

COMP3331 Computer Networks and Applications

Assignment Report

Zeneng Zhao
Z5369232

Introduction

The application is implemented with Java (OpenJDK 11.0.21). The application is a client-server model consisting of one server and multiple clients. The text messages are sent via TCP while the videos are sent via UDP client to client.

Program Design

Application files:

- Audience.java
encapsulate the UDP port and user address.
- Authenticator.java
Helps to authenticate the user by reading credentials.txt.
- Client.java
Client application, communicate with the server.
- Command.java
Helps to process the user's command (e.g.: /activeuser, /msgto, /logout ... etc.)
- Group.java
A group object helps to manage the group mechanism of the application.
- GroupLog.java
Helps to manage all the groups created in the application.
- GroupMessageLog.java
Extends the MessageLog.java class helps to log the messages in a group.
- Message.java
Helps to process the message, contains the sender's name (on group chat) or receiver name (on private message), also a timestamp.
- MessageLog.java
Helps to log the message sent in server in file.
- Messenger.java
Helps to send and receive messages between server and client. Encapsulate the DataInputStream and DataOutputStream and provides more features.
- Server.java
Server application, communicate with multiple clients.
- UserLog.java
Helps to manage all users' activities and log them in userlog.txt.
- UserRecord.java
Helps to manage the user's basic information.

Figure 1 Application Files and Description

For a better view of the design, here is the UML diagram:

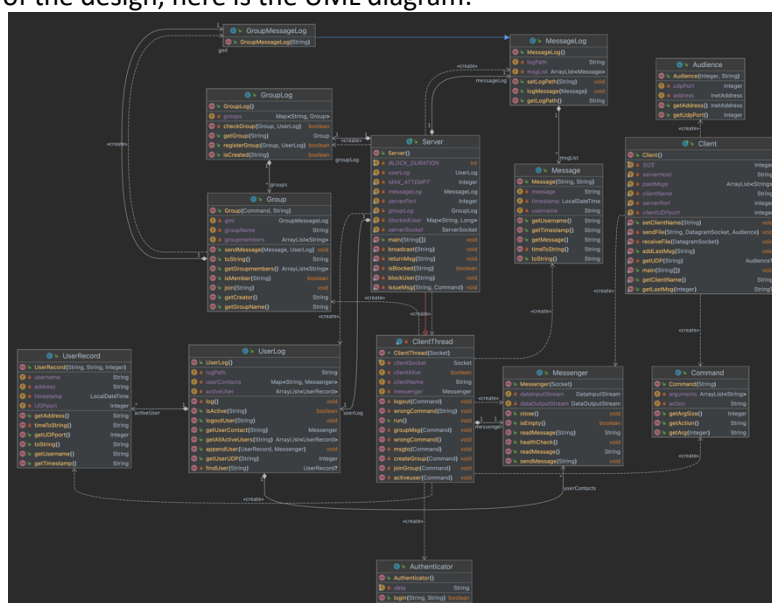


Figure 2 Application UML diagram

The application requires a **credential.txt** file to store all the users and their password. The application on run will generate extra files, such as group message log file. The application will automatically update content of the log files (e.g.: userlog.txt, messagelog.txt and group message log files).

Application Layer Message Format

Messages are conveyed as Command objects; during communication, the client sends plain text, and the server reads and parses it into a Command object.

The Command object comprises an action and arguments. The action represents one of the provided commands. The server takes an action and then executes the corresponding command by interpreting the subsequent arguments.

In more detail, when the client sends a message to execute a command, the message is split into words using the space character as the delimiter. The first word denotes the action, and the subsequent words are the arguments. If the action is not one of the provided commands, the server promptly informs the client that the command is invalid.

System Functionality

Terminal 1: Server



```
java Server 8000 3
```

Figure 3 Server starts command.

On running, server listen to every new connect and create a thread for it.

Terminal 2: Client



```
java Client localhost 8000 8964
```

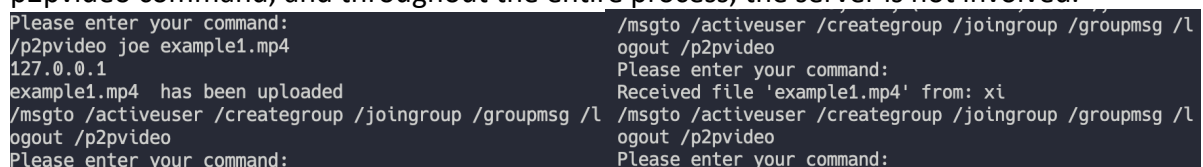
Figure 4 Client starts command.

On running, client establish a TCP connect to the server with given address. Client creates two Threads:

- One thread to display all incoming response from the server.
- One thread waiting for any incoming UDP files.

P2P:

The client-side implementation of the p2pvideo command assumes that the client has previously executed the /activeuser command. Upon execution, the client's UDP socket is utilized to send the source file in chunks of packets. The first packet contains metadata with information about the sender and filename, while the end packet is an empty packet signalling the completion of the transfer. Importantly, the server is not aware of the p2pvideo command, and throughout the entire process, the server is not involved.



```
Please enter your command: /msgto /activeuser /creategroup /joingroup /groupmsg /l
/p2pvideo joe example1.mp4 ogout /p2pvideo
127.0.0.1 Please enter your command:
example1.mp4 has been uploaded Received file 'example1.mp4' from: xi
/msgto /activeuser /creategroup /joingroup /groupmsg /l /msgto /activeuser /creategroup /joingroup /groupmsg /l
ogout /p2pvideo ogout /p2pvideo
Please enter your command: Please enter your command:
```

Figure 5 p2pvideo, xi send exampl1.mp4 to joe.

Design Trade-offs

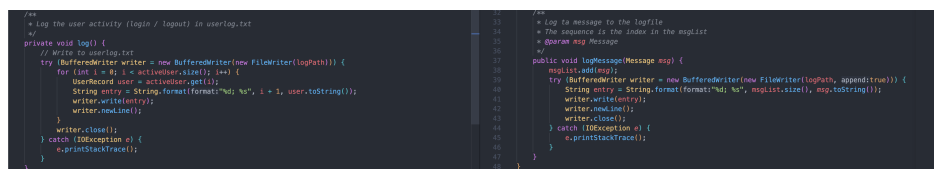
In the client-side implementation of the p2pvideo command, knowledge of the UDP port number relies on executing the activeuser command to obtain this information from the server. The past message mechanism is utilized to extract the UDP port number from the returned message. However, this approach might impact client performance during extended sessions of communication with the server, particularly in the presence of a large message history.

Building upon the past message mechanism, for sending files via UDP, knowledge of the receiver's address is essential. The Audience.java class is employed to encapsulate both the receiver's address and UDP port, extracting this information from the return message obtained through the activeuser command. This functionality is implemented in the getUDP method within the Client.java class.

Possible Improvements and Extensions

To enhance the amount of information conveyed during communication, messages can be encapsulated in JSON format. This approach streamlines the message processing stage on both ends, allowing for a more efficient and structured exchange of data. Additionally, it enables the encapsulation of data that is not intended for direct user visibility.

Certain codes and classes can benefit from refactoring based on object-oriented programming principles. Specifically, there is an opportunity to consolidate classes where features overlap. An opportunity arises to consolidate shared features, exemplified by combining functionality from UserLog.java and MessageLog.java into a new class, say Log.java. Both UserLog.java and MessageLog.java can then extend this common class, minimizing redundancy and promoting a more modular and organized code structure.



```
1 // Log the user activity (login / logout) in userlog.txt
2
3 private void log() {
4     // Write to userlog.txt
5     try {BufferedWriter writer = new BufferedWriter(new FileWriter(logPath)); {
6         for (int i = 0; i < activeuser.size(); i++) {
7             UserRecord user = activeuser.get(i);
8             String entry = String.format("%d: %s", i + 1, user.toString());
9             writer.write(entry);
10            writer.newLine();
11        }
12        writer.close();
13    } catch (IOException e) {
14        e.printStackTrace();
15    }
16 }
17
18 // Log a message to the logfile
19 // The message is the index in the msglist
20 // @param msg Message
21
22 public void logMessage(Message msg) {
23     msgList.add(msg);
24     try {BufferedWriter writer = new BufferedWriter(new FileWriter(logPath, append=true)); {
25         String entry = String.format("%d: %s", msgList.size(), msg.toString());
26         writer.write(entry);
27         writer.newLine();
28         writer.close();
29     } catch (IOException e) {
30         e.printStackTrace();
31     }
32 }
```

Figure 6 Code redundancy in UserLog.java and MessageLog.java

Limitations

The absence of unit tests in the code makes it challenging to guarantee the accuracy of individual components. All testing is conducted manually by human intervention.

Reference

Sever.java and Client.java is developed based on the Multi-threaded Code given in course website.