Imama Reza, Bhavpreet Singh
1288441, 1291491

CSCI-345
Computer Networks
Fall 2023
Dr. T. Zhang
12/06/23

Instant Messenger

Project 2

Introduction

In this project, we implemented a robust chat application using Java, focusing on secure MySQL authentication for user registration and login. Leveraging the client-server architecture, we established communication channels through sockets, facilitating seamless data exchange between clients and the central server. The application prioritizes data integrity and reliability, employing the Transmission Control Protocol (TCP) to ensure ordered and error-checked delivery of messages and files. Our solution not only provides a user-friendly interface but also emphasizes secure authentication mechanisms and efficient communication protocols

Link: - https://github.com/fangedShadow/Messenger.git

Features

1)**MySQL Authentication**: Our chat application prioritizes security by implementing robust MySQL authentication. User registration and login processes are seamlessly integrated, ensuring a secure and reliable user experience.

2)**Socket Communication**: Leveraging socket programming, the application establishes efficient communication channels between clients and the central server. This allows for real-time data exchange, enabling users to send messages and files seamlessly across the network.

3)**User-Friendly Java Swing UI**: The application boasts an intuitive and visually appealing Java Swing user interface. Designed for user convenience, the interface provides a friendly environment for engaging in conversations and managing various features.

4) **Message and File Transfer:** Users can send both messages and files across the client-server-client architecture. Whether it's sharing text-based messages or transferring files of diverse formats, the application ensures a versatile and comprehensive communication platform.

Languages, tools, libraries :

- **Socket.io**
- **Mysql**
- **Java**
- **Netbeans**

Implementation

I) Authentication

Class Overview:

ServiceUser: This class manages user-related operations such as registration, login, and retrieving user information. It interacts with the MySQL database to perform these actions. The constructor initializes the class and establishes a database connection using the DatabaseConnection class.

```
public class ServiceUser {

    public ServiceUser() {
        this.con = DatabaseConnection.getInstance().getConnection();
    }
```

This method registers a new user based on the provided registration data (Model_Register). It first checks if the username already exists in the database. If the username is available, it begins a transaction to insert the user into the user and user_account tables. If successful, it commits the transaction; otherwise, it rolls back and sets an error message.

```
public Model_Message register(Model_Register data)
```

This method performs user login authentication. It takes a Model_Login object containing the username and password.Executes an SQL query to verify the credentials against the database. Returns a Model_User_Account object if the login is successful, containing user details; otherwise, it returns null.

```
public Model_User_Account login(Model_Login login) throws SQLException
```

Retrieves a list of user accounts excluding the specified user ID (exitUser) Checks the online status of each user by comparing with the list of connected clients.

```
    public List<Model_User_Account> getUser(int exitUser) throws SQLException {
```

(checkUserStatus) : Checks if a user is currently online based on the connected clients.

```
private boolean checkUserStatus(int userID) {
    List<Model_Client> clients = Service.getInstance(textArea: null).getListClient();
    for (Model_Client c : clients) {
        if (c.getUser().getUserID() == userID) {
            return true;
        }
    }
    return false;
}
```

SQL statements are defined as private constants within the class for better maintainability.

The code handles SQLExceptions and provides appropriate error messages. In case of an error during user registration, it performs a rollback to maintain data consistency.

This code essentially manages user registration, login, and retrieval of user information from the database, forming the foundation for user authentication in the chat application.

II)Server

This class manages the overall functionality of the chat server using the Socket.IO library. It handles user registration, login, message/file sending, and user status tracking.

The class follows the Singleton pattern to ensure only one instance is created. The getInstance method returns the singleton instance.

```
private Service(JTextArea textArea)
```

The constructor initializes the class, taking a JTextArea parameter for logging server activities. It creates instances of ServiceUser and ServiceFile for user-related and file-related operations. Initializes the list of connected clients

```
public void startServer()
```

Configures and starts the Socket.IO server on a specified port. Defines event listeners for user registration, login, user list retrieval, message/file sending, and disconnect events. Sends user status updates (online/offline) to clients.

Event Listener:

- Connect Listener: Logs when a client connects to the server.
- Register Event Listener: Handles user registration requests, updating the user list and notifying clients.

- Login Event Listener: Validates user login credentials, updates the user list, and notifies clients.
- List User Event Listener: Retrieves the user list for a specific user and sends it to the client.
- Send to User Event Listener: Handles message/file sending requests, including acknowledging file receptions.
- Send File Event Listener: Receives file data, sends acknowledgments, and notifies clients when a file transfer is complete.
- Get File Event Listener: Retrieves file information such as extension and size.
- Request File Event Listener: Retrieves file data based on the request.
- Disconnect Listener: Handles client disconnection events, updating the user list and notifying clients.

```
private void addClient(SocketIOClient client, Model_User_Account user)
private int removeClient(SocketIOClient client)
public List<Model_Client> getListClient()
```

Manages the list of connected clients, adding, removing, and retrieving client information.

```
private void sendToClient(Model_Send_Message data, AckRequest ar)
private void sendTempFileToClient(Model_Send_Message data, Model_Recei
```

Handles sending messages and temporary files to specific clients.

The textArea is used for logging server activities.

This class serves as the backbone of the chat server, managing user connections, events, and communications in a Socket.IO environment.

III Client

This class represents the client-side logic of the chat application. It handles interactions with the chat server, such as connecting to the server, sending/receiving messages and files, managing user status, and maintaining a message history.

The class follows the Singleton pattern to ensure only one instance is created. The getInstance method returns the singleton instance.

```
private Service()
```

The constructor initializes the class. Creates instances of ArrayList for managing file senders and receivers. Creates an instance of MessageHistoryManager for handling message history.

```
private final String SERVER_IP = "192.168.200.10";
```

Defines the IP address of the chat server. Modify this value to match the server's actual IP address.

```
private String getServerIpAddress()
```

Attempts to retrieve the local IP address. Falls back to "localhost" if unsuccessful.

```
public void startServer()
```

Configures the Socket.IO client and connects to the chat server. Defines event listeners for receiving user lists, user status updates, and incoming messages. Opens the Socket.IO connection.

Event Listeners:

- List User Event Listener: Receives updates about the list of connected users and notifies the application's UI.
- User Status Event Listener: Receives updates about the online/offline status of users and notifies the application's UI.
- Receive Message Event Listener: Receives incoming messages and notifies the application's UI.

```
public Model_File_Sender addFile(File file, Model_Send_Message
```

Adds a file sender for the specified file and message. Initiates the file sending process.

```
public void fileSendFinish(Model_File_Sender data) throws
public void fileReceiveFinish(Model_File_Receiver data) th
```

Removes the finished file sender/receiver and initiates the next one if available.

```
public void addFileReceiver(int fileID, EventFileReceiver event)
```

Adds a file receiver for the specified file ID and event. Initiates the file receiving process.

```
public void receiveMessage(Model_Receive_Message message)
```

Receives incoming messages and notifies the application's UI.

getClient: Returns the Socket.IO client instance.

getUser: Returns the current user's information.

setUser: Sets the current user's information.

error(Exception e): Prints errors to the standard error output.

This class manages the client-side functionality of the chat application, including connecting to the server, handling messages and files, managing user status, and maintaining a message history.

File Handling

This class handles file-related operations on the server-side of the chat application. It manages file sending, receiving, and storage.

```
public ServiceFIle()
```

The constructor initializes the class. Creates a connection to the database using DatabaseConnection. Initializes maps (fileReceivers and fileSenders) to keep track of file receivers and senders.

```
public Model_File addFileReceiver(String fileExtension)
```

Adds a new file to the database with the specified file extension. Returns a Model_File object representing the added file.

```
public void initFile(Model_File file, Model_Send_Message message)
public Model_File getFile(int fileID) throws SQLException
public synchronized Model_File initFile(int fileID) throws IOExcep
```

Initializes file senders and receivers. initFile creates a new Model_File_Receiver for the specified file and message. getFile retrieves file information from the database. initFile (synchronized) ensures thread-safe initialization of file senders.

```
public byte[] getFileData(long currentLength, int fileID) throws
public long getFileSize(int fileID)
```

getFileData retrieves data from the file sender based on the current length. getFileSize gets the total size of the file being sent.

```
public void receiveFile(Model_Package_Sender dataPackage)
```

Handles receiving file data packages. Writes data to the file receiver until the file is complete.

```
public Model_Send_Message closeFile(Model_Receive_Image dataImage)
```

Closes the file receiver, updates database information, and returns a message to be sent to the client.

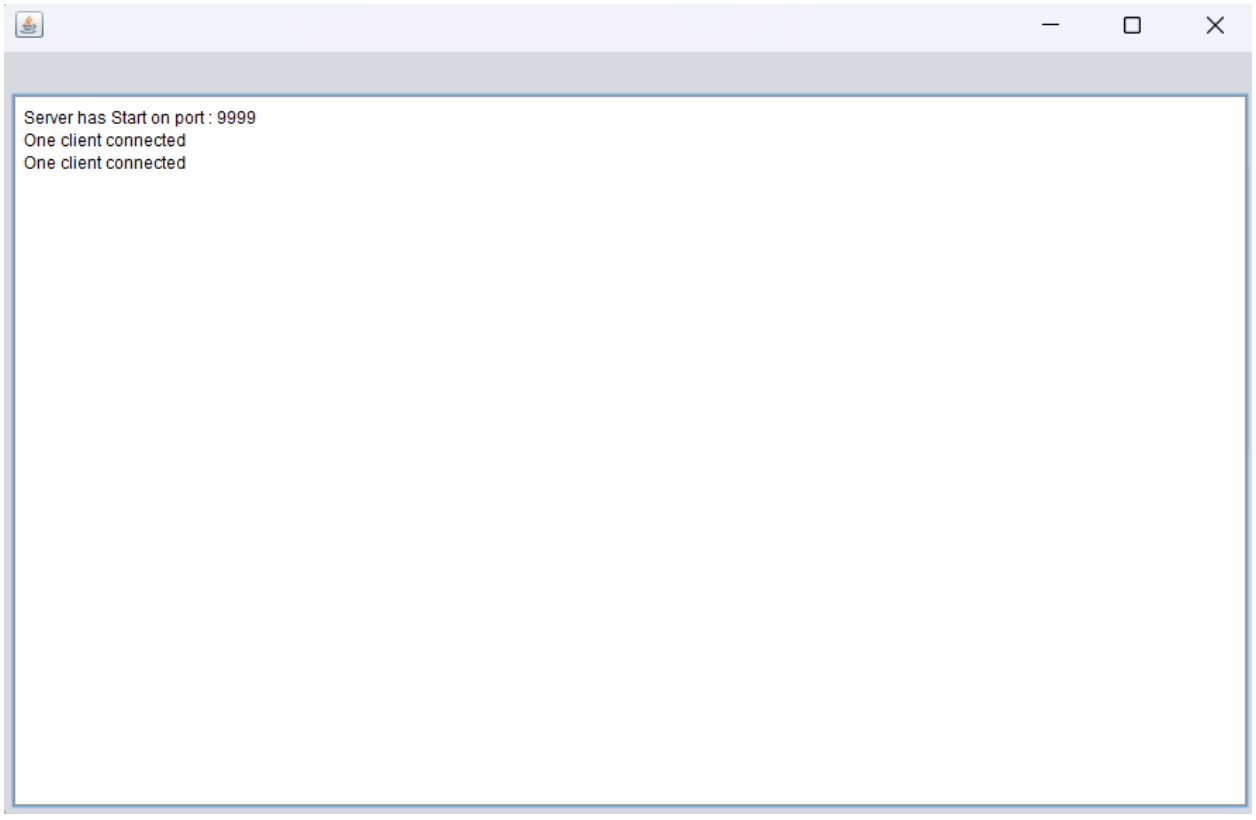
```
private final String PATH_FILE = "server_data/";
private final String INSERT = "insert into files (Fil
private final String UPDATE_BLUR_HASH_DONE = "update
private final String UPDATE_DONE = "update files set
private final String GET_FILE_EXTENSION = "select Fil
```

Defines SQL queries for file-related operations.

This class manages the file-related operations on the server, including adding files, updating file information, handling file senders and receivers.

Run

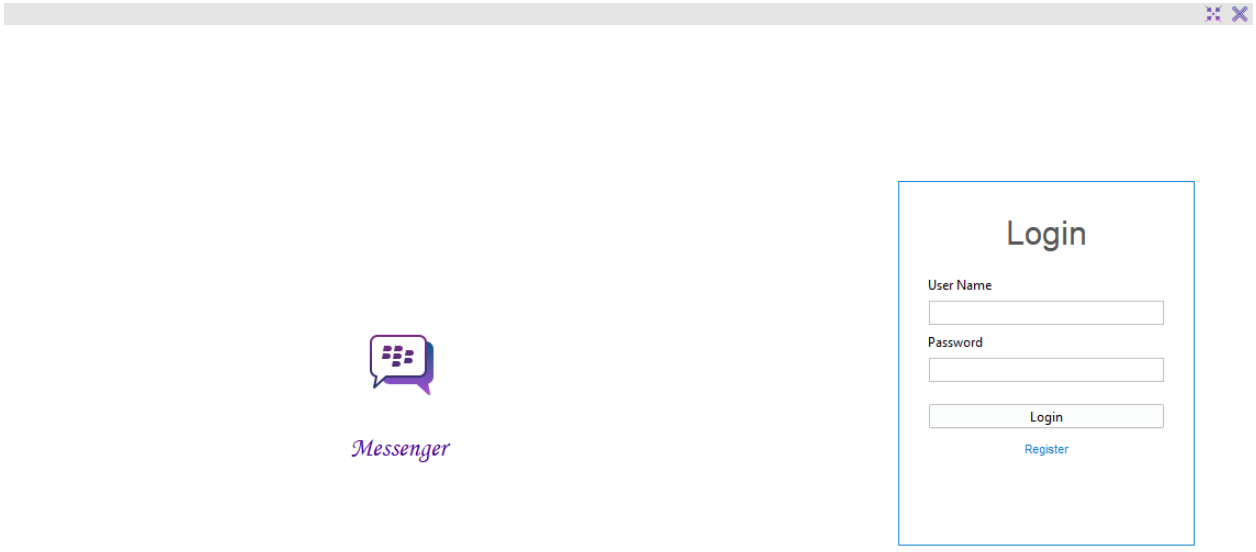
Server



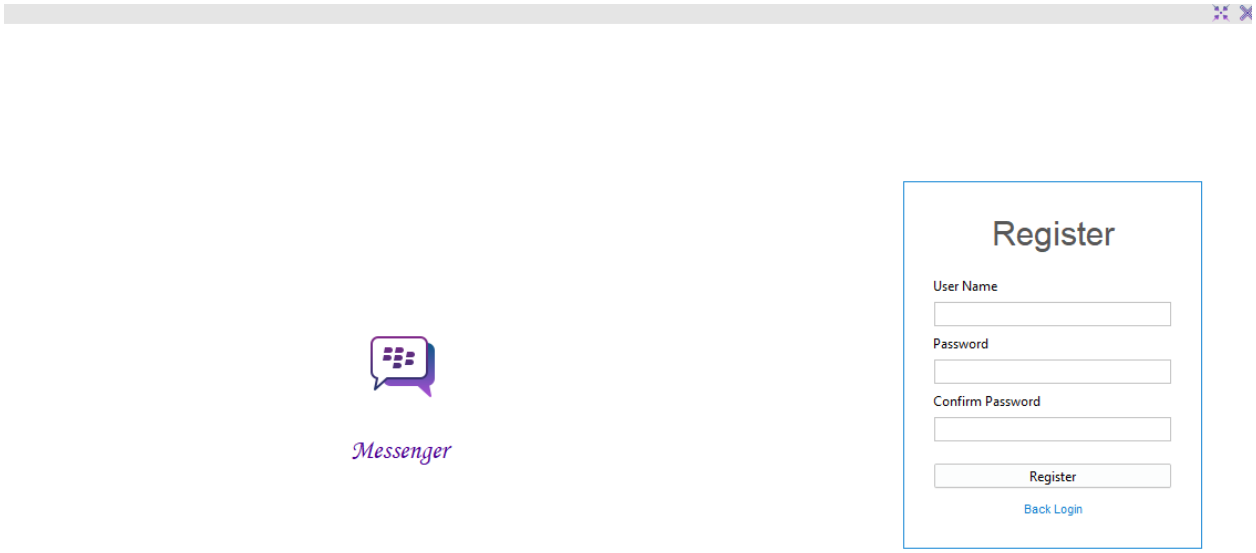
Server shows how many of the user are on the server. It also tells what port it is running on.

Authentication

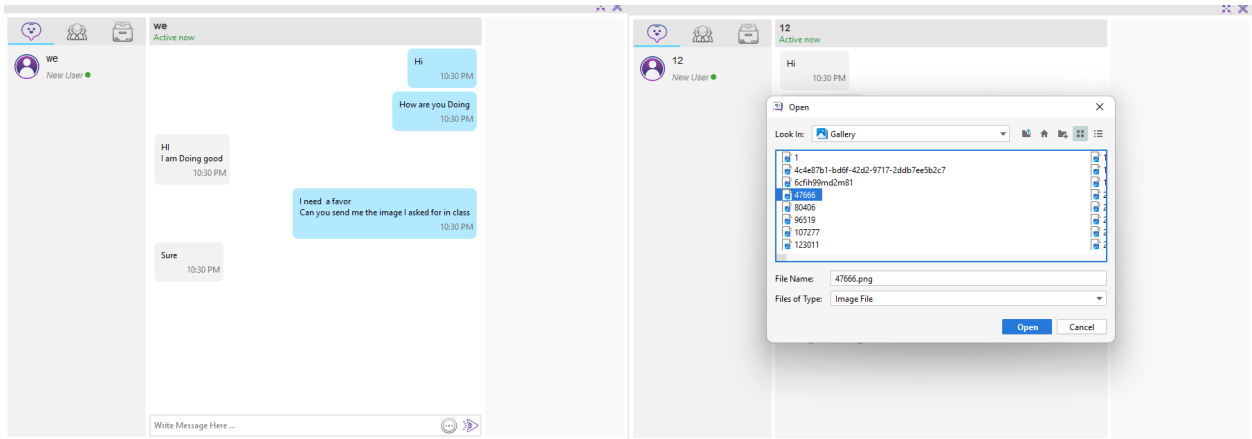
Login Tab



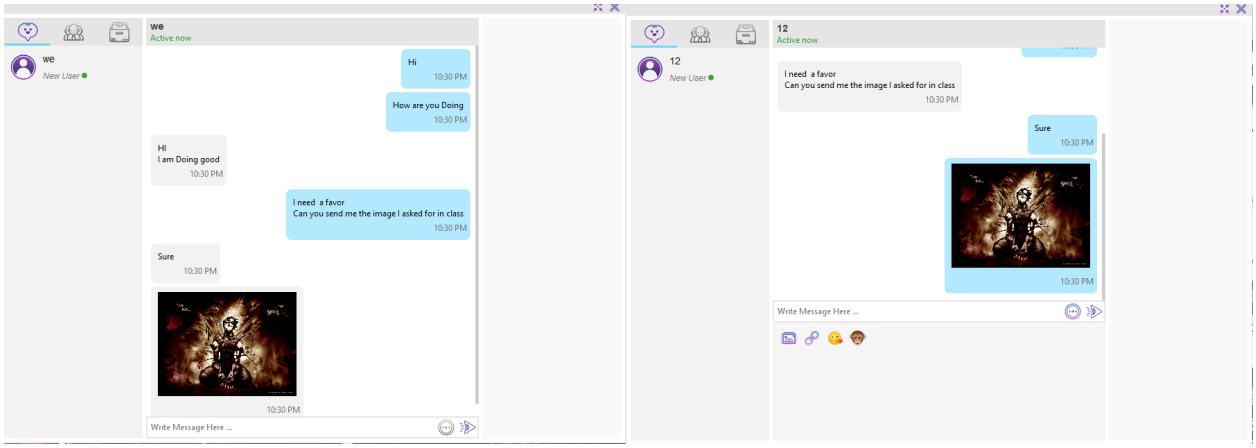
Registration Tab



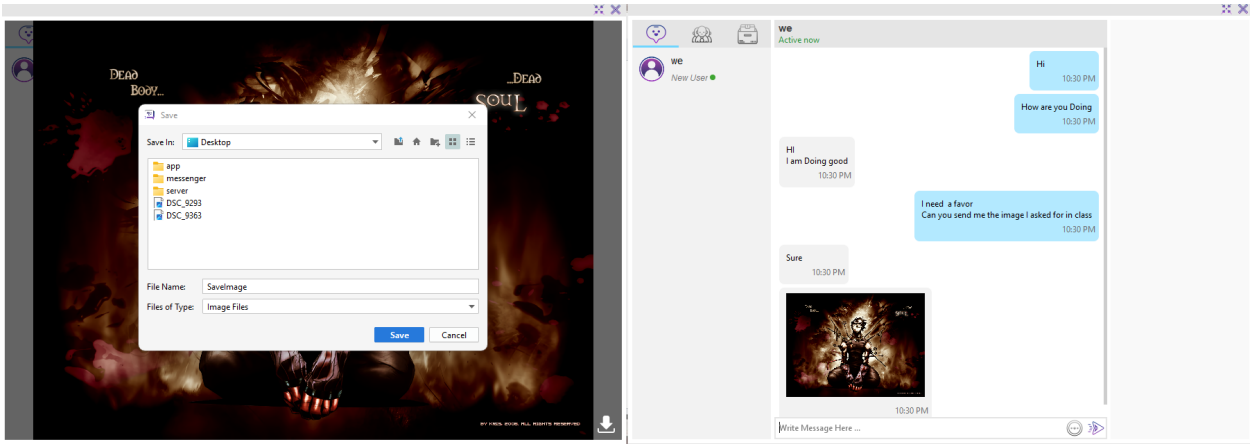
Two clients using the app



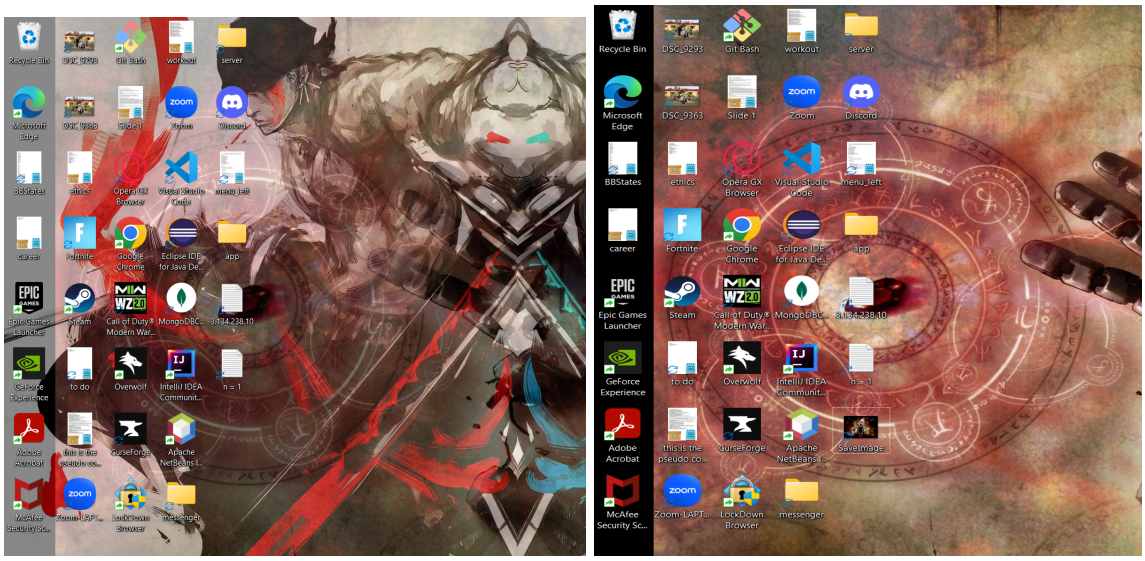
As you can see, they are having a conversation by sending and transferring the messages to each other. The application also shows the active tab. One of the user is also shown using the file transfer.



This is how it looks after the file is transferred. The image/file is stored on the mysql server in the file Table(Attached in the github).



I was able to install the file download from the server by clicking on the image and then pressing the download Icon/label. As you can see, the save window pops up(using the JfileChooser)



The left is the before and right is the after of the file save.

Limitation

The chat application, while providing essential communication features, has certain limitations. Firstly, the application relies on a centralized server architecture, which could introduce scalability issues as the user base grows substantially. One notable restriction is the inability for users to view their saved message history within the application. Additionally, the application lacks end-to-end encryption for messages and files, potentially compromising user data security during transmission. The absence of robust error handling mechanisms might lead to unexpected behavior in case of network interruptions or server downtimes. Furthermore, the application's

user interface, although functional, may not be optimized for a diverse range of devices and screen sizes.

Conclusion

In conclusion, the development and implementation of our chat application have resulted in a feature-rich platform that facilitates seamless communication and file sharing between users. Leveraging MySQL authentication, socket communication, and a user-friendly Java Swing UI, the application ensures secure and efficient interactions. The integration of TCP and UDP protocols enables reliable data transmission across the client-server-client architecture. While the app excels in providing real-time communication, it is essential to acknowledge certain limitations, such as the absence of a built-in feature to view saved message history. Future iterations could focus on addressing these limitations and expanding the application's capabilities.