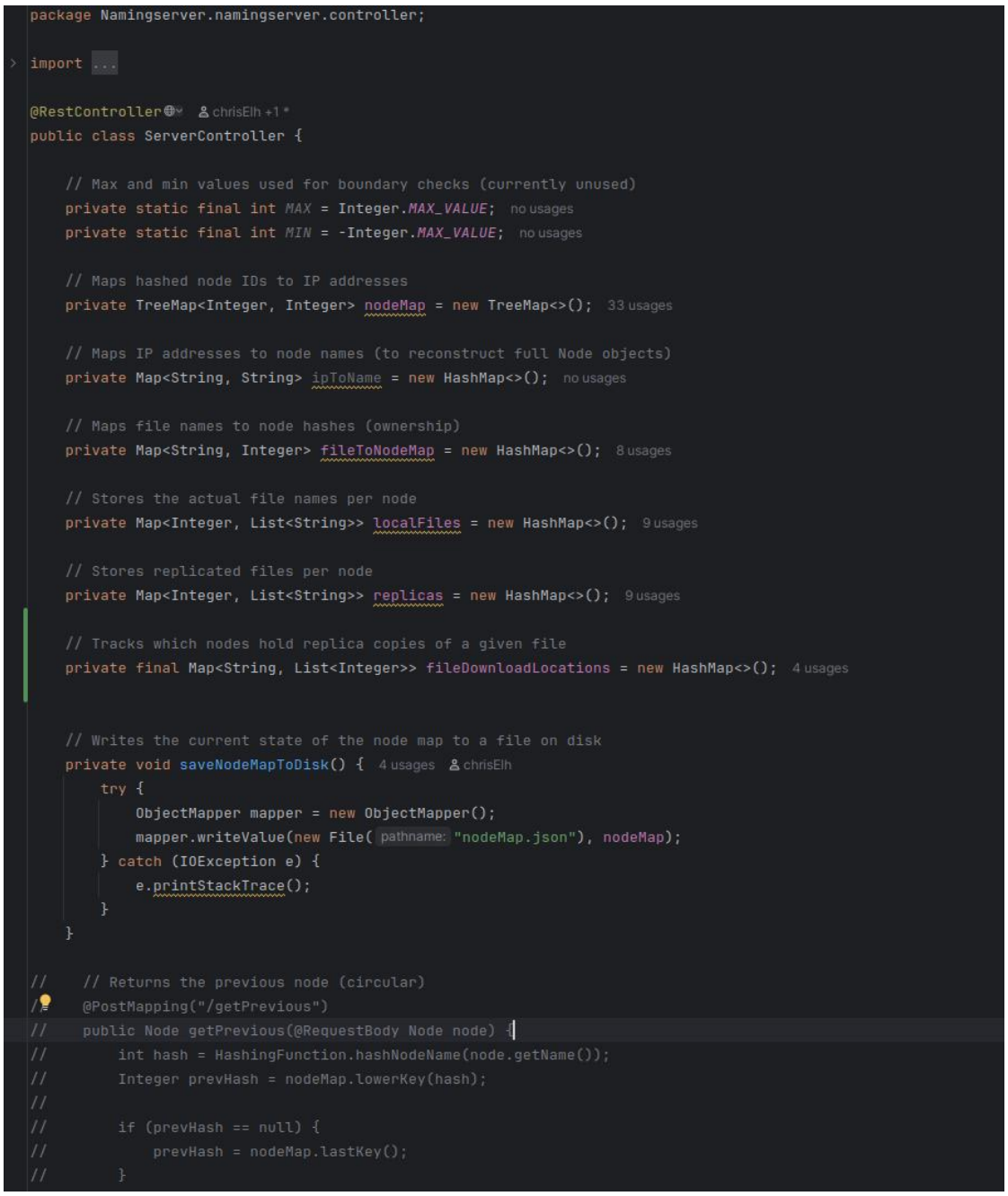
Upon receiving the multicast, the Naming Server processes it using the MulticastReceiver and addNodeFromMulticast(...) methods. The server does not access the contents of the files—only their names. It then computes a hash value for each filename and determines its placement on the logical ring using consistent hashing.

Each file is assigned to the node with the smallest hash value greater than or equal to the file's hash. This node becomes the owner, responsible for managing and serving the file. If no such node exists, ownership wraps around to the first node in the ring.

To increase redundancy and ensure fault tolerance, a replica node is also selected using the same consistent hashing logic. The replica is typically the next node in clockwise order that is not already the owner.

All mappings—file-to-owner and file-to-replica—are stored in the Naming Server’s internal structures (fileToNodeMap, localFiles, and replicas) for consistent coordination.

4.



```java
package Namingserver.namingserver.controller;

import ...

@RestController  chrisElh +1 *
public class ServerController {

    // Max and min values used for boundary checks (currently unused)
    private static final int MAX = Integer.MAX_VALUE;  no usages
    private static final int MIN = -Integer.MAX_VALUE;  no usages

    // Maps hashed node IDs to IP addresses
    private TreeMap<Integer, Integer> nodeMap = new TreeMap<>();  33 usages

    // Maps IP addresses to node names (to reconstruct full Node objects)
    private Map<String, String> ipToName = new HashMap<>();  no usages

    // Maps file names to node hashes (ownership)
    private Map<String, Integer> fileToNodeMap = new HashMap<>();  8 usages

    // Stores the actual file names per node
    private Map<Integer, List<String>> localFiles = new HashMap<>();  9 usages

    // Stores replicated files per node
    private Map<Integer, List<String>> replicas = new HashMap<>();  9 usages

    // Tracks which nodes hold replica copies of a given file
    private final Map<String, List<Integer>> fileDownloadLocations = new HashMap<>();  4 usages


    // Writes the current state of the node map to a file on disk
    private void saveNodeMapToDisk() {  4 usages  chrisElh
        try {
            ObjectMapper mapper = new ObjectMapper();
            mapper.writeValue(new File( pathname: "nodeMap.json"), nodeMap);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

//    // Returns the previous node (circular)
//    @PostMapping("/getPrevious")
//    public Node getPrevious(@RequestBody Node node) {
//        int hash = HashingFunction.hashNodeName(node.getName());
//        Integer prevHash = nodeMap.lowerKey(hash);
//
//        if (prevHash == null) {
//            prevHash = nodeMap.lastKey();
//        }
```

Coordination Without Centralized Transfer

Importantly, the Naming Server does not transfer any files itself. Instead, it sends a unicast UDP instruction to the owner node using the sendReplicaInstruction() method. This instruction includes:

- The filename that must be replicated
- The port of the designated replica node

This decouples the control plane from the data plane, ensuring scalability and reducing server load.

```
        }
    }

    public static void sendReplicaInstruction(String targetPort, String filename, String destinationPort) {  5 usages  Seti_RF
        try {


            String message = "REPLICA:" + filename + ":" + destinationPort;
            InetAddress address = InetAddress.getByName( host: "localhost");
            int port = Integer.parseInt(targetPort);

            byte[] buf = message.getBytes();
            DatagramPacket packet = new DatagramPacket(buf, buf.length, address, port);

            DatagramSocket socket = new DatagramSocket();
            socket.send(packet);
            socket.close();

            System.out.println("Sent replica instruction to port " + port + " → " + message);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }


}
```

4. Owner-Driven File Replication via TCP

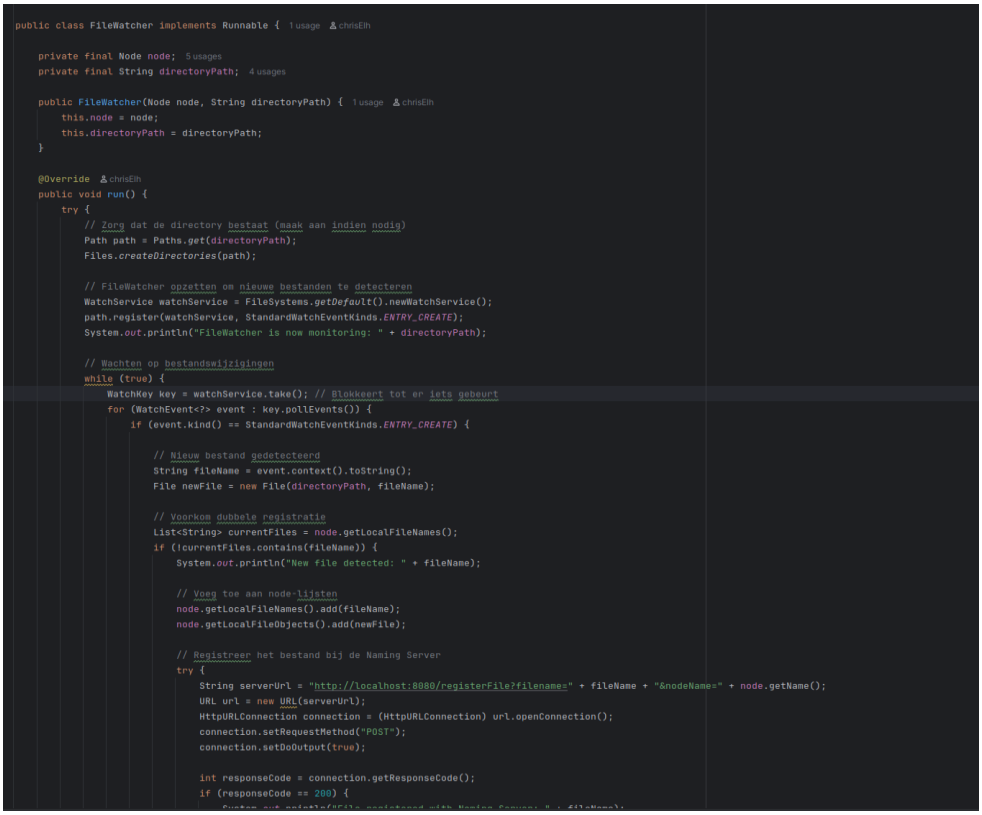
Once the owner node receives a "REPLICA:" instruction via its UnicastReceiver, the message is parsed in the handler logic within NodeApp.java. The handler extracts the filename and the replica port, confirms the file's existence in localFileObjects, and then uses FileSender.sendFile() to initiate a TCP connection with the replica node.

FileSender streams the file in chunks, ensuring full and reliable delivery regardless of size. Meanwhile, the receiving node's FileReceiver thread, which starts automatically at node initialization, listens for incoming TCP connections.

When the file transfer begins, the handleIncomingFile() method is invoked. It identifies the file using a header such as STORE_FILE:<filename>, writes the data to the replica directory, and finalizes the process by calling addReplicatedFile(). This function registers the replicated file in the node's internal structures, maintaining consistency.

3.3 Update Phase

- A FileWatcher monitors the local folder for new files.



```java
public class FileWatcher implements Runnable {  1 usage  chrisElh

    private final Node node;  5 usages
    private final String directoryPath;  4 usages

    public FileWatcher(Node node, String directoryPath) {  1 usage  chrisElh
        this.node = node;
        this.directoryPath = directoryPath;
    }

    @Override  chrisElh
    public void run() {
        try {
            // Zorg dat de directory bestaat (maak aan indien nodig)
            Path path = Paths.get(directoryPath);
            Files.createDirectories(path);

            // FileWatcher opzetten om nieuwe bestanden te detecteren
            WatchService watchService = FileSystems.getDefault().newWatchService();
            path.register(watchService, StandardWatchEventKinds.ENTRY_CREATE);
            System.out.println("FileWatcher is now monitoring: " + directoryPath);

            // Wachten op bestandswijzigingen
            while (true) {
                WatchKey key = watchService.take(); // Blokkeert tot er iets gebeurt
                for (WatchEvent<?> event : key.pollEvents()) {
                    if (event.kind() == StandardWatchEventKinds.ENTRY_CREATE) {

                        // Nieuw bestand gedetecteerd
                        String fileName = event.context().toString();
                        File newFile = new File(directoryPath, fileName);

                        // Voorkom dubbele registratie
                        List<String> currentFiles = node.getLocalFileNames();
                        if (!currentFiles.contains(fileName)) {
                            System.out.println("New file detected: " + fileName);

                            // Voeg toe aan node-lijsten
                            node.getLocalFileNames().add(fileName);
                            node.getLocalFileObjects().add(newFile);

                            // Registreer het bestand bij de Naming Server
                            try {
                                String serverUrl = "http://localhost:8080/registerFile?filename=" + fileName + "&nodeName=" + node.getName();
                                URL url = new URL(serverUrl);
                                HttpURLConnection connection = (HttpURLConnection) url.openConnection();
                                connection.setRequestMethod("POST");
                                connection.setDoOutput(true);

                                int responseCode = connection.getResponseCode();
                                if (responseCode == 200) {
```

The FileWatcher class is implemented as a Runnable and launched in its own thread immediately after a node starts. It receives two parameters:

- The active Node instance
- The full path to the node's local directory (e.g., "./data/Christian")

Using Java's WatchService, FileWatcher listens specifically for **ENTRY_CREATE** events, allowing it to detect the moment a new file appears in the directory.

- Upon file creation:
    1. File is added to internal lists
    2. Naming Server receives registration via HTTP POST
    3. Hashing determines new owner and replica
    4. If node is owner, it receives replication instruction via UDP
    5. File is transferred over TCP to replica node

What Happens When a New File is Detected?
Once a file creation event is captured, FileWatcher follows a structured procedure:

1. Duplicate Check: The filename is compared with the node's existing localFileNames to avoid redundant entries.
2. Metadata Update: If the file is new, it is added to both localFileObjects and localFileNames.

3. Registration: A POST request is sent to the Naming Server with the filename and node identity:

```
            System.err.println("Invalid file, cannot add to replicated list: " + file);
        }
    }


    // Return only the names of the local files (for communication with server)
    public List<String> getLocalFileNames() {  9 usages   chrisElh
        return localFileNames;
    }

    // Return actual File objects
    public List<File> getLocalFileObjects() {  3 usages   chrisElh
        return localFileObjects;
    }

    // Optional setter if needed manually
    public void setLocalFileNames(List<String> names) {  no usages   chrisElh
        this.localFileNames = names;
    }

    // Other existing methods (name, port, etc.) remain unchanged



}

```

Naming Server's Role: Assigning Ownership and Triggering Replication

Upon receiving the registration request, the Naming Server performs the following within registerFile(...):

- Computes the hash of the filename
- Determines the owner node by finding the first node whose hash is greater than or equal to the file's hash
- Updates fileToNodeMap and the owner's entry in localFiles
- Selects a replica node using consistent hashing
- If the replica differs from the owner, it adds the file to the replicas map and sends a UDP unicast instruction to the owner node, directing it to replicate the file to the chosen replica

```
// Maps file names to node hashes (ownership)
private Map<String, Integer> fileToNodeMap = new HashMap<>();  8 usages

// Stores the actual file names per node
private Map<Integer, List<String>> localFiles = new HashMap<>();  9 usages

// Stores replicated files per node
private Map<Integer, List<String>> replicas = new HashMap<>();  9 usages

// Tracks which nodes hold replica copies of a given file
private final Map<String, List<Integer>> fileDownloadLocations = new HashMap<>();  4 usages
```

Note: This exact replication instruction logic is also used in addNodeFromMulticast(...) during startup, ensuring behavioral consistency between startup and runtime file events.

The owner node receives the "REPLICA:" instruction via its UnicastReceiver. This is parsed in the handler within NodeApp.java, where the filename and replica port are extracted. If the file is found in the node's local directory, the file is sent to the replica node using:

FileSender.sendFile(filename, replicaPort);

The file is transmitted via a **TCP connection**, byte-by-byte, ensuring complete and reliable delivery.

Replica Node: Receiving and Registering the File
The receiving replica node runs a persistent FileReceiver thread, always listening for incoming TCP connections. Upon receiving a file:

- It reads the header (e.g., STORE_FILE:<filename>)
- Writes the file to the **replica directory**
- Calls addReplicatedFile(...) to update replicatedFileObjects and replicatedFileNames, finalizing the replication process

This ensures that the newly added file is successfully mirrored and recorded within the replica node's internal state.

In parallel with additions, deletions are also tracked. When a file is removed locally from an owner node, the node checks its internal metadata to identify all replica holders. It then sends deletion instructions to those replicas, prompting them to erase the file and update their replica logs. This ensures clean and coordinated removal of obsolete files across the system.

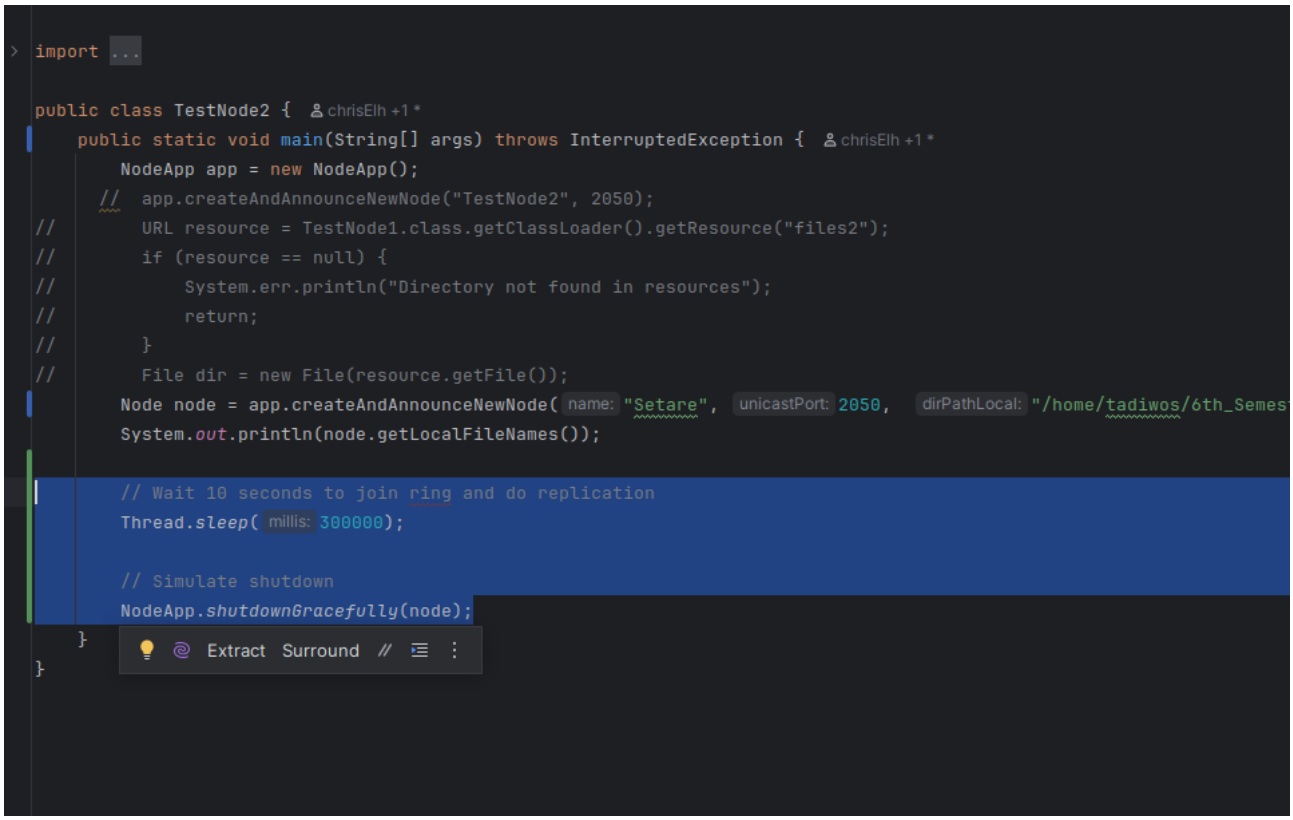
3.4 Shutdown Phase (Partially Implemented) | Graceful Node Removal and Replica Reallocation

1. Node sends shutdown request to Naming Server
2. Server reassigns replicated files to suitable predecessors
3. Ensures no duplication: skips nodes already owning the file
4. Sends unicast to shutdown node to perform TCP transfer
5. File log and mapping updates are pending final implementation

Shutdown Trigger and Initial Actions

When a node (e.g., **Node A**) intends to leave the system voluntarily, it initiates the shutdown process by sending a **GET request** to the Naming Server:

/shutdown?port=<nodePort>



Transferring Replicated Files to Predecessors

The first and most crucial step in the shutdown flow is ensuring that all replicated files stored on the departing node are safely transferred to other nodes. This is handled as follows:

1. The server queries the replicas map for all files stored on the departing node.
2. For each file:
    - It locates the previous node in the ring (using TreeMap.lowerKey(...)).
    - If this previous node already owns the file locally, the algorithm steps back one more node to avoid duplicate storage.
3. The file is:
    - Reassigned in fileToNodeMap as a local file of the new node
    - Added to the node's local file list in localFiles
4. A UDP unicast REPLICA instruction is sent to the shutting-down node, directing it to send the file over TCP to the newly selected node using FileSender.sendFile(...)

```
        fileToNodeMap.put(filename, prevHash);
        localFiles.computeIfAbsent(prevHash, Integer k -> new ArrayList<>()).add(filename);

        // 4. instruct the shutting-down node to send the file via unicast
        int sourcePort = node.getPort();           // the port of the node being removed
        int targetPort = nodeMap.get(prevHash);     // the port of the new owner
        System.out.println("Shutdown: moving '" + filename +
                "' → new owner hash=" + prevHash +
                " (port=" + targetPort + ")");
        ServerUnicastSender.sendReplicaInstruction(
                String.valueOf(sourcePort),
                filename,
                String.valueOf(targetPort)
        );
    }
```

File Transmission and Confirmation

The shutting-down node iterates through its replica files and, for each one, sends a **REPLICA instruction** containing the filename and the target port. Each instruction triggers a TCP connection, where:

- The **receiving node** (e.g., Node B) listens on its unicast port using FileReceiver.
- Once the file is received, it is stored in the replica directory and registered internally using addReplicatedFile().

```
    }

    // Adds a replicated file to the node's list (both File object and name)
    public void addReplicatedFile(File file) {  1 usage   chrisElh
        if (file != null && file.exists() && file.isFile()) {
            replicatedFileObjects.add(file);
            replicatedFileNames.add(file.getName());
            System.out.println("Replicated file added: " + file.getName());
        } else {
            System.err.println("Invalid file, cannot add to replicated list: " + file);
        }
    }
```

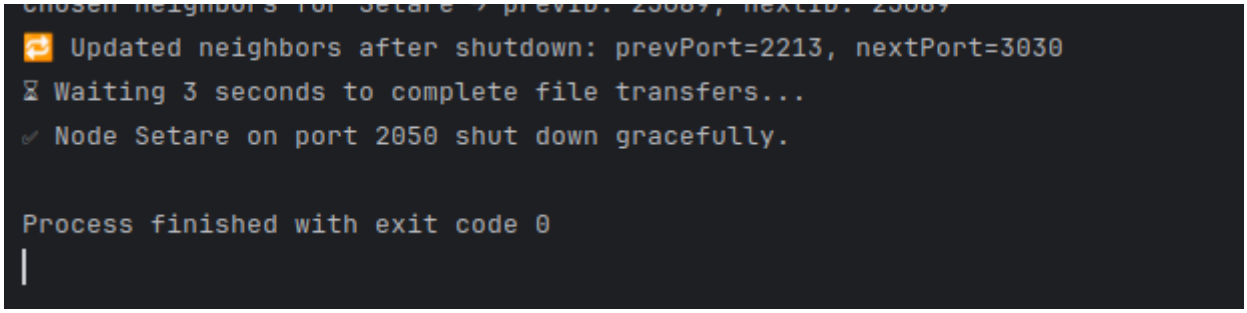
The system ensures that all transfers are complete before the node proceeds to deregister.

Post-Transfer Node Deregistration

Once all replica files have been successfully reassigned and transferred, the shutting-down node is:

- **Removed from** nodeMap in the Naming Server
- Deleted from all other data structures, including fileToNodeMap, localFiles, and replicas

This completes the **Phase 1** of the shutdown flow: the safe reallocation of replicated data.



Planned Features (Not Yet Implemented)

Although the file transfer logic is functional, the shutdown protocol is not fully complete. The following features are part of the intended design but have yet to be realized:

Replica Log Transfer

- Each file should carry metadata that includes:
    - Original owner
    - Download locations
- When replicas are transferred, this log data should also move to the new holder to maintain traceability.

Owner Notification

- The original file owner should be notified when one of its replica nodes is being shut down.
- This allows the owner to:
    - Update its internal log of active replica locations
    - Remove the file completely if no other downloads have occurred

Design Considerations and Edge Case Handling

- The shutdown process is built with redundancy avoidance in mind: No node should end up storing both a local and replica version of the same file.
- The use of consistent hashing ensures that reassignment follows the deterministic placement rules of the system.
- By preserving ring topology and only shifting ownership clockwise or counter-clockwise, the balance of the system is maintained.

```java
@PostMapping("/removeNode")
public String removeNode(@RequestBody Node node) {
    int hash = HashingFunction.hashNodeName(node.getName());

    if (!nodeMap.containsKey(hash)) {
        return "Node not found for removal: " + node.getName();
    }

    // ── Phase 1: transfer replicated files ──────────────────────
    // Get list of files that were replicas on this node
    List<String> replicatedFiles = replicas.getOrDefault(hash, Collections.emptyList());

    for (String filename : replicatedFiles) {
        // 1. find previous node in the ring
        Integer prevHash = nodeMap.lowerKey(hash);
        if (prevHash == null) prevHash = nodeMap.lastKey(); // wrap around

        // 2. edge case: if prev node already has this file locally, skip back one more
        List<String> prevLocal = localFiles.getOrDefault(prevHash, Collections.emptyList());
        if (prevLocal.contains(filename)) {
            Integer prevPrev = nodeMap.lowerKey(prevHash);
            if (prevPrev == null) prevPrev = nodeMap.lastKey();
            prevHash = prevPrev;
        }

        // 3. update maps: new owner is prevHash
        fileToNodeMap.put(filename, prevHash);
```

```java
        fileToNodeMap.put(filename, prevHash);
        localFiles.computeIfAbsent(prevHash, Integer k -> new ArrayList<>()).add(filename);

        // 4. instruct the shutting-down node to send the file via unicast
        int sourcePort = node.getPort();           // the port of the node being removed
        int targetPort = nodeMap.get(prevHash);     // the port of the new owner
        System.out.println("Shutdown: moving '" + filename +
                "' → new owner hash=" + prevHash +
                " (port=" + targetPort + ")");
        ServerUnicastSender.sendReplicaInstruction(
                String.valueOf(sourcePort),
                filename,
                String.valueOf(targetPort)
        );
    }
    // ────────────────────────────────────────────────────────────


    nodeMap.remove(hash);
    localFiles.remove(hash);
    replicas.remove(hash);
    fileToNodeMap.values().removeIf(Integer value -> value == hash);
    saveNodeMapToDisk();
```

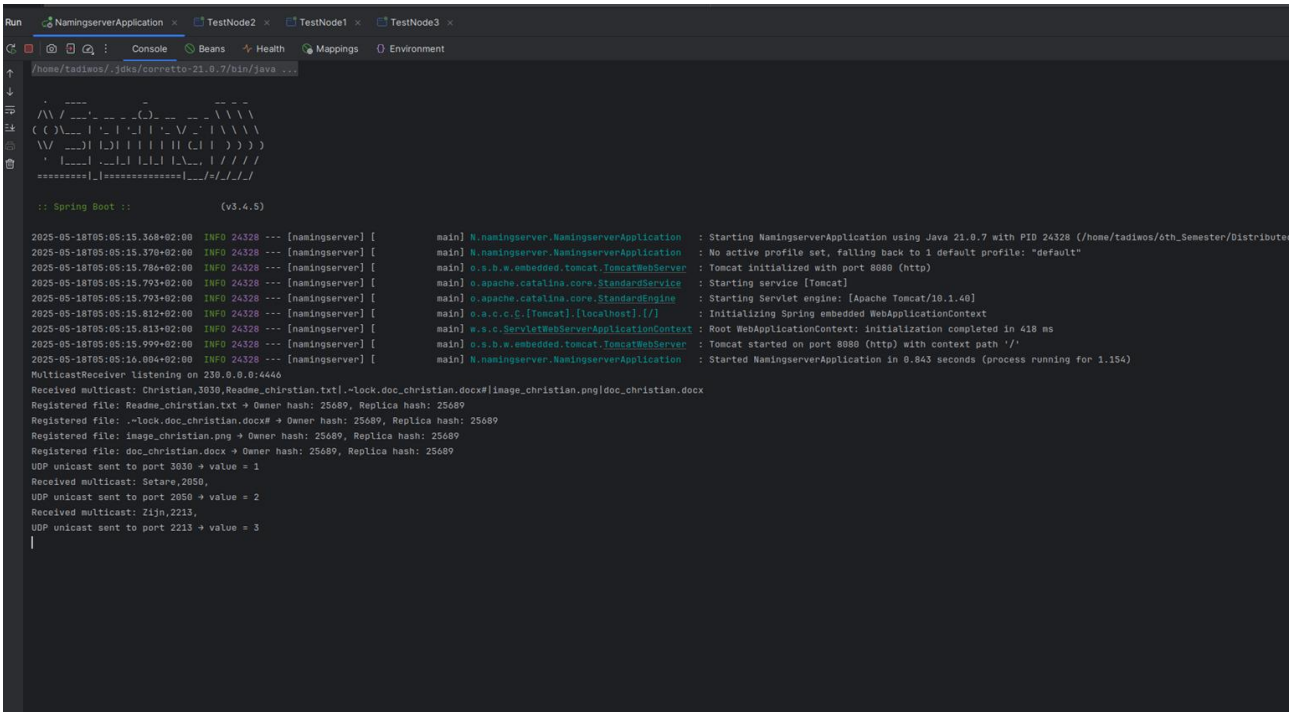
4. Implementation

- NodeApp.java: Node lifecycle logic, directory scanning, and UDP handlers
- MulticastSender.java: Assembles and sends multicast packets
- MulticastReceiver.java: Server-side receiver for multicast packets
- FileSender.java: Sends files using TCP
- FileReceiver.java: Constantly listens for incoming files
- FileWatcher.java: Runnable class using Java WatchService for live monitoring

5. Results and Tests

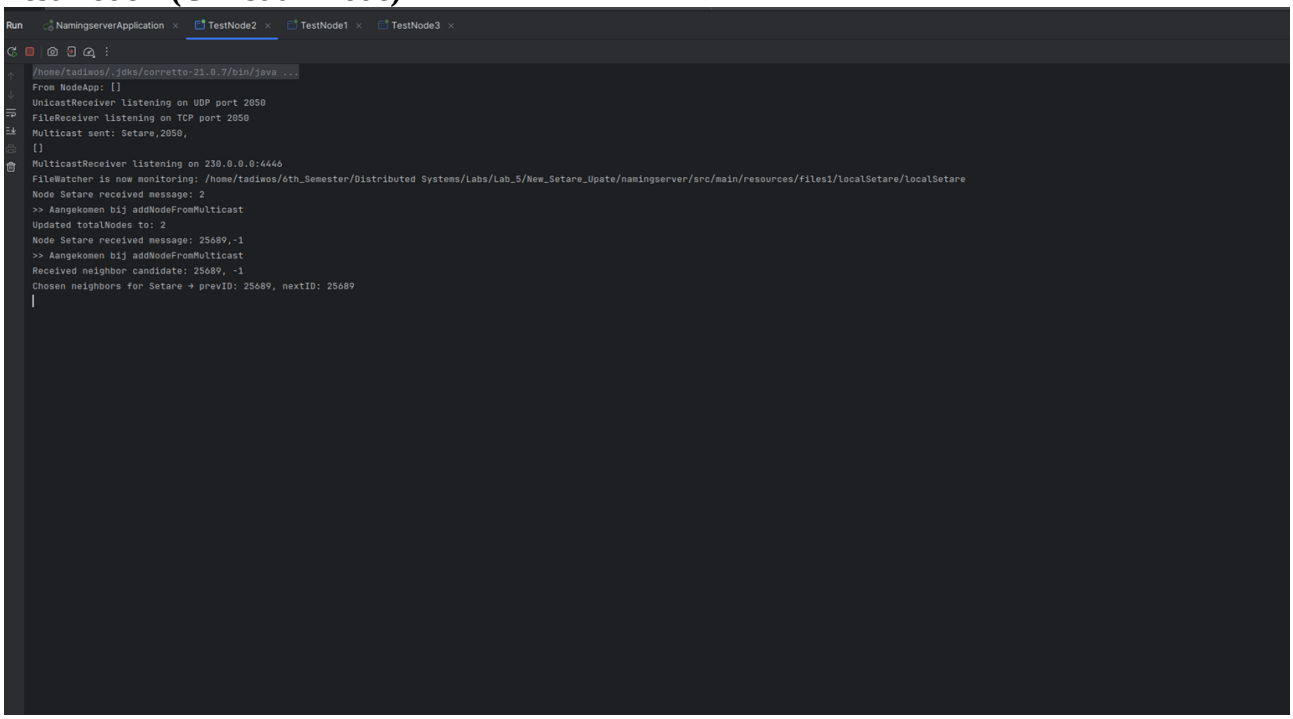
Test Case	Expected Result	Status
Node joins with files	Files registered and replicated	✅
File added during runtime	Watcher triggers registration and replication	✅
File hash == node hash	File is not replicated; node is its own owner	✅
Node shutdown	Replicated files pushed to predecessor	✅ (partial)
Owner notification & download log	Metadata update	❌ Pending

Server Running

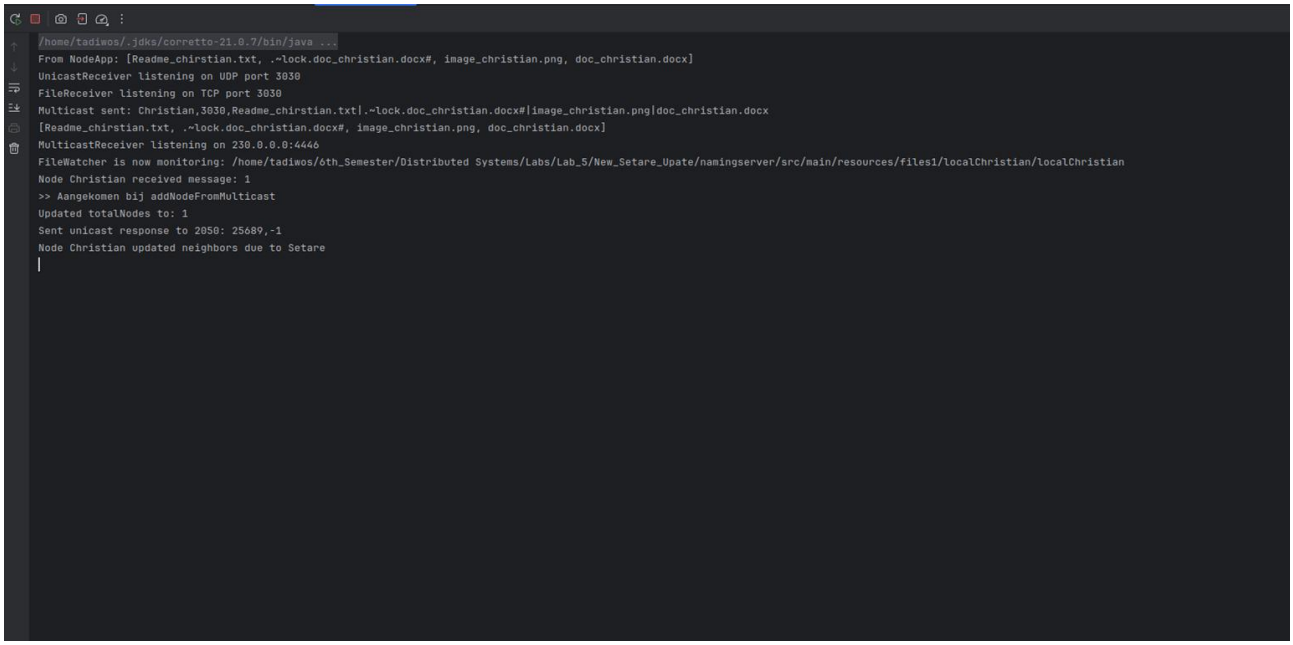


Node Tests for Starting, Update and Shutdown

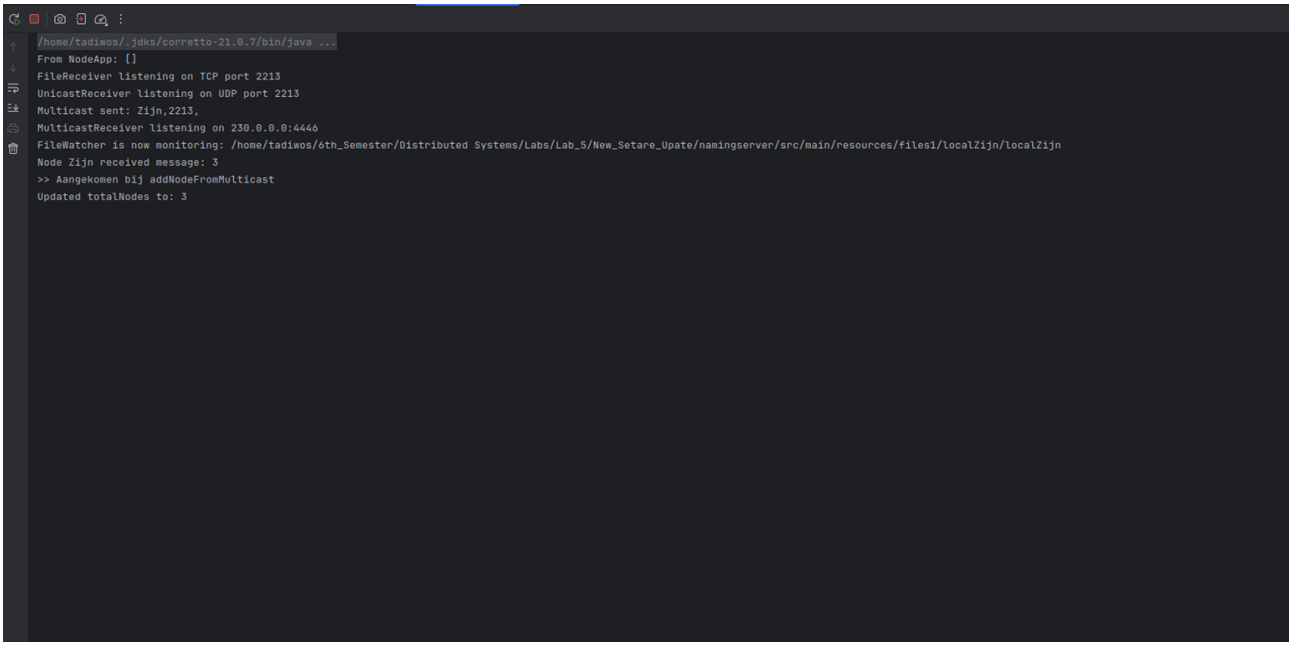
Test Node 1 (Christian Node)



Test Node 2 (Setare Node)



Test Node 3 (Zijn Node)



6. Challenges and Solutions

- Replica collision: Avoiding replication to the same node that owns the file.
    - *Solution*: Lookup adjusted to skip nodes with existing local copies.
- Multicast formatting: Handling filenames with special characters.
    - *Solution*: Use | as a safe delimiter.
- Incomplete shutdown protocol:
    - *Pending*: Download log update and acknowledgment mechanism.

7. Conclusion

This lab successfully demonstrated key distributed systems concepts such as consistent hashing, fault-tolerant replication, and real-time update propagation. Although parts of the shutdown protocol are still under development, the system's core functionality is reliable and scalable. The architecture mirrors modern decentralized storage systems by decoupling coordination from data transfer.

LAB – 6 AGENT

1 Introduction

The purpose of Lab 6 was to implement an autonomous Agent-based failure detection and recovery mechanism within a distributed ring-based file replication system. Each node in the system is designed to monitor its neighbors through periodic pinging. If a node failure is detected, an internal FailureAgent is initiated to travel around the ring and trigger proper file re-replication and neighbor reconfiguration. This mechanism ensures that system reliability and data consistency are maintained even in the presence of node failures.
This lab extends the functionalities developed in previous labs (such as node registration, file replication, and message propagation) by focusing on resilience and robustness in dynamic distributed environments.

2. Objective

- To design and integrate a fault-tolerant agent-based failure handling system.
- To enable nodes to detect failures of their neighbors via consistent heartbeat (ping) checks.
- To propagate failure information around the network through an autonomous software agent (FailureAgent).
- To trigger re-replication of files and reconfiguration of ring connections upon confirmed failure.
- To log and verify failure handling through terminal outputs and internal data structures.

3. System Architecture and Components

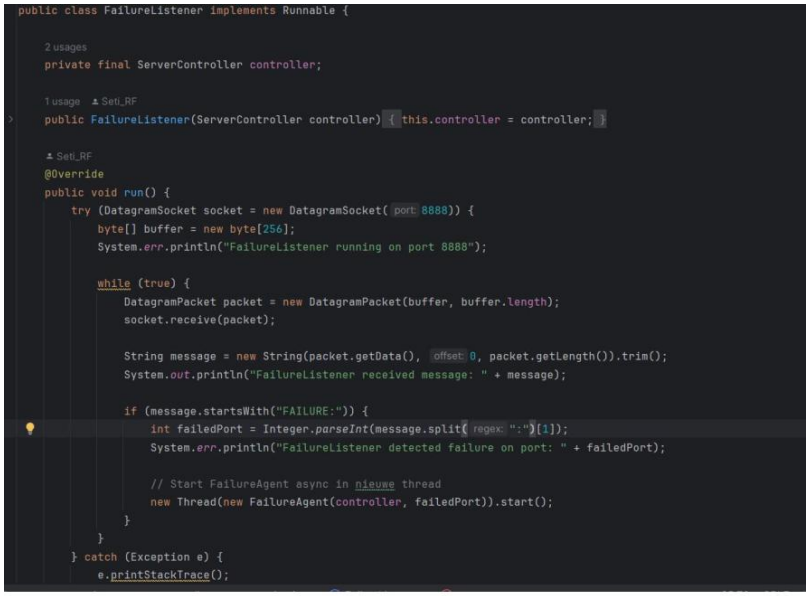
The lab implementation introduces the following core components:

Starting Failure Monitoring

```
public void startFailureMonitor(Node node) {
    scheduler.scheduleAtFixedRate(() -> {
        int failedPort = node.getNextPort();
        System.out.printf("…pinging neighbor at port %d…%n", failedPort);
        try (Socket s = new Socket( host: "localhost", failedPort)) {
            s.setSoTimeout(2000);
            s.getOutputStream().write("PING".getBytes());
        } catch (IOException ioe) {
            System.err.println("Failure detected on port " + failedPort);
            FailureReporter.reportFailure(failedPort);
            int[] nb = getUpdatedNeighborsFromNamingServer(failedPort);
            node.setPreviousPort(nb[0]);
            node.setNextPort(nb[1]);
            System.out.printf("   → new prev=%d, new next=%d%n", nb[0], nb[1]);
            notifyNamingServerToRemoveNode(failedPort);
        }
    }, initialDelay: 5, period: 5, TimeUnit.SECONDS);
    node.markFailureMonitorStarted();
}
```

3.1 FailureListener.java

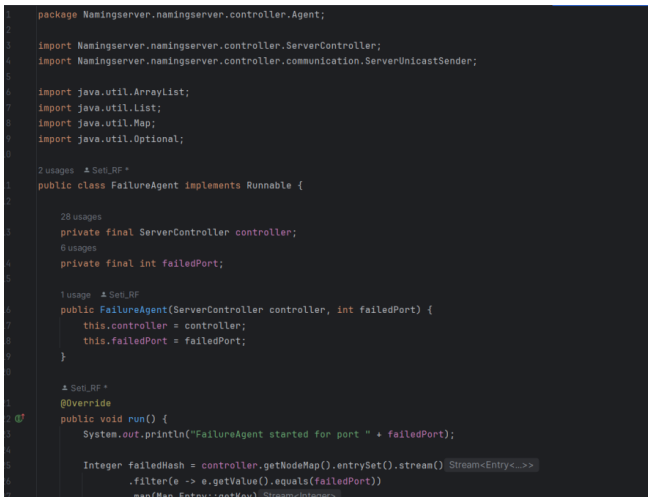
- A continuously running thread on each node.
- Periodically pings the next and previous neighbors in the ring using TCP sockets.
- If a node fails to respond after a fixed number of retries, it is flagged as unresponsive.
- Triggers the launch of the FailureAgent with relevant metadata about the suspected failure.'



```
public class FailureListener implements Runnable {

    2 usages
    private final ServerController controller;

    1 usage   Seti_RF
    public FailureListener(ServerController controller) { this.controller = controller; }

    Seti_RF
    @Override
    public void run() {
        try (DatagramSocket socket = new DatagramSocket( port: 8888)) {
            byte[] buffer = new byte[256];
            System.err.println("FailureListener running on port 8888");

            while (true) {
                DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
                socket.receive(packet);

                String message = new String(packet.getData(),  offset: 0, packet.getLength()).trim();
                System.out.println("FailureListener received message: " + message);

                if (message.startsWith("FAILURE:")) {
                    int failedPort = Integer.parseInt(message.split( regex: ":")[1]);
                    System.err.println("FailureListener detected failure on port: " + failedPort);

                    // Start FailureAgent async in nieuwe thread
                    new Thread(new FailureAgent(controller, failedPort)).start();
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
```

3.2 FailureAgent.java

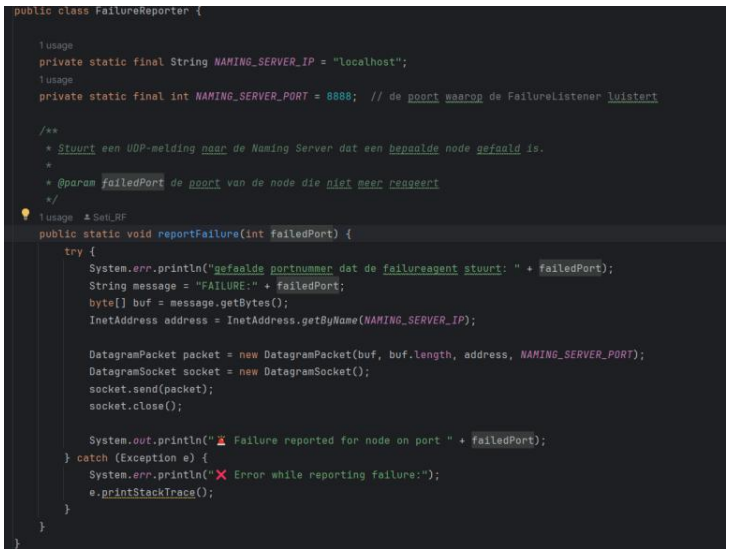
- A mobile software agent class that travels across the ring.
- Carries information about the failed node and its files.
- Visits all alive nodes in a circular manner until it returns to the origin.
- At each hop, it updates the replica or owner assignments for the files held by the failed node.
- Concludes its operation once all necessary re-replication decisions are made.



```
package Namingserver.namingserver.controller.Agent;

import Namingserver.namingserver.controller.ServerController;
import Namingserver.namingserver.controller.communication.ServerUnicastSender;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.Optional;

2 usages   Seti_RF *
public class FailureAgent implements Runnable {

    28 usages
    private final ServerController controller;
    6 usages
    private final int failedPort;

    1 usage   Seti_RF
    public FailureAgent(ServerController controller, int failedPort) {
        this.controller = controller;
        this.failedPort = failedPort;
    }

    Seti_RF *
    @Override
    public void run() {
        System.out.println("FailureAgent started for port " + failedPort);

        Integer failedHash = controller.getNodeMap().entrySet().stream() Stream<Entry<...>>
                .filter(e -> e.getValue().equals(failedPort))
```

3.3 FailureReporter.java

- Acts as the sender module to broadcast or forward the FailureAgent to the next node.
- Uses unicast communication with serialized data transfer for transporting the agent.
- Ensures consistency by retrying failed attempts and logs successful receptions.



```java
public class FailureReporter {

    1 usage
    private static final String NAMING_SERVER_IP = "localhost";
    1 usage
    private static final int NAMING_SERVER_PORT = 8888;  // de poort waarop de FailureListener luistert

    /**
     * Stuurt een UDP-melding naar de Naming Server dat een bepaalde node gefaald is.
     *
     * @param failedPort de poort van de node die niet meer reageert
     */
    1 usage   ♟ Seti_RF
    public static void reportFailure(int failedPort) {
        try {
            System.err.println("gefaalde portnummer dat de failureagent stuurt: " + failedPort);
            String message = "FAILURE:" + failedPort;
            byte[] buf = message.getBytes();
            InetAddress address = InetAddress.getByName(NAMING_SERVER_IP);

            DatagramPacket packet = new DatagramPacket(buf, buf.length, address, NAMING_SERVER_PORT);
            DatagramSocket socket = new DatagramSocket();
            socket.send(packet);
            socket.close();

            System.out.println("⚠️ Failure reported for node on port " + failedPort);
        } catch (Exception e) {
            System.err.println("❌ Error while reporting failure:");
            e.printStackTrace();
        }
    }
}
```

For each file:

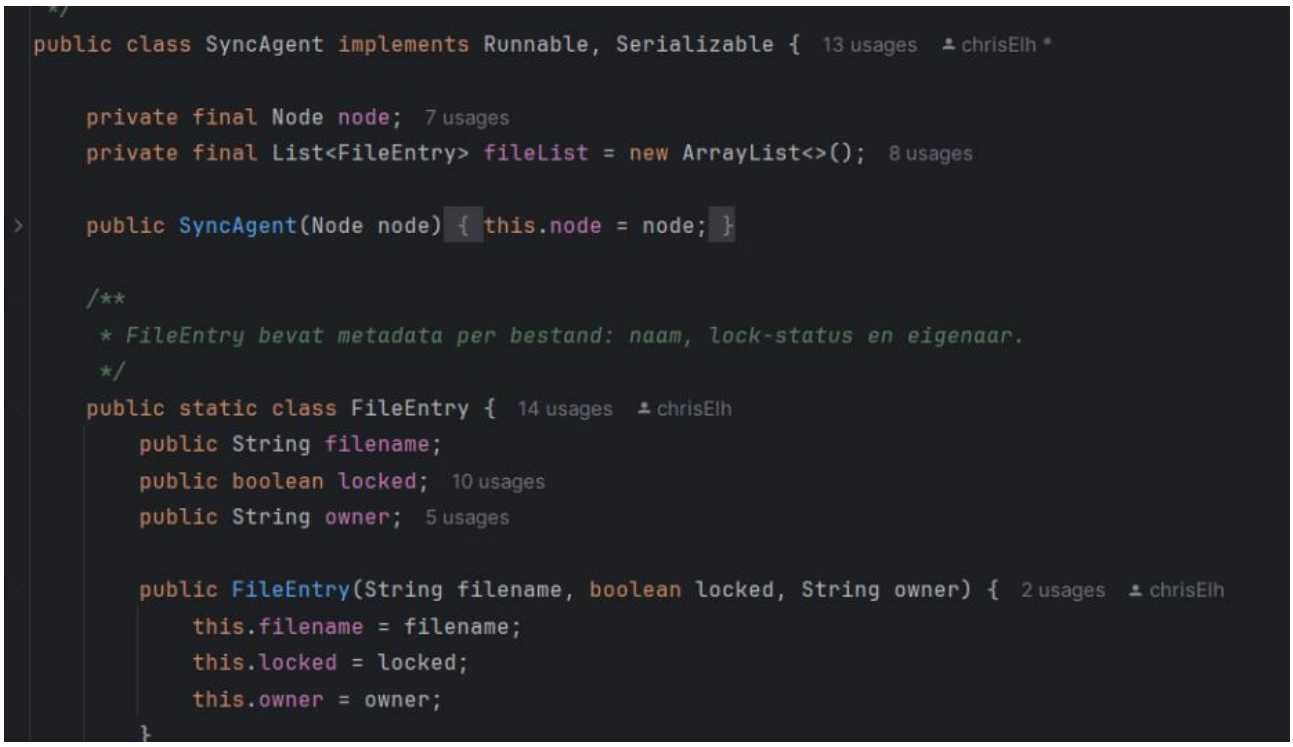
- If the failed node was its registered owner (via fileToNodeMap), the system looks for a replica holder from replicas.
- If a replica exists:
    - ▪ That replica node is promoted to become the new owner (fileToNodeMap.put(...)).
    - ▪ The file is moved from the replica list to the local files of the new owner.
    - ▪ A new replica is assigned elsewhere (not the owner) by hashing the filename and choosing the nearest available node using consistent hashing logic (floorKey, fallback to
      lastKey).
    - ▪ A unicast message (sendReplicaInstruction) is sent to transfer the file from the new owner to the new replica node.
    - ▪ Additionally, a second-level replica is optionally created for redundancy, skipping any node already involved.
- If no replica exists for the file, it is marked as lost and removed from the ownership map.

3.4 NodeApp.java Integration

- Handles invocation of failure response logic on agent reception.
- Coordinates the reallocation of files, updating the file-to-node maps maintained by the Naming Server.
- Deletes replicas associated with failed nodes and initiates regeneration of those files on alternative nodes.

3.5 **SyncAgent.java**

This class implements Runnable and contains the synchronization loop. It maintains an internal list of all known files (FileEntry) and communicates with the neighbor to update its metadata. It also enforces locking rules based on file ownership



• The class implements Runnable so it can be executed in a thread.
• Serializable is used to support agent transfer, e.g., in the future for mobile agents.
• The node reference is passed via constructor and gives access to the node's name, port, and local file list.
• fileList is a list that contains all known file metadata (FileEntry objects) for this node

4. Workflow

1. Ping Monitoring: Each node monitors its prev and next via periodic TCP pings.
2. Failure Detection: If 3 consecutive pings fail (customizable threshold), the node flags a failure.
3. Agent Dispatch: The FailureAgent is created with:
    - ID of failed node.
    - List of files owned or replicated by the failed node.
4. Agent Propagation: The agent is sent clockwise around the ring:
    - Each alive node receives it via TCP socket.
    - It logs file updates or reassignments.
5. Ring Reconfiguration:
    - prev and next values of affected neighbors are updated.
    - New replicas are selected for orphaned files using consistent hashing.
6. Agent Termination: When it completes a full ring traversal, it terminates.

Local File Discovery

• Retrieves the list of files currently in the node's local storage.
• For each file, checks if it is already in the fileList.

• If not, creates a new FileEntry with the current node as owner and locked=false.
• Adds it to fileList

Lock and Unlocking Files
Step-by-step breakdown:

1. Search the file in the internal list: Optional<FileEntry> entryOpt = fileList.stream().filter(e -> e.filename.equals(filename)).findFirst();
This uses Java streams to find the first entry in fileList that matches the requested filename.

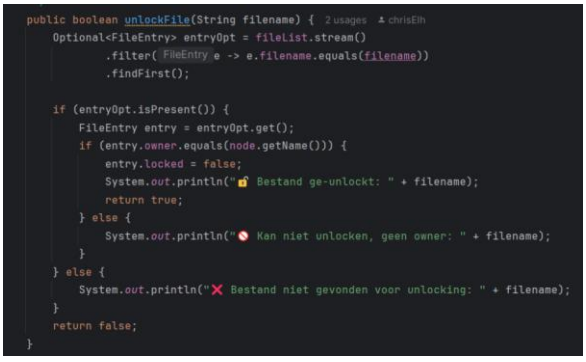
2. Check if the file exists: if (entryOpt.isPresent())
If no matching file is found, the method logs:
" Bestand niet gevonden voor locking: " + filename, and returns false.

3. Verify ownership:
if (entry.owner.equals(node.getName()))
The file can only be locked if the current node is the designated owner. This enforces access control.

4. Apply the lock:

entry.locked = true;
The file is marked as locked. The operation is logged:
" Bestand gelockt: " + filename, and the method returns true.

5. If not the owner:
System.out.println(" Kan niet locken, geen owner: " + filename);
An error is logged and false is returned.



Merge Neighbor Files into Local List:

Step-by-step breakdown:

1. Iterate over all neighbor files: for (FileEntry nf : neighborFiles)
Each received FileEntry from the neighbor is processed individually.

2. Check if the file already exists locally:
Uses Java streams to check whether this file already exists in the local metadata list.

3. If the file is new: fileList.add(nf);
The file is added to the local fileList, including its locked status and owner.
A log message confirms this:
" Nieuw bestand van neighbor: ...".

4. If the file already exists: FileEntry current = existing.get();
Retrieves the existing file entry.

5. Update the lock status if needed: if (current.locked != nf.locked)
If the neighbor has a different lock state, the local entry is updated:
current.locked = nf.locked;
and a message is printed:
" Lock-status bijgewerkt: ..."

File watcher:

is responsible for monitoring the file system in real time. It supports event-based updates and parallel polling to detect file usage and changes. The following parts cover the core logic of lock detection, event handling for modification and deletion, and cooldown management to avoid redundant actions.

```
    // Polling-thread voor file locks
    new Thread(() -> {
        while (true) {
            try {
                for (String fileName : node.getLocalFileNames()) {
                    File file = new File(directoryPath, fileName);
                    boolean isLocked = isFileLocked(file);

                    if (isLocked && !lockState.getOrDefault(fileName, defaultValue: false)) {
                        System.out.println("🔒 File locked (extern geopend): " + fileName);
                        syncAgent.lockFile(fileName);
                        lockState.put(fileName, true);
                    } else if (!isLocked && lockState.getOrDefault(fileName, defaultValue: false)) {
                        System.out.println("🔓 File unlocked (extern gesloten): " + fileName);
                        syncAgent.unlockFile(fileName);
                        lockState.put(fileName, false);
                    }
                }
                Thread.sleep( millis: 1000); // 1 seconde pauze
            } catch (Exception e) {
                System.err.println("❌ Lock polling error:");
                e.printStackTrace();
            }
        }
    }).start();
    private boolean isFileLocked(File file) { 1 usage  chrisElh
        try (FileChannel channel = new RandomAccessFile(file, mode: "rw").getChannel()) {
            FileLock lock = channel.tryLock();
            if (lock == null) {
                return true; // al gelocked
            }
            lock.release();
            return false;
        } catch (Exception e) {
            return true; // fout bij openen = waarschijnlijk gelocked
        }
    }
}
```

private boolean isFileLocked(File file):

This utility method checks whether a file is currently locked by another process. It uses Java's FileChannel mechanism to try and acquire a lock on the file.

- new RandomAccessFile(file, "rw").getChannel() opens a read/write channel to the file.
- channel.tryLock() attempts to acquire an exclusive lock.

• If the lock is null, it means the file is already locked by another process → return true.
• If the lock is successfully acquired, it is immediately released and the method returns false.
• Any exception (e.g. file in use, permission denied) is interpreted as locked and results in return true.

This method is used in the polling loop to detect whether a file is externally opened in another program such as a text editor

TCP Message Handler

```
if (header.startsWith("GET_FILELIST")) {
    PrintWriter writer = new PrintWriter(socket.getOutputStream(), autoFlush: true);
    List<SyncAgent.FileEntry> entries = node.getSyncAgent().getFileList();

    for (SyncAgent.FileEntry entry : entries) {
        // nieuwe formaat: <filename>:<locked>:<owner>
        writer.println(entry.filename + ":" + entry.locked + ":" + entry.owner);
    }

    writer.println("END"); // Signaal dat alle entries verzonden zijn
    return;
}
```

The GET_FILELIST instruction is a dedicated TCP command used by a SyncAgent to request the most recent list of file metadata from another node. This includes the filename, its current lock status, and the file's owner. It plays a crucial role in enabling decentralized file synchronization across nodes in the ring.

Command Detection and Response Setup

- if (header.startsWith("GET_FILELIST")) This conditional block checks whether the incoming message starts with the GET_FILELIST keyword. If true, the node interprets the message as a synchronization request.

- PrintWriter writer = new PrintWriter(socket.getOutputStream(), true); A writer is created to send back a response over the established TCP connection. The autoFlush parameter is set to true, ensuring each call to println() is immediately pushed to the output stream.

5. Challenges Encountered

- Serialization issues: While sending the agent via TCP, serialization and classpath mismatches occasionally led to deserialization errors.
- Concurrency in shutdown: Race conditions between failure detection and node shutdown led to duplicate failure reports, which were resolved via locking mechanisms.
- Unreachable ports: Incomplete TCP shutdown from previously closed nodes caused false positives in failure detection.

Solution **Implementations:**

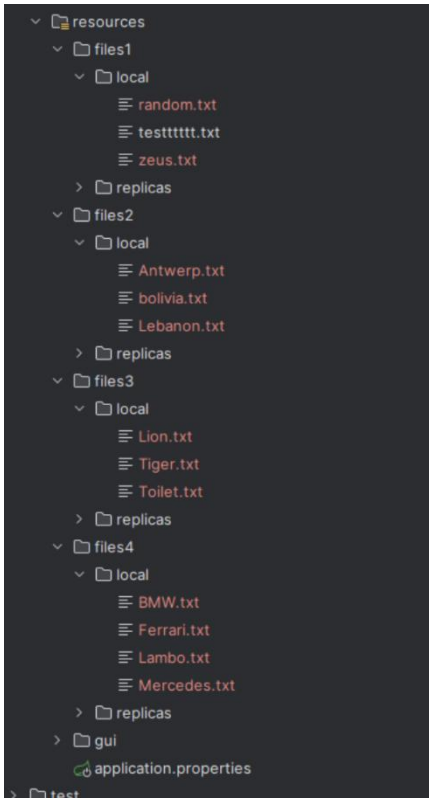
Thread.sleep(500);
• If connection to the neighbor fails, it logs an error.
• Sleeps for 500ms before repeating the sync process

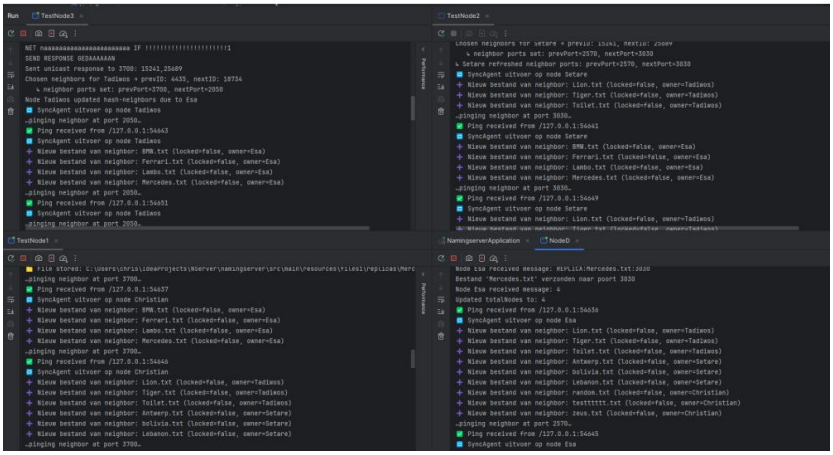
6. Testing and Results

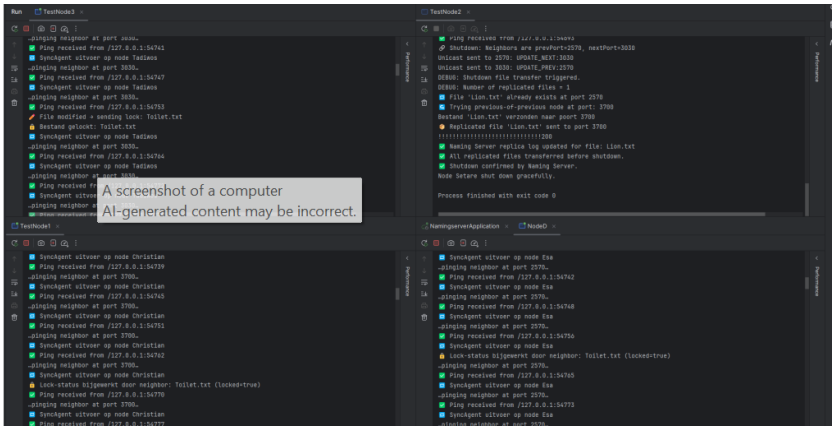
- Simulated failure by forcefully killing a node terminal.
- Observed FailureAgent being dispatched and received by neighbors.
- Neighbor connections (prev and next) were updated successfully.
- Replica reassignment occurred as expected. File ownership logs were updated.
- Verified clean log output indicating successful shutdown and re-replication.
- The system recovered from failure and remained operational without manual intervention.

When we start the nodes up we can clearly see that all of the syncing agents start running. They all sync their lists to that of their neighbours and update it if there are modifications, the lists will be updated (Not all of the updates are included because of the limited space).
When we go to modify a file, for example a file that is owned by TestNode3, We can see in the terminal that it is locked, and all the neighbouring agents will follow since the list of Node3 has been updated.

File Directory and logs (Node 2 shutting down):







(We tested it together so scenarios will be the same for some group members)

Scenario 1 – Failure of a Node with Local Files Preconditions:

• Node running on port 9060 owns the files Christian1.txt and filetjeeeuh.txt.

• Replicas of these files are stored on Node 3030.

Action:

• Manually terminate Node 9060.

Expected System Behavior:

- The replica on Node 3030 is promoted to become the new owner.
- A new replica is automatically generated on a different node, such as Node 4060.
- Node 9060 is entirely removed from the nodeMap, ensuring the ring remains consistent

Scenario 2 – Failure of a Node Holding Only Replicas

Preconditions:

Node 3030 contains only replica files and is not the owner of any data.• Simulate a node failure by

• Action:

shutting down Node 3030.

Expected System Behavior:

- Each replica previously held on Node 3030 is reassigned to another valid node.
- The FailureAgent ensures that the new replica is **not** placed on either the original file owner or the failed node, enforced by the condition:

  if (!candidate.equals(ownerHash) && !candidate.equals(failedHash)) { … }

- If the system has only two nodes remaining and no valid candidate is found, the system logs the message:

  No suitable new replica node found for file 'X'.

Scenario B – Node Crash: Replica-Only Node (Port 3030)

Preconditions:

- **Node 3030 functions solely as a replica holder and does not own any original files.**

Action:

• Node 3030 is forcefully terminated or crashes unexpectedly.

Expected Behavior:

• The system reassigns each replica that was hosted on Node 3030 to a new valid node in the ring.

• During reassignment, the original file owner and the failed node are excluded from selection, as
enforced in the logic.

• If the system is left with only two nodes and no eligible replica target exists, an error is logged for that

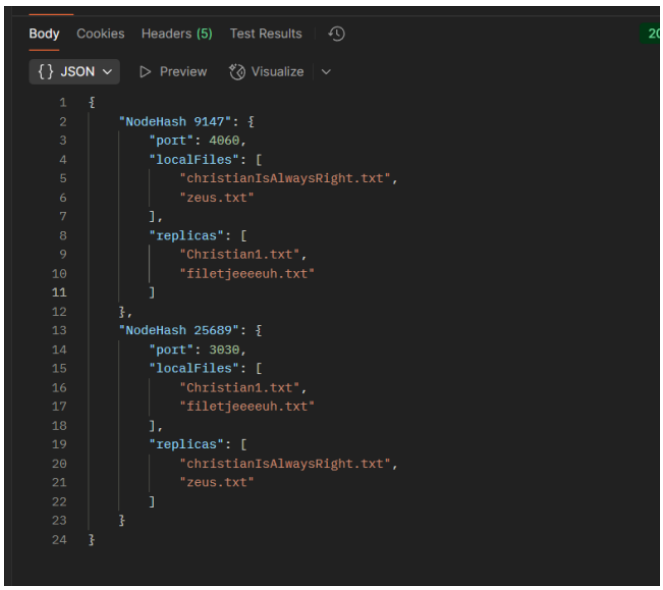
specific file:
No suitable new replica node found for file 'X'

• Finally, Node 3030 is removed from the nodeMap, completing the cleanup process.

Scenario C: Crash of Node 9060 (Only Local Files)

• **Precondition:** Node 9060 owns Christian1.txt and filetjeeeuh.txt; replicas live on Node 3030.
• **Action:** Node 9060 is stopped.
• **Result:**

o Replica on Node 3030 is promoted to owner.
o A new replica is created on Node 4060.
o Node 9060 is removed from nodeMap



Scenario D – Node Failure: Mixed Ownership and Replicas (Port 4060)
Preconditions:

- Node 4060 maintains a combination of locally owned files and replicas of files owned by other nodes.

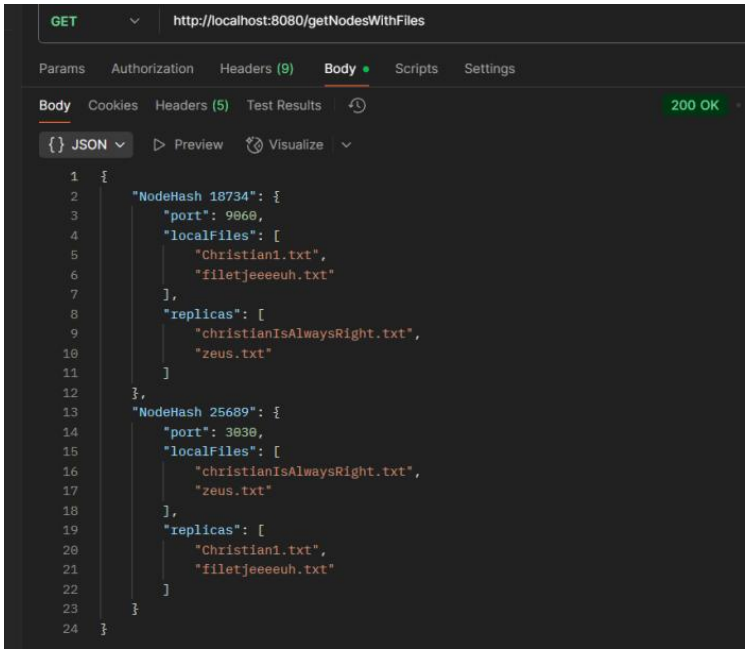
Action:

• Node 4060 is abruptly terminated or crashes.

Expected Behavior:

- All files for which Node 4060 was the owner are recovered through standard failure-handling: an existing
  replica is promoted to owner, and a new replica is designated elsewhere.
  • Files stored as replicas on Node 4060 are reassigned to new, eligible nodes, explicitly excluding both the original owner and the failed node.

  • Node 4060 is ultimately removed from the system's nodeMap



7. Conclusion

Lab 6 successfully demonstrated the implementation of fault-tolerant, agent-based failure detection in a distributed ring topology. By deploying autonomous agents that encapsulate logic and data, the system maintained high availability and consistency even in adverse conditions. This implementation lays the foundation for further dynamic behaviors such as automatic load balancing or hot standby nodes.

Lab – 7 GUI

1. Introduction

The goal of Lab 7 was to design and implement a **JavaFX-based GUI** to control, monitor, and manage a distributed file replication system. The GUI acts as an administrative console for the naming server, allowing users to add nodes, visualize node metadata (name, port, hash), retrieve stored files, and execute graceful shutdowns. This lab represents the transition from command-line interactions to an intuitive visual interface suitable for practical deployments in multi-host environments.
This GUI integrates closely with the existing Spring Boot REST backend developed in earlier labs, ensuring that all actions performed through the GUI trigger the appropriate backend logic.

2. Objective

- To create a **JavaFX GUI** connected to the distributed system backend.
- To allow dynamic **addition of nodes** with user-defined names and file counts.
- To visualize all **registered nodes** and their corresponding metadata.
- To enable fetching and display of **local and replicated files** per node.
- To allow **shutdown of nodes** with proper update of system state.
- To integrate REST-based communication between frontend and backend.

3. Key Functionalities Implemented

3.1 Add Node Interface

- Fields:
  - **Node Name**: Text input for naming the node.
  - **File Count**: Choice box (0–5) representing number of random .txt files to generate.
- On submit:
  - Generates a **random 4-digit port (×10)**.
  - Computes an index for folder allocation using current node count from server.
  - Files are generated in local directory and replica directory is prepared.
  - A **POST request** is sent to http://localhost:8080/addNode with node metadata.
  - Naming server creates the node using the backend logic and updates its internal maps.

MainController: hanleAddNode()

```
private void handleAddNode() {
    String name = nodeNameField.getText().trim();
    Integer fileCount = fileCountChoiceBox.getValue();

    if (name.isEmpty()) {
        showAlert( msg: "Please enter a node name.");
        return;
    }

    int port = generateRandomPort();
      int index = nodeCreationIndex++; // Static counter

    int index = fetchNodeCountFromServer() + 1;

      int index = Node.getTotalNodes();
    String basePath = "/home/tadiwos/6th_Semester/Distributed_Systems/Labs/Lab_7/GUI_Lab_7/namingserver/src/ma
    String localPath = basePath + "files" + index + "/local";
    String replicaPath = basePath + "files" + index + "/replica";

    // Ensure folders exist
    new File(localPath).mkdirs();
    new File(replicaPath).mkdirs();
```

Generating Random Files

```
try {
    generateFiles(localPath, fileCount);  // Create .txt files in local folder

    // === SEND TO NAMING SERVER VIA REST ===
    String jsonBody = String.format("""
            {
              "name": "%s",
              "port": %d,
              "localPath": "%s",
              "replicaPath": "%s"
            }
            """, name, port, localPath, replicaPath);

    String namingServerIP = "localhost";  // or use actual IP of G4c1.6dist
    URL url = new URL( spec: "http://" + namingServerIP + ":8080/addNode");
    HttpURLConnection con = (HttpURLConnection) url.openConnection();
    con.setRequestMethod("POST");
```

Generation of Random Port Numbers:

```
private int generateRandomPort() { return (new Random().nextInt( bound: 900) + 100) * 10; }

private void generateFiles(String folderPath, int count) throws IOException {  1 usage   tadiwos
    String chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";
    Random random = new Random();

    for (int i = 0; i < count; i++) {
        String filename = random.ints( streamSize: 8,  randomNumberOrigin: 0, chars.length()) IntStream
                .mapToObj( int index -> String.valueOf(chars.charAt(index))) Stream<String>
                .collect(Collectors.joining()) + ".txt";

        File file = new File(folderPath, filename);
        if (file.createNewFile()) {
            try (FileWriter writer = new FileWriter(file)) {
```

```
                try (FileWriter writer = new FileWriter(file)) {
                    writer.write( str: "Auto-generated file: " + filename);
                }
            }
        }
    }

    private String listLocalFiles(String folderPath) {  1 usage   tadiwos
        File folder = new File(folderPath);
        String[] files = folder.list(( File dir,  String name) -> name.endsWith(".txt"));
        return (files != null) ? Arrays.toString(files) : "[]";
    }

    private void showAlert(String msg) {  8 usages   tadiwos
        Alert alert = new Alert(Alert.AlertType.INFORMATION);
        alert.setTitle("Info");
        alert.setHeaderText(null);
        alert.setContentText(msg);
        alert.showAndWait();
    }
```

3.2 Node Table (Node Registry Viewer)

- Two TableViews display all registered nodes:
  - **nodeTable**: Displays real node name, port, and hash.
  - **nodeTable2**: Shows "Node@Port" format for selection when fetching files.
- Populated via GET /getAllNodes which returns the current nodeMap from the server.

3.3 File Viewer (ListView of Local and Replica Files)

- On selection of a node in nodeTable2, a REST call to:
  GET /getFilesForNode?nodeName=<name>

is made.

- Returns a JSON object containing "local" and "replica" file arrays.
- The GUI parses this and updates:
  - **Local Files View** – Lists all .txt files in that node's local folder.
  - **Replica Files View** – Lists all replicated files.

3.4 Configuration Viewer

- Fetches and displays the **previous and next node hash IDs** using:

GET /getNodeConfiguration?port=<port>

- The result is shown in a label or alert, assisting debugging and understanding of ring topology.

3.5 Graceful Node Shutdown

- On clicking "Shutdown" for a selected node:
  - GUI makes POST /removeNode with { name, port } payload.
  - NodeApp triggers:
    - Update of prev and next neighbors.
    - Deletion of local and replica files.
    - Optional reassignment of replicas if active.
  - Shutdown handled asynchronously in a Thread to prevent GUI freeze.
  - Naming server remains alive, and node is removed from registry.

4. GUI Architecture

Layer	Component	Function
Frontend	MainView.fxml	GUI Layout with VBox, TableView, etc.
Controller	MainController.java	Logic for handling UI interactions
Backend API	Spring Boot REST	/addNode, /getAllNodes, /removeNode
Backend Logic	NodeApp.java, ServerController	Node creation, registry, shutdowns

MainController.java –

Graphical User Interface (GUI)

The goal of the GUI is to abstract away the underlying network and file operations, allowing users to add nodes, configure them with files, and inspect their status through intuitive controls.
The GUI is structured around two main components:

- **FXML files**, which define the visual layout using declarative markup.
- **The MainController class**, which handles all user interactions and connects to the backend through REST APIs.

This separation of concerns ensures that both the design and logic remain clean and maintainable.

Design Structure

- **FXML Components**:
  All interface elements—such as input fields, dropdowns, and tables—are defined within FXML files. This layout-based approach allows for quick styling and adjustments without affecting program logic.
- **Controller Logic**:
  The MainController class handles all backend communication and user-triggered events. By linking FXML elements via @FXML annotations, this controller manages the interaction between the GUI and the REST endpoints of the naming server.

5. Challenges Encountered

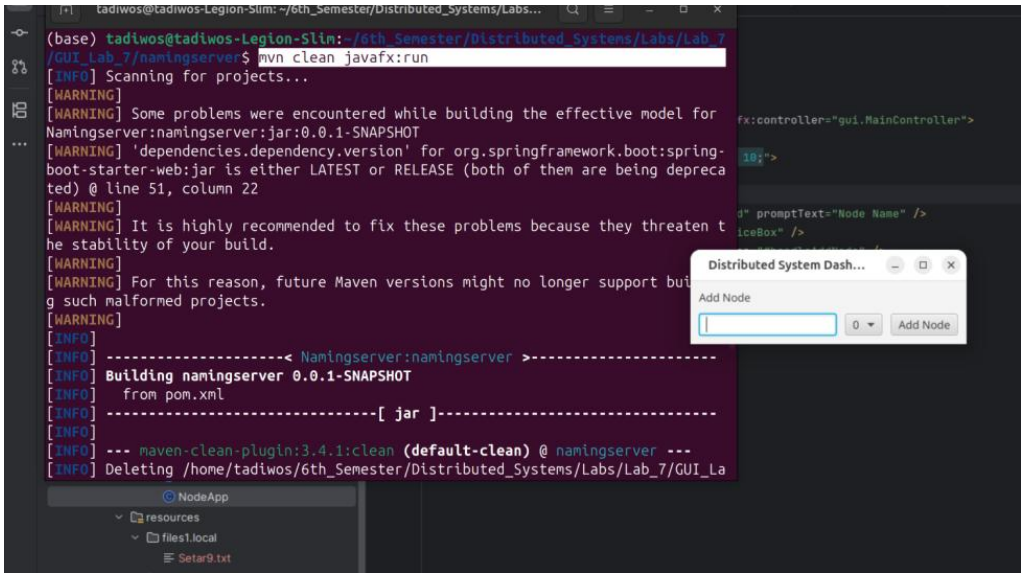
- **Concurrency:** Shutdown initially blocked GUI using synchronous HttpURLConnection. Fixed using background threads.
- **Duplicate Indexing:** Multiple GUI instances created nodes at index 1. Fixed by REST-based node count retrieval.
- **FXML Linking Errors:** Mismatches between controller IDs and MainView.fxml elements caused null pointer exceptions.
- **Replica File Fetching Bug:** Initial parsing logic showed all files under replica view. Fixed with precise JSON parsing and key matching.
- **Graceful Shutdown Race Condition:** Even though the backend test of the Shutdown works perfectly fine, the test using GUI attempted System.exit(0) prematurely, entire naming server would close. To address System.exit(0) was removed, but this caused a bug where the File transfer of the replica never happens. The bug hasn't been identified during the time or writing this report.

6. Testing and Demonstration

- GUI successfully added multiple nodes using the visual interface.
- File generation verified at correct file paths.
- File fetcher displayed .txt files as expected from local and replica directories.
- Node registry updated dynamically upon addition/removal.
- Shutdown confirmed to reconfigure neighbors and remove node from registry

Node Creation with File Generation

Users can create new nodes by entering a custom name and selecting the number of .txt files to generate (between 0 and 5). These files are randomly named and stored in a separate folder for each node. The folder structure follows the format filesX/local and filesX/replica, where X is the index determined by the total number of nodes, fetched dynamically from the naming server via a REST call.
Each new node is assigned a random four-digit port number (ending in zero) to prevent port collisions during multi-node deployments. Once the local and replica folders are created and populated with files, the GUI composes a JSON object with the node's metadata and submits it to the naming server via an HTTP POST request to the /addNode endpoint. This triggers the multicast announcement and starts the internal replication logic on the backend.

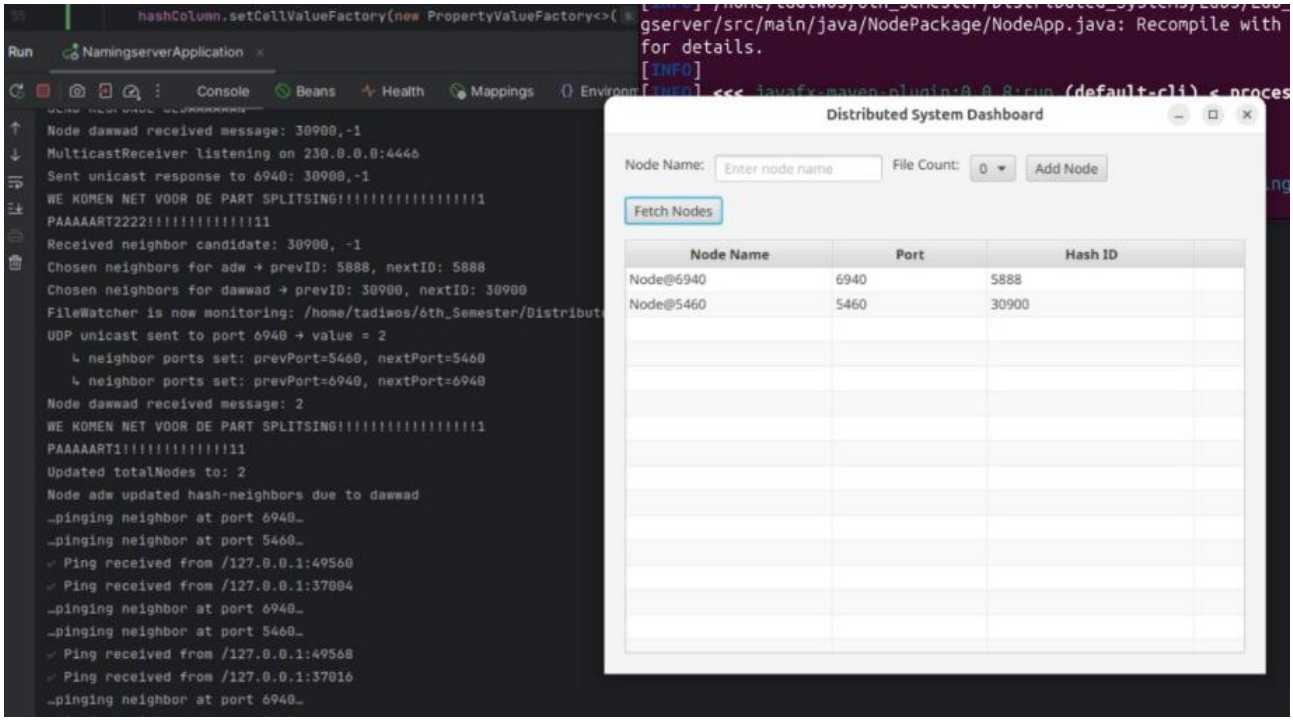


Fetching, removing and Displaying Active Nodes

The interface includes a table component that lists all registered nodes in the system. Each row in the table displays the node's:

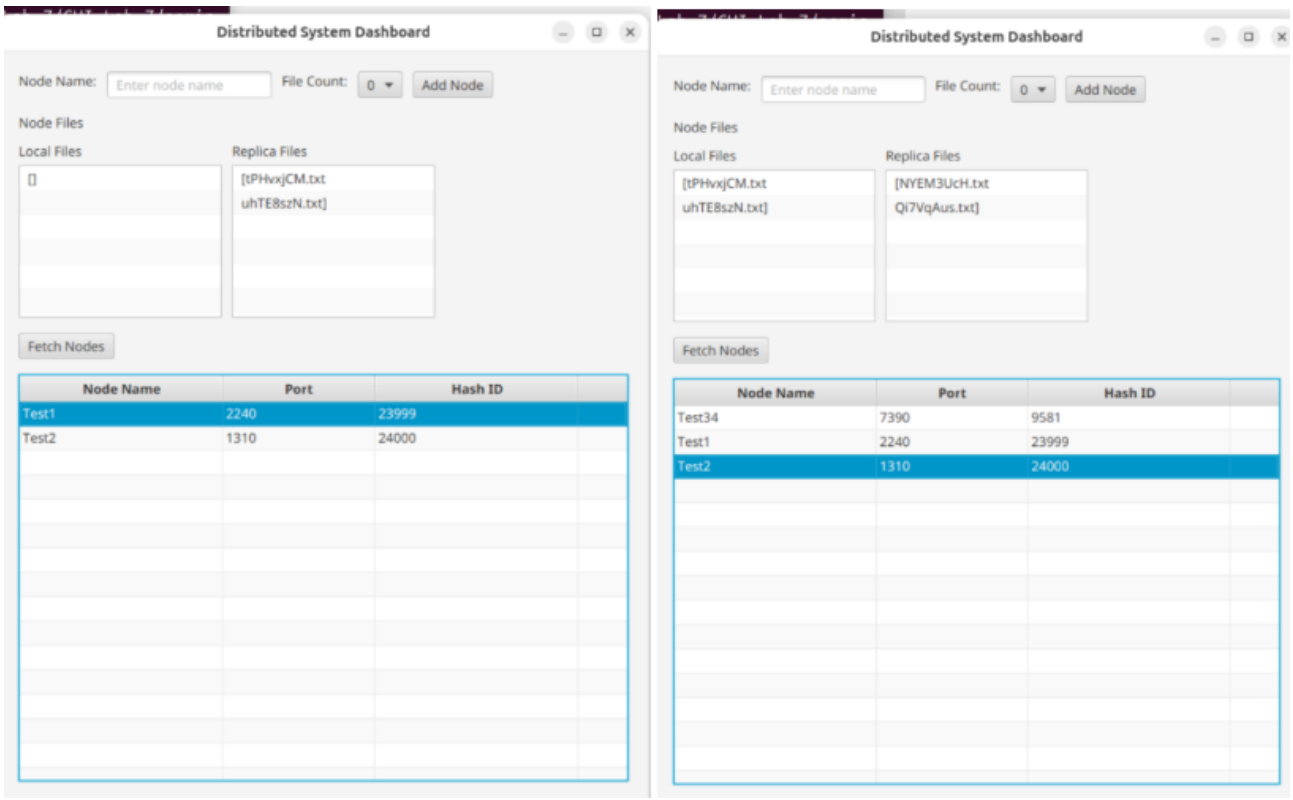
- Name (entered by the user)
- TCP port number
- Consistent hash (computed from the name)

This data is fetched through a GET request to /getAllNodes and parsed into the JavaFX TableView using an observable list of NodeDisplay objects.
The GUI includes logic to remove an existing node from the ring. When a user removes a node, it is deregistered from the system and its entry is removed from the interface. This ensures that the ring stays consistent and that files are redistributed according to replication logic.

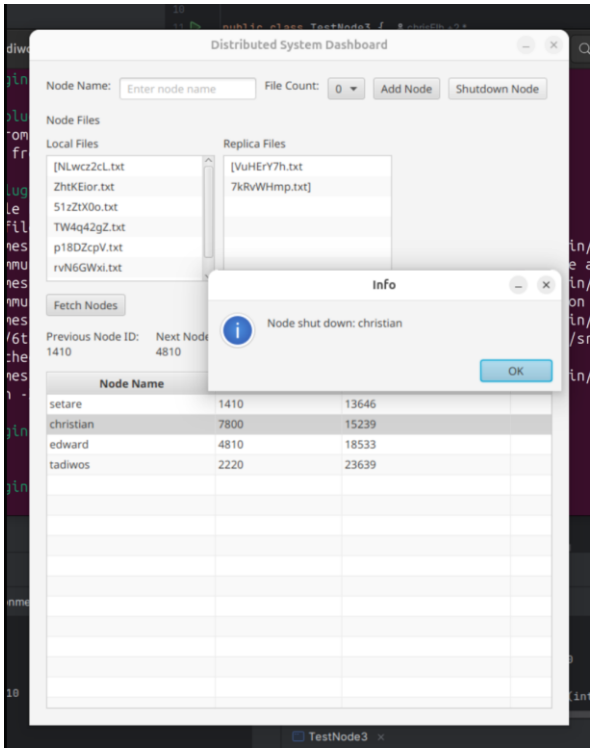


Viewing Node Files

After selecting a node, the GUI can display all local and replica files associated with it. This provides insight into file distribution and replication across the ring. The system distinguishes between original files and backup copies, reflecting the underlying replication logic.



Shutdown Button



7. Conclusion

Lab 7 successfully extended the distributed system with a **complete graphical user interface** built using JavaFX and REST. The GUI supports real-time node management, system monitoring, and graceful shutdown operations. It allows end-users (or testers on remote containers) to manage the system without direct code modifications, greatly improving usability, maintainability, and observability of the entire system. The Shutdown as mentioned earlier, has a bug where the backend testing but for some reason the GUI introduces a bug that skips the file transfer of the replica files. This is issue hasn't been addressed during the time of writing this paper