# Assignment Report

Tsun Wang Sau

November 2021

## Program Design

The programs are written in Java. All the files that start with Client are used for the client application and all the files that start with Server are used for the server application.

I use state pattern for the login proccess as the client and the server handles the inputs differently to the state after login. I also create a private permission state in client application, which only allows the input "y" or "n".

For the rest of the time (ClientNormalState and ServerNormalState), both the applications are basically stateless in which they treat every message independently without the need to know the state of the other application. Not tracking the states makes the applications simpler.

For example, the client application does not remember that it has broadcasted or sent a message. However, the client app can still print the fail message if the message or the broadcast is failed as it is told explicitly in the message("failmessage\n" and "failbroadcast\n") sent by the server.

This also applies for the startprivate function. The client and the server applications do not remember that the user has tried to start a private connection with others but when thay receive the "privateaccept (username) (ip) (port)\n" or "privatedecline (username) (failreason)\n" messagae they know whether the private connection is accepted by the others.

## Data Structure Design

The server stores quite a number of data structures. Most of the data structures are hashmaps. Since the server is a multi-threaded program, ConcurrentHashMaps are used.

The server stores a map called blockedLogins, which the key of it is the username of the blocked users who fail to login three times and the value is the time they are allowed to login again.

The server also stores a map called unreadMessages for unread messages. The key is the username and the value is the unread messages for the users. Every key in the unreadMessages is an username which haven't logged in successfully.

The server moves all the credentials from the credentials.txt to the main memory in a map when it starts. The key of the map is the username and the value is the password. The server will check this creadential map whenever a user login.

The server also stores a blacklist map which is used for block and unblock users. The key of the blacklist is the username and the value is a list of the user who has blocked the user.

There is also a history map which stores the last login time for each user which is used for the whoelsesince command. The connections map stores the

thread for each online user which allows the user to send message to the others user.

The client only stores the p2pConnections map which store the private connections. The key is the name of the peer user which they have created private connections with, and the value is the p2pThread data type which handles reading from and writing to the other user.

## Message Format

The messages format is quite straight forward. The message is delimetered by spaces and '\n'(the newline character) denotes the end of the message. This works as usernames do not contain space and messages do not contais the newline character. The first word of the message is the function type and the rest are the parameters. Using newline for each messages allows the both client and server relies on readline to read the message and using spaces as a delimeter allows them to split the string really easily with split(' ') method.

## How Does The System Works

The server simply create a new thread whenever there is a new connection. The client create a thread for handling messages recieved form the server and the main thread continues to handle inputs from the command line. The client will create a new thread for each private connection.

For the timeout mechanism, the server simply use the setSoTimeout for each socket. When the socket does not receive any command from the client, it sents a "timeout\n" to the client app and the clinet app will exit the program. Since every command that will go to the server must be valid as the client has already filtered all the invalid command, the server does not need to think about setting timeout for invalid command.

For the blocking mechanism, the server simply add the username to the blockedLogins map with the current tiem + blocking duration for the value and send back a "block\n" message after three login failures. When the user tries to relogin, the server checks the blockedLogins first, if the name is in the map and the value is larger than the current time, it sends the "block\n", otherwise it sends "welcome\n".

After the succesful login, the server sends the unread messages of that users (the format is "unread (username) (message)\n" for each unread message) and broadcast its precense to the rest of the users (the format is "precense log in/out\n").

For the private message, when the user send a "startprivate (username) \n" to the server, the server sends a "askprivatepermission (username)\n" to the other user. Then, the user will be in the PrivatePermissionState. When the user types 'y', the client app creates a server socket with a random selected port, and send the "privateaccept (username) (port)\n" message back to the server, then the server can forward the "privateaccept (username) (ip) (port)"

to the client. When the client receives it, it can then create a socket with the given ip and port for the connection. If the user rejects, the client sends the "privatedeclien (username)\n" to the server. Then, the server will tell the other client. When the users sends a private message, another message will also be sent to the server, so that the server can reset teh timeout.

For stopping the private connection, the user will send the "stopprivate\n" to the other user, and both of them will close the socket. One thing to notice is about closing the p2pThread. An IOExeption will be raised when the thread is reading from a closed socket. When the p2pThread catches that exception, it will break the while loop and the thread will then be closed.

## Design Trade-offs

For the message, it is human readable which makes it easier to debug but it is not space efficient and the time to transmit will also be longer. Also, for the unread messages, it can be more effiecient to send all the messages with just one "unread" instead of sending an "unread" for each message. One possible solution is to contains the number of unread messages in the message instead. However, it is slightly harder to implement.

The other design trade-off is about choosing multithreading or non-blocking io with select for the client application. The benefits with select are that it reduces the memory use for threads and the proccess, can make use of the time while reading from the socket and does not need to take care about concurrency issues. The benefits of multithreading is that it is a bit easier as the server is also multi-threaded and it also allows multiple users accessing data concurrently.

## Circumstances That The Program May not work

If the program is used in the real world, the starting private connection process does not work if the clients use NAT, as the port in the porgram is not the actual port used in the NAT. Somthing like ICE for WebrRTC is needed.

The program also won't work if multiple users asking the same user for private connection at the same time.

If in the future, the messges with the newline character is allowed, the programs will also not work.

## References

To get the ip of the client, I use the code from https://www.baeldung.com/java-client-get-ip-address in line 142-143. I learnt the idea from https://stackoverflow.com/a/3421617 to close the p2pThread by closing the socket to raise an IOExcpetion while the thread is blocking in the readline method, then we can catch the exception to close the thread.