University of Houston-Clear Lake

College of Science and Engineering

Project Report

Advanced Operating Systems

Distributed Peer to Peer System

Umar Abdul Aziz
2310655

TABLE OF CONTENTS

I. INTRODUCTION 3

II. DATA STRUCTURE AND ALGORITHMS 5

III. PROBLEM SOLVING – INDEX SERVER & USER DEFINED FUNCTIONS AND METHODS 7

IV. PROBLEM SOLVING – PEER & USER DEFINED FUNCTIONS AND METHODS 12

V. DIAGRAMATIC REPRESENTATION OF THE SYSTEM 17

VI. SCREENSHOTS 22

VII. CONCLUSION 25

INTRODUCTION

Peer-to-peer (P2P) networks provide an effective way to share resources decentralized in the context of contemporary distributed systems, instead of depending on a centralized server for direct file delivery. Through the use of TCP and/or UDP protocols, a robust communication system between peers and an index server will be implemented in this project's distributed peer-to-peer group-based content-sharing system. To enable effective client-server communication, the system is implemented in Java, taking advantage of its concurrency features.

The Master Index-Server and the Peer-to-Peer client processes are the two primary functions that need to be built as part of the project for the CSCI 5531 Advanced Operating Systems course. In addition to handling user registration and login, the Master Index-Server serves as a TCP concurrent server and informs users about the availability of requested content. Peer-to-peer processes enable a distributed and cooperative file-sharing environment by hosting shared material and requesting files from other peers in the network.

The IP address and port number provided by the user are used by the Peer-to-Peer client process to connect to the Index Server when it first launches. Through registration or login, the Index Server verifies the user's identity. Once authentication has been completed successfully, the user can look for specific content inside the group. The Index Server locates the resource and gives the required details for downloading if the requested content is accessible to any of the group members. The download procedure can then be started by the peer-to-peer client by establishing a direct TCP connection with the peer that was found to have the required files.

The Peer-to-Peer process complies with these specifications by having two child threads: one serves as the client and communicates with the Index Server to start downloads, while the other serves as the music/file server and shares files with other peers upon request. The utilization of a multi-threaded technique guarantees effective handling of both incoming and outgoing connections, hence promoting smooth content exchange among peers.

This project's implementation centers on a number of important features of distributed systems, including scalability, secure communication, and concurrent connection management. The project highlights the importance of the decentralized resource management in contemporary networked contexts by providing a hands-on investigation of distributed architectures and protocols within the framework of a peer-to-peer content-sharing system.

DATA STRUCTURE AND ALGORITHMS

In the implementation of the peer-to-peer file-sharing system using **Java socket programming**, two key HashMaps are utilized for managing **user credentials** and **file information**. These data structures play a critical role in enabling fast lookups, efficient storage, and dynamic updates as users and files change within the system.

1. HashMap for User Credentials

A **HashMap** is used to store the user credentials, where the **username** acts as the key, and the value is a `User` object containing the user's **email** and **password**.

```
HashMap<String, User> users = new HashMap<>();
```

- **Key**: `String` representing the **username**.
- **Value**: A `User` object that stores the user's **email** and **password**.

- **Purpose**:
  - The HashMap allows for efficient **O(1)** lookup of user information by username. This is crucial during operations like **user registration** and **login**, where the system needs to quickly verify whether a username already exists or whether the password matches the stored credentials.

2. HashMap for File Information

Another **HashMap** is used to store information about the files being shared by peers. The **filename** serves as the key, and the value is a combination of the **IP address** and **port** of the peer sharing that file.

```
HashMap<String, FileInfo> files = new HashMap<>();
```

- **Key**: `String` representing the **filename**.
- **Value**: A `FileInfo` object that stores the **IP address** and **port number** of the peer sharing the file.

The `FileInfo` class is structured as:

- **Purpose**:
    - The HashMap enables **O(1)** access to file-sharing information. When a peer requests a file, the system can quickly look up the file's name and retrieve the **IP address** and **port** of the peer offering the file, enabling direct peer-to-peer communication for file transfers.

Benefits of Using HashMaps

- **Constant-Time Lookups**: Both HashMaps provide efficient **O(1)** average time complexity for search, insert, and delete operations. This is vital in a distributed system where frequent user authentication and file lookups are required.
- **Dynamic Updates**: Peers can join, share files, or leave at any time. HashMaps allow easy and fast insertion, removal, or update of user credentials and file information without significant overhead.
- **Scalability**: As the number of users and shared files grows, HashMaps continue to offer fast access times, making them suitable for large-scale systems with many peers.

By leveraging these data structures, the system ensures that user management and file-sharing operations are performed efficiently, supporting a smooth and scalable peer-to-peer experience.

PROBLEM SOLVING - INDEX SERVER & USER-DEFINED FUNCTIONS AND METHODS

Overview

The `IndexServer` class is a server application that handles requests related to user registration, user login, content searching, and file loading. It is designed to run as a multi-threaded server, capable of managing multiple client connections simultaneously. The server listens on a specified port and uses different commands to process client requests, including registering new users, logging in, searching for content, and loading files.

Class Summary

`IndexServer`

This is the main class that sets up and runs the server, as well as manages user and content data.

Key Features:

- Starts a server that listens for client requests.
- Handles user registration and login.
- Manages a content database to store content names and their locations.
- Loads user data from a file and persists new registrations to the same file.
- Loads content files and stores them in the server's content database.

Class Components

Fields

- `private static final int INDEX_SERVER_PORT = 10655`
  - The port number on which the index server listens for client connections.
- `private static int clientCounter = 0`
  - Keeps track of the number of clients connected to the server.
- `private final Map<String, String> userDatabase`

    - Stores the user data with the username as the key and the password as the value.
- `private final Map<String, String> contentDatabase`
    - Stores the content data with the content name as the key and the content location (IP address and port) as the value.

Constructor

- `public IndexServer()`
    - Initializes the `userDatabase` by loading existing users from a file.
    - Adds some example content to the `contentDatabase`.

Methods

`public void start()`

- Starts the server and listens for incoming client connections.
- For each client connection, creates a new thread to handle the request.

`private class IndexRequestHandler implements Runnable`

- This inner class handles client requests. Each instance of this class runs in a separate thread.

Fields

- `private final Socket socket`
    - The client socket through which communication takes place.

Methods

- `public void run()`
    - Handles client commands received through the socket.
    - Processes different commands, including `REGISTER`, `LOGIN`, `LOADFILE`, and `SEARCH`.
    - Uses switch-case to process different commands received from the client.
- `private void registerUser(BufferedReader in, PrintWriter out) throws IOException`
    - Handles user registration.

    - Reads user details (name, phone, email, username, password) from the client.
    - If the username is unique, stores the user in `userDatabase` and writes the user data to `userList.txt`.
    - Sends REGISTER_SUCCESS or REGISTER_FAILURE to the client based on the result.
- `private int loginUser(BufferedReader in, PrintWriter out) throws IOException`
    - Handles user login.
    - Reads username and password from the client.
    - Checks if the provided credentials match those in the `userDatabase`.
    - Sends AUTH_SUCCESS and an assigned port number if the login is successful; otherwise sends AUTH_FAILURE.
- `private void searchContent(String command, PrintWriter out)`
    - Handles content search requests.
    - Searches the `contentDatabase` for the requested content name.
    - Sends FOUND with the content location or NOT_FOUND based on the search result.
- `private void saveUserToFile(String name, String phone, String email, String username, String password)`
    - Saves user data to the `userList.txt` file.
    - Appends new user information to the file.
- `private void loadUsersFromFile()`
    - Loads users from `userList.txt` into `userDatabase`.
    - Reads each line, splits it by commas, and stores the username and password in the database.
- `private void loadFiles(BufferedReader in, PrintWriter out)`
    - Handles loading of files from clients.
    - Reads a CSV string containing content names and locations from the client.
    - Adds the content to `contentDatabase` and appends it to `fileList.txt`.
    - Sends FILES_ADDED to the client once the files are successfully added.

Main Method

- `public static void main(String[] args)`
    - Creates an instance of `IndexServer` and starts the server.

Example Commands

Commands Handled by the Server

1. **REGISTER**: Registers a new user.
   - Expected input: `REGISTER`, followed by user details (name, phone, email, username, password).
   - Response: `REGISTER_SUCCESS` or `REGISTER_FAILURE`.
2. **LOGIN**: Logs in an existing user.
   - Expected input: `LOGIN`, followed by username and password.
   - Response: `AUTH_SUCCESS` and an assigned port number, or `AUTH_FAILURE`.
3. **SEARCH**: Searches for content in the server's database.
   - Expected input: `SEARCH <contentName>`.
   - Response: `FOUND <contentLocation>` or `NOT_FOUND`.
4. **LOADFILE**: Loads files from the client to the server.
   - Expected input: `LOADFILE`, followed by CSV data representing content names and their locations.
   - Response: `FILES_ADDED`.

Additional Notes

- **Threading**: Each client request is handled in a separate thread, allowing multiple clients to connect to the server simultaneously.
- **Persistence**: User data and content data are stored in `userList.txt` and `fileList.txt`, respectively, allowing for persistent storage between server restarts.
- **Error Handling**: The server gracefully handles errors, such as client disconnections, by catching `IOException` and logging the error.

Example Workflow

1. A client connects to the `IndexServer`.
2. The client sends a `REGISTER` command to create a new account.
3. The client logs in using the `LOGIN` command, and if successful, the server assigns a unique port.
4. The client sends a `SEARCH` command to search for specific content.
5. The client can also send a `LOADFILE` command to add new content to the server.

This documentation provides an overview and detailed explanation of the `IndexServer` class, its purpose, and its implementation. It highlights how to interact with the server, the functionality provided, and how the different components of the code work together to achieve the desired functionality.

PROBLEM SOLVING - PEER & USER-DEFINED FUNCTIONS AND METHODS

Overview

This code represents the client-side implementation of a Peer in a Peer-to-Peer (P2P) network. The Peer communicates with the Index Server to register, log in, and share content information, and interacts with other peers for file sharing. It can perform functions like authentication, content registration, content search, and downloading files from other peers.

Components

- **Main Method**: The entry point for the Peer application.
- **Authenticate User**: Handles user registration and login.
- **Start Peer Server**: Starts a server to listen for incoming file requests from other peers.
- **Load Files**: Sends content details to the Index Server for registration.
- **Download Content**: Downloads a file from a peer.

Code Components Explained

1. Main Method

The main method is the starting point for the Peer application.

- It prompts the user for the IP address and port number of the Index Server.
- Connects to the Index Server and handles user authentication.
- If authentication is successful, it starts a server socket on the assigned port.
- Handles the registration of available content with the Index Server.
- Provides options to search for and download files from other peers.

2. Index Server Connection

```
Java code
Socket socket = new Socket(indexServerIP, indexServerPort);
```

This line connects the Peer to the Index Server using the provided IP and port.

3. User Authentication

- The `authenticateUser()` function handles both user registration and login.
- Depending on the user's choice, it calls either `loginUser()` or `registerUser()`.
- The server then responds with AUTH_SUCCESS or REGISTER_SUCCESS, indicating whether the authentication was successful.

```
Java code
if (response.equals("FAILURE")) {
    System.out.println("Authentication failed.");
}
```

If authentication fails, the application terminates.

4. Start Peer Server

```
Java code
new Thread(() -> startPeerServer(assignedPort)).start();
```

After successful login, a new thread is started to host a Peer server at the assigned port. The `startPeerServer(int port)` method listens for incoming requests from other peers to share content.

5. Load Files

- The `loadFiles()` method reads all files from a directory named `ContentFile` and registers them with the Index Server.
- This helps other peers to find the content hosted by this Peer.

```
Java code
File contentFolder = new File("ContentFile");
```

This line defines the directory where the files to be shared are stored.

6. Search and Download Content

- Once authenticated, the Peer can search for content using the SEARCH command.
- If the content is found, it displays the peer's address hosting that content.
- The user can then initiate a download by providing the content name and IP address of the peer.

```
Java code
downloadContent(downloadName.split(" ")[1], downloadName.split("
")[0]);
```

This line initiates the download from the provided IP address.

7. `FileRequestHandler` Inner Class

- This class is used to handle file requests from other peers.
- It reads the requested filename from the incoming connection and sends the file if it exists.

```
Java code
new Thread(new FileRequestHandler(clientSocket)).start();
```

Each incoming file request is handled by starting a new thread, which runs an instance of `FileRequestHandler`.

8. Downloading Content

The downloadContent`(String peerInfo, String fileName)` method is used to download files from another peer.

- The method connects to the peer at the specified IP and port.
- If the file is available, it is received and stored in a local directory named `receivefiles`.

```
Java code
File receiveDir = new File("receivefiles");
if (!receiveDir.exists()) {
    receiveDir.mkdir();
}
```

This block creates the directory to store received files if it does not already exist.

Class Details

`authenticateUser(BufferedReader userInput, PrintWriter out, BufferedReader in)`

Handles the user authentication flow by either logging in or registering the user.

`loginUser(BufferedReader userInput, PrintWriter out, BufferedReader in)`

Prompts the user for email and password to log in, then sends the credentials to the Index Server.

`registerUser(BufferedReader userInput, PrintWriter out, BufferedReader in)`

Prompts the user for their details (name, phone, email, username, and password) and registers them with the Index Server.

`startPeerServer(int port)`

Starts a server socket at the specified port and listens for incoming file requests from other peers.

`loadFiles(Socket socket, int assignedPort, PrintWriter out)`

Loads available files from the `ContentFile` directory and registers them with the Index Server in CSV format.

`FileRequestHandler implements Runnable`

Handles incoming requests from peers for file downloads. If the requested file exists, it sends the file to the requesting peer.

`sendFile(File file, Socket socket)`

Reads a file from disk and sends it over the socket connection.

`downloadContent(String peerInfo, String fileName)`

Handles downloading a file from another peer given its IP address and port number. Saves the downloaded file in the `receivefiles` directory.

Example Flow

1. **Start Application**: The user provides the Index Server's IP and port.
2. **Authenticate**: The user logs in or registers.
3. **Register Files**: Available files are registered with the Index Server.
4. **Search and Download**:
    - The user searches for a file.
    - If found, the user downloads the file from the specified peer.

Error Handling

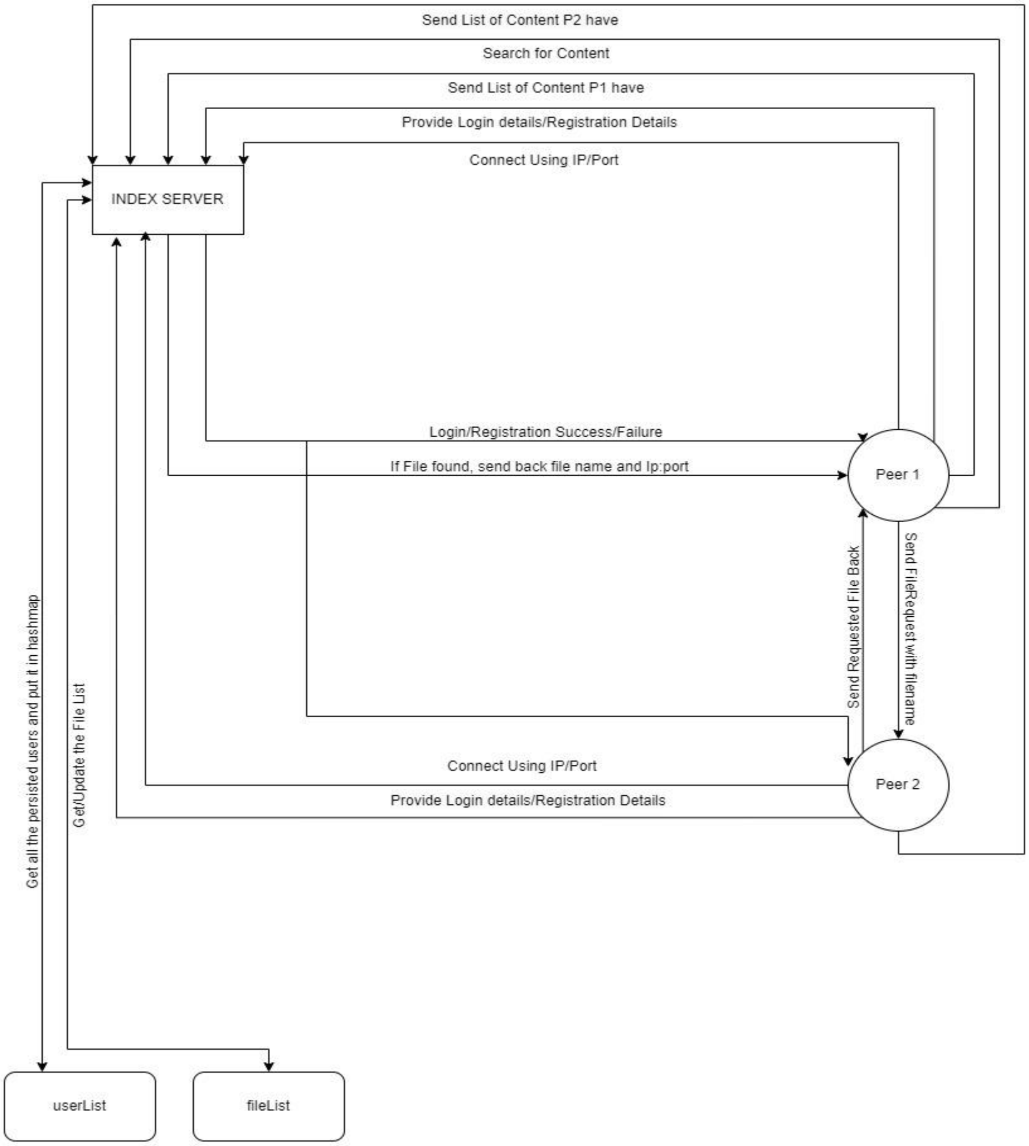
- The program handles various `IOException` cases during network communication, file reading, and writing.
- If authentication fails, the user is informed and given the option to exit.

Considerations

- **Security**: Passwords are transmitted in plain text, which is not secure. Consider encrypting sensitive information.
- **Scalability**: The current implementation assigns a unique port to each peer. For a larger network, dynamic port allocation or load balancing might be necessary.
- **Exception Handling**: More specific exception handling could be added for better error diagnostics.

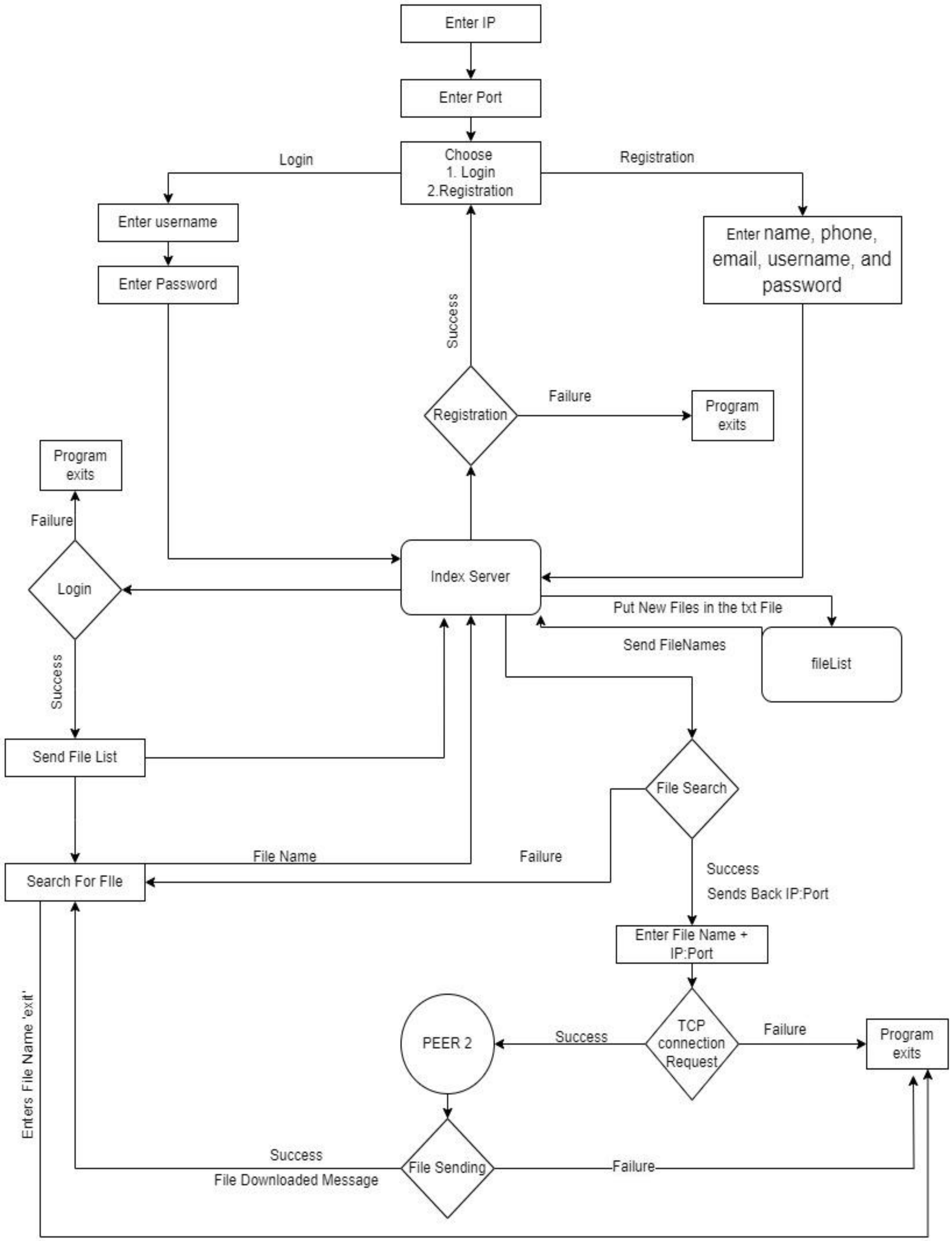
This documentation provides an overview of how the Peer class operates in a P2P file-sharing environment, describing each component's role in the process of content sharing and retrieval.

DIAGRAMATIC REPRESENTATION OF THE SYSTEM

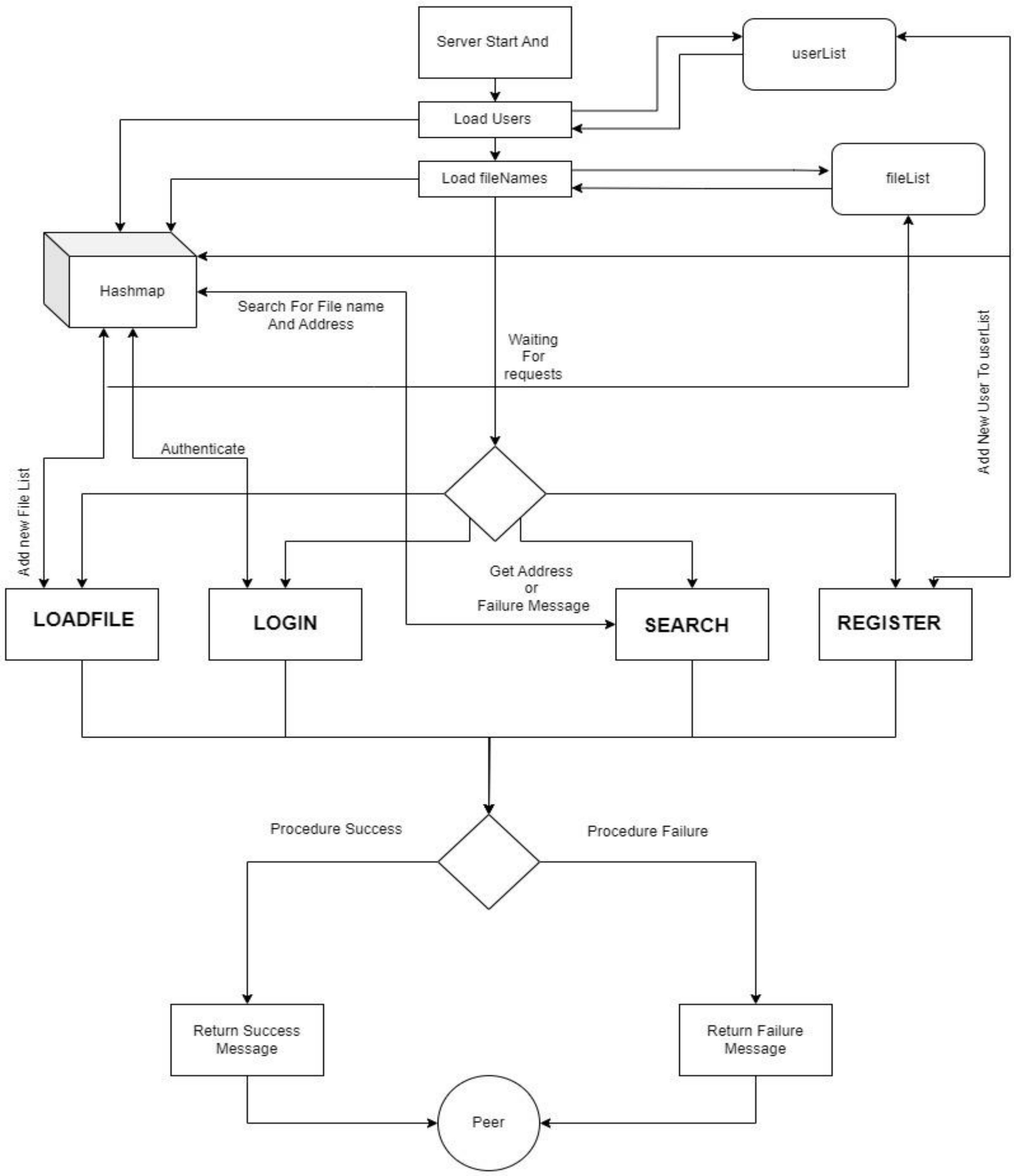

Send List of Content P2 have
Search for Content
Send List of Content P1 have
Provide Login details/Registration Details
Connect Using IP/Port
INDEX SERVER
Login/Registration Success/Failure
If File found, send back file name and Ip:port
Peer 1
Send Requested File Back
Send FileRequest with filename
Connect Using IP/Port
Provide Login details/Registration Details
Peer 2
Get all the persisted users and put it in hashmap
Get/Update the File List
userList
fileList

Distributed Peer To Peer System

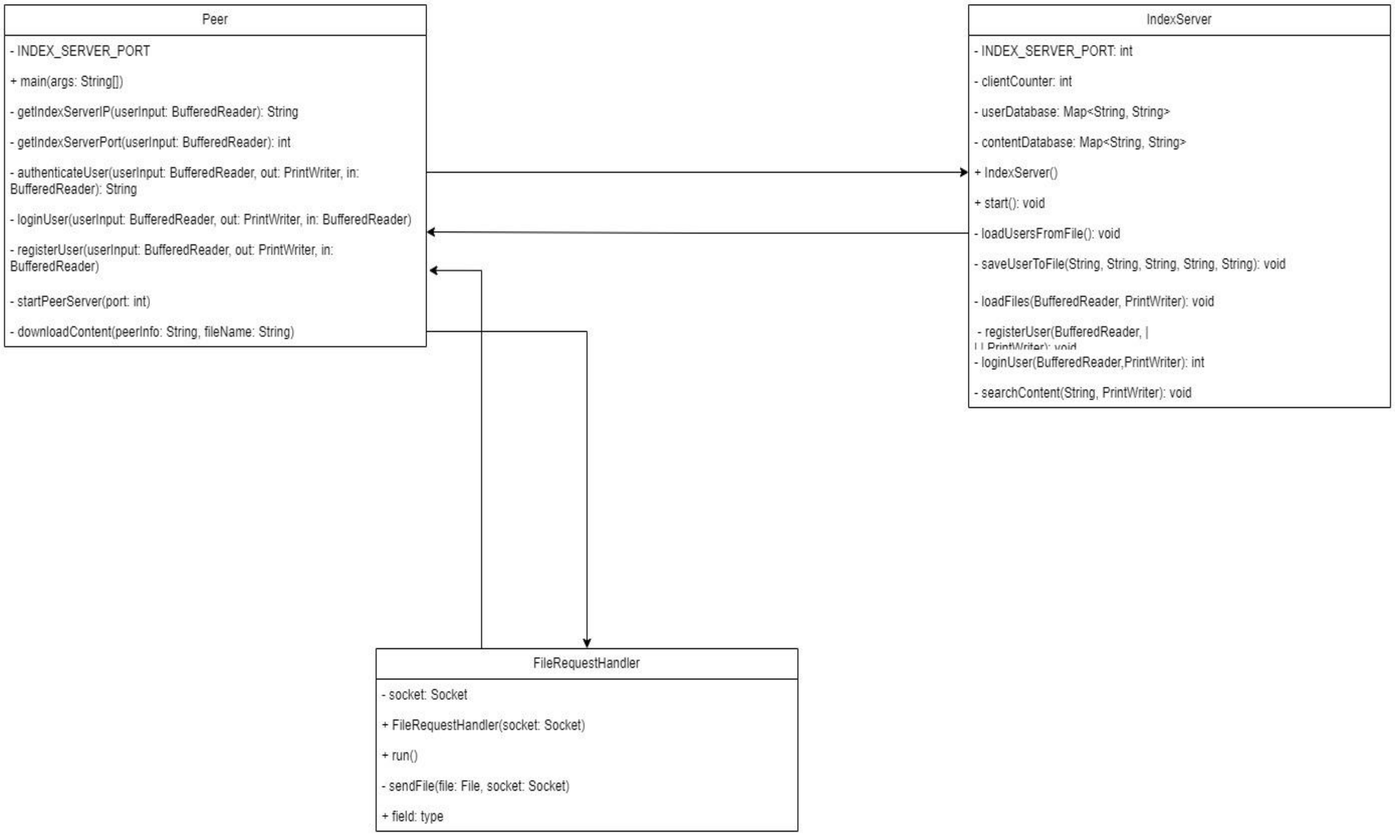
Flow Chart For Peer



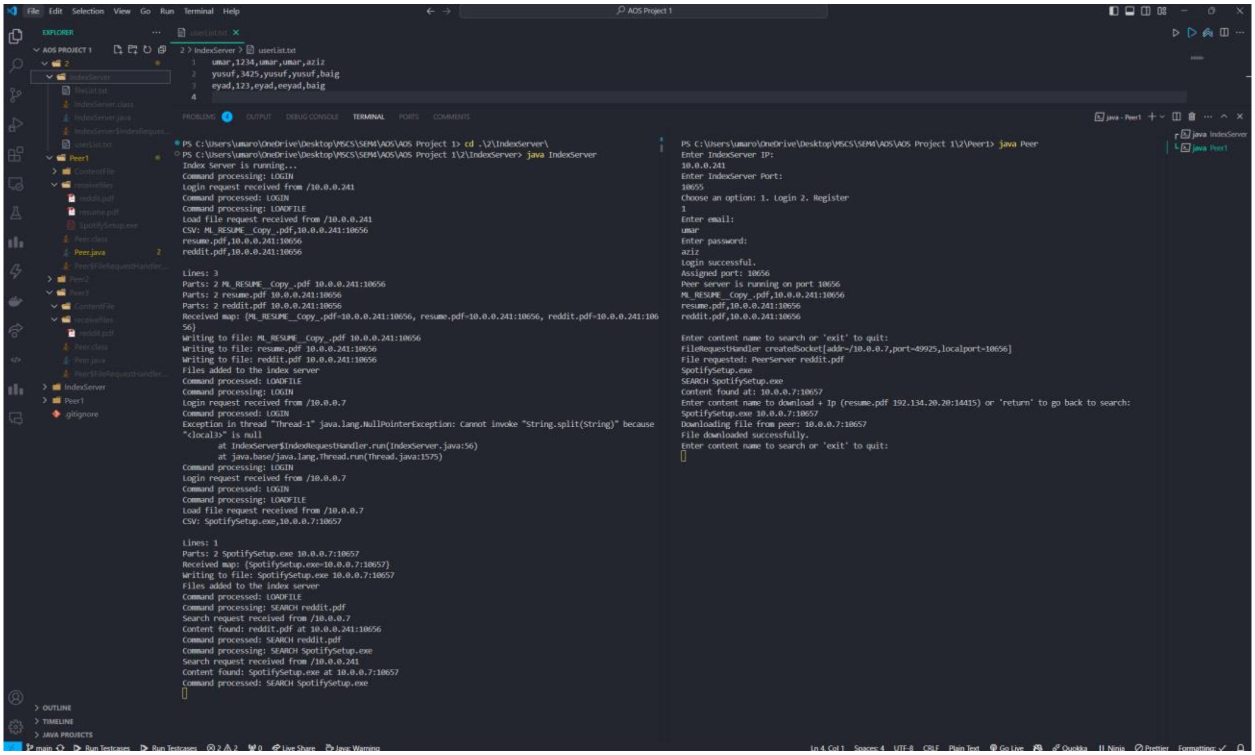
Flow Chart For Index Server



Class Diagram


Peer
- INDEX_SERVER_PORT
+ main(args: String[])
- getIndexServerIP(userInput: BufferedReader): String
- getIndexServerPort(userInput: BufferedReader): int
- authenticateUser(userInput: BufferedReader, out: PrintWriter, in: BufferedReader): String
- loginUser(userInput: BufferedReader, out: PrintWriter, in: BufferedReader)
- registerUser(userInput: BufferedReader, out: PrintWriter, in: BufferedReader)
- startPeerServer(port: int)
- downloadContent(peerInfo: String, fileName: String)
IndexServer
- INDEX_SERVER_PORT: int
- clientCounter: int
- userDatabase: Map<String, String>
- contentDatabase: Map<String, String>
+ IndexServer()
+ start(): void
- loadUsersFromFile(): void
- saveUserToFile(String, String, String, String, String): void
- loadFiles(BufferedReader, PrintWriter): void
- registerUser(BufferedReader, |
| | PrintWriter): void
- loginUser(BufferedReader,PrintWriter): int
- searchContent(String, PrintWriter): void
FileRequestHandler
- socket: Socket
+ FileRequestHandler(socket: Socket)
+ run()
- sendFile(file: File, socket: Socket)
+ field: type

SCREENSHOTS

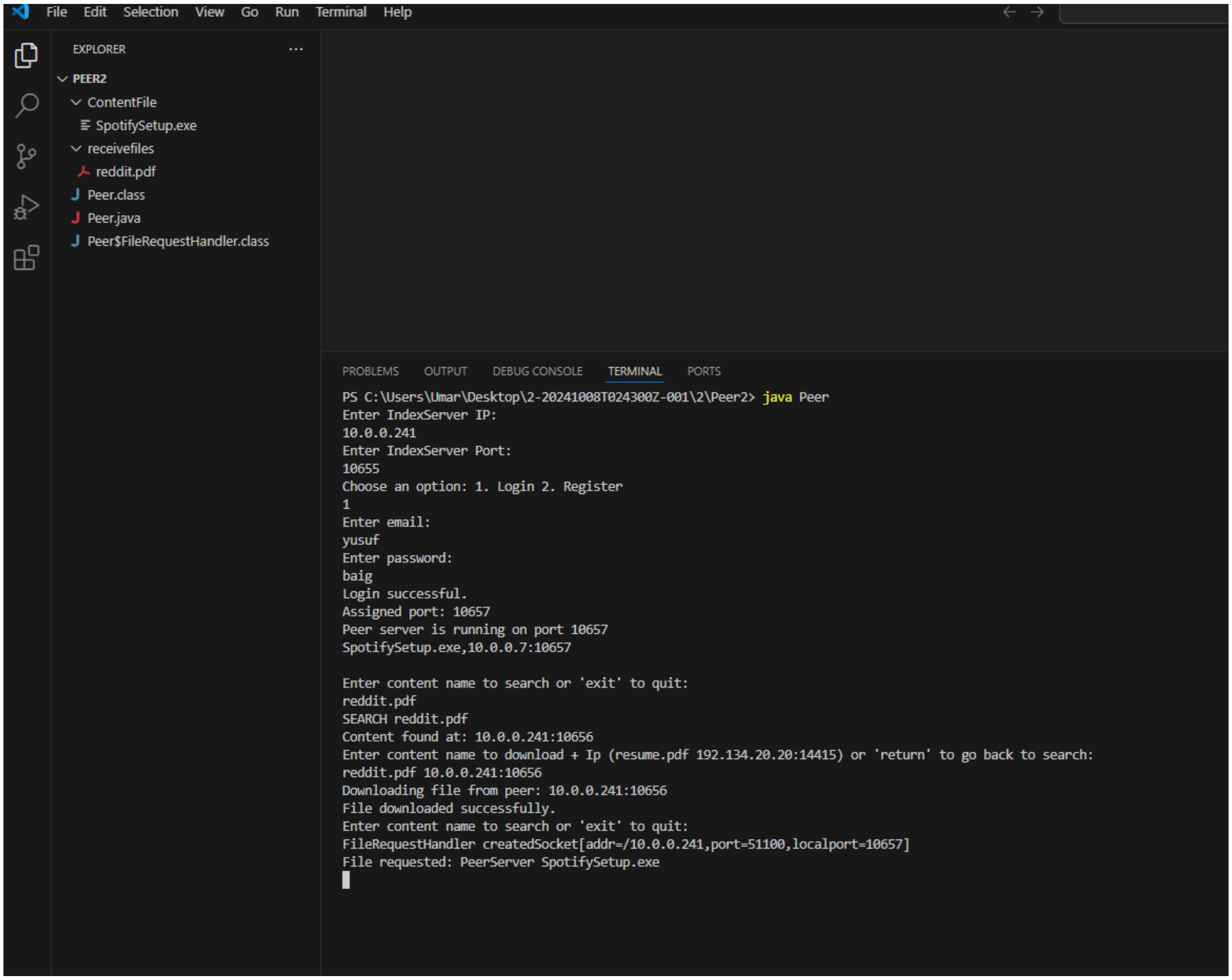


```
umar,1234,umar,umar,aziz
yusuf,3425,yusuf,yusuf,baig
eyad,123,eyad,eeyad,baig
```

```
PS C:\Users\umaro\OneDrive\Desktop\MSCS\SEM4\AOS\AOS Project 1> cd .\2\IndexServer\
PS C:\Users\umaro\OneDrive\Desktop\MSCS\SEM4\AOS\AOS Project 1\2\IndexServer> java IndexServer
Index Server is running...
Command processing: LOGIN
Login request received from /10.0.0.241
Command processed: LOGIN
Command processing: LOADFILE
Load file request received from /10.0.0.241
CSV: ML_RESUME__Copy_.pdf,10.0.0.241:10656
resume.pdf,10.0.0.241:10656
reddit.pdf,10.0.0.241:10656

Lines: 3
Parts: 2 ML_RESUME__Copy_.pdf 10.0.0.241:10656
Parts: 2 resume.pdf 10.0.0.241:10656
Parts: 2 reddit.pdf 10.0.0.241:10656
Received map: {ML_RESUME__Copy_.pdf=10.0.0.241:10656, resume.pdf=10.0.0.241:10656, reddit.pdf=10.0.0.241:106
56}
Writing to file: ML_RESUME__Copy_.pdf 10.0.0.241:10656
Writing to file: resume.pdf 10.0.0.241:10656
Writing to file: reddit.pdf 10.0.0.241:10656
Files added to the index server
Command processed: LOADFILE
Command processing: LOGIN
Login request received from /10.0.0.7
Command processed: LOGIN
Exception in thread "Thread-1" java.lang.NullPointerException: Cannot invoke "String.split(String)" because
"<local3>" is null
        at IndexServer$IndexRequestHandler.run(IndexServer.java:56)
        at java.base/java.lang.Thread.run(Thread.java:1575)
Command processing: LOGIN
Login request received from /10.0.0.7
Command processed: LOGIN
Command processing: LOADFILE
Load file request received from /10.0.0.7
CSV: SpotifySetup.exe,10.0.0.7:10657

Lines: 1
Parts: 2 SpotifySetup.exe 10.0.0.7:10657
Received map: {SpotifySetup.exe=10.0.0.7:10657}
Writing to file: SpotifySetup.exe 10.0.0.7:10657
Files added to the index server
Command processed: LOADFILE
Command processing: SEARCH reddit.pdf
Search request received from /10.0.0.7
Content found: reddit.pdf at 10.0.0.241:10656
Command processed: SEARCH reddit.pdf
Command processing: SEARCH SpotifySetup.exe
Search request received from /10.0.0.241
Content found: SpotifySetup.exe at 10.0.0.7:10657
Command processed: SEARCH SpotifySetup.exe
```

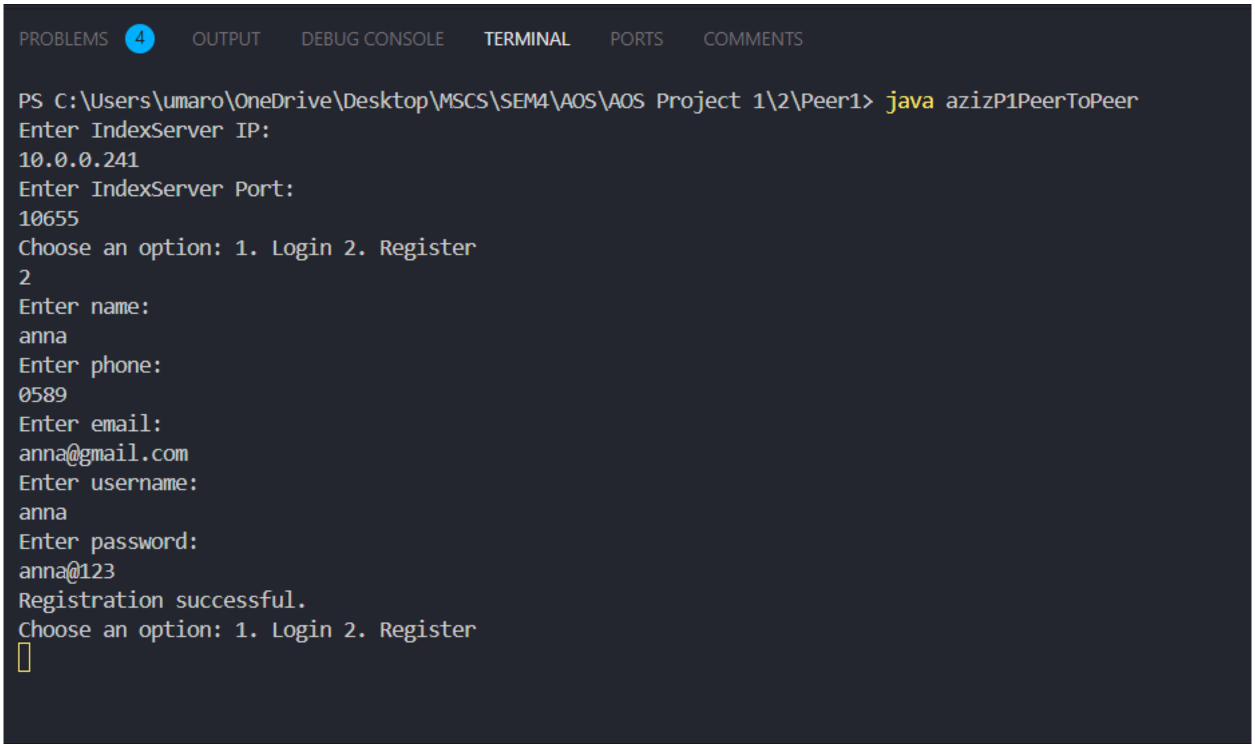
```
PS C:\Users\umaro\OneDrive\Desktop\MSCS\SEM4\AOS\AOS Project 1\2\Peer1> java Peer
Enter IndexServer IP:
10.0.0.241
Enter IndexServer Port:
10655
Choose an option: 1. Login 2. Register
1
Enter email:
umar
Enter password:
aziz
Login successful.
Assigned port: 10656
Peer server is running on port 10656
ML_RESUME__Copy_.pdf,10.0.0.241:10656
resume.pdf,10.0.0.241:10656
reddit.pdf,10.0.0.241:10656

Enter content name to search or 'exit' to quit:
FileRequestHandler createdSocket[addr=/10.0.0.7,port=49925,localport=10656]
File requested: PeerServer reddit.pdf
SpotifySetup.exe
SEARCH SpotifySetup.exe
Content found at: 10.0.0.7:10657
Enter content name to download + Ip (resume.pdf 192.134.20.20:14415) or 'return' to go back to search:
SpotifySetup.exe 10.0.0.7:10657
Downloading file from peer: 10.0.0.7:10657
File downloaded successfully.
Enter content name to search or 'exit' to quit:
```

Login And Download File



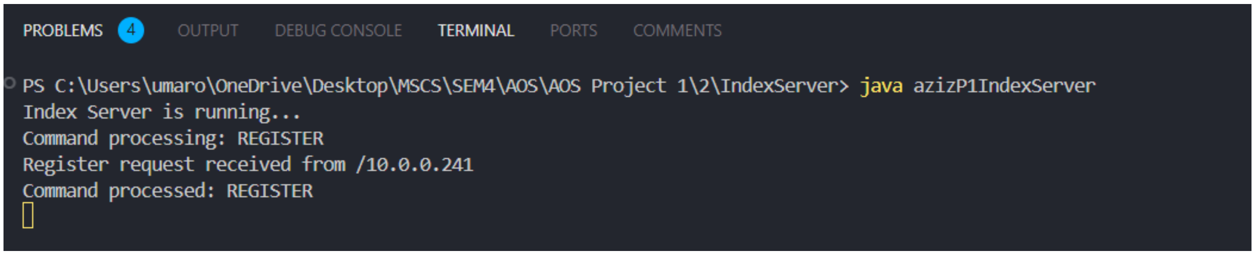
```
PS C:\Users\Umar\Desktop\2-20241008T024300Z-001\2\Peer2> java Peer
Enter IndexServer IP:
10.0.0.241
Enter IndexServer Port:
10655
Choose an option: 1. Login 2. Register
1
Enter email:
yusuf
Enter password:
baig
Login successful.
Assigned port: 10657
Peer server is running on port 10657
SpotifySetup.exe,10.0.0.7:10657

Enter content name to search or 'exit' to quit:
reddit.pdf
SEARCH reddit.pdf
Content found at: 10.0.0.241:10656
Enter content name to download + Ip (resume.pdf 192.134.20.20:14415) or 'return' to go back to search:
reddit.pdf 10.0.0.241:10656
Downloading file from peer: 10.0.0.241:10656
File downloaded successfully.
Enter content name to search or 'exit' to quit:
FileRequestHandler createdSocket[addr=/10.0.0.241,port=51100,localport=10657]
File requested: PeerServer SpotifySetup.exe
```

Login And Download from Different Computer and IP



```
PS C:\Users\umaro\OneDrive\Desktop\MSCS\SEM4\AOS\AOS Project 1\2\Peer1> java azizP1PeerToPeer
Enter IndexServer IP:
10.0.0.241
Enter IndexServer Port:
10655
Choose an option: 1. Login 2. Register
2
Enter name:
anna
Enter phone:
0589
Enter email:
anna@gmail.com
Enter username:
anna
Enter password:
anna@123
Registration successful.
Choose an option: 1. Login 2. Register
```

Registration successful on Peer Client



```
PS C:\Users\umaro\OneDrive\Desktop\MSCS\SEM4\AOS\AOS Project 1\2\IndexServer> java azizP1IndexServer
Index Server is running...
Command processing: REGISTER
Register request received from /10.0.0.241
Command processed: REGISTER
```

Registration on index server successful

CONCLUSION

This project's distributed peer-to-peer content-sharing system effectively demonstrates the benefits of multi-threading and TCP communication in creating effective, decentralized file-sharing networks. By virtue of a central IndexServer, peers can easily sign up, verify their identities, and find certain content on the network. Redundancy and latency are decreased thanks to this centralized coordination, which also improves organization and search functionality. Furthermore, the server efficiently handles several connections by allocating distinct ports to each client, allowing for scalability without sacrificing dependability. Additionally, clients can function as both content creators and consumers thanks to the design, which promotes a real peer-to-peer environment and lessens reliance on a single source.

In addition to highlighting the value of distributed systems in improving resource availability and fault tolerance, the peer-to-peer model created for this project offers practical exposure to important networking ideas like concurrent servers, socket programming, and the client-server model. It enables users to understand error handling, connection management, and the complexities of data exchange between clients and servers on a deeper level. Real-world content-sharing scenarios can benefit from the resilience and adaptability of the solution, which is enhanced by the dynamic roles that clients might play within the system. This project provides a strong basis for comprehending the difficulties and advantages of developing decentralized systems and illustrates how to overcome those difficulties with useful programming techniques.