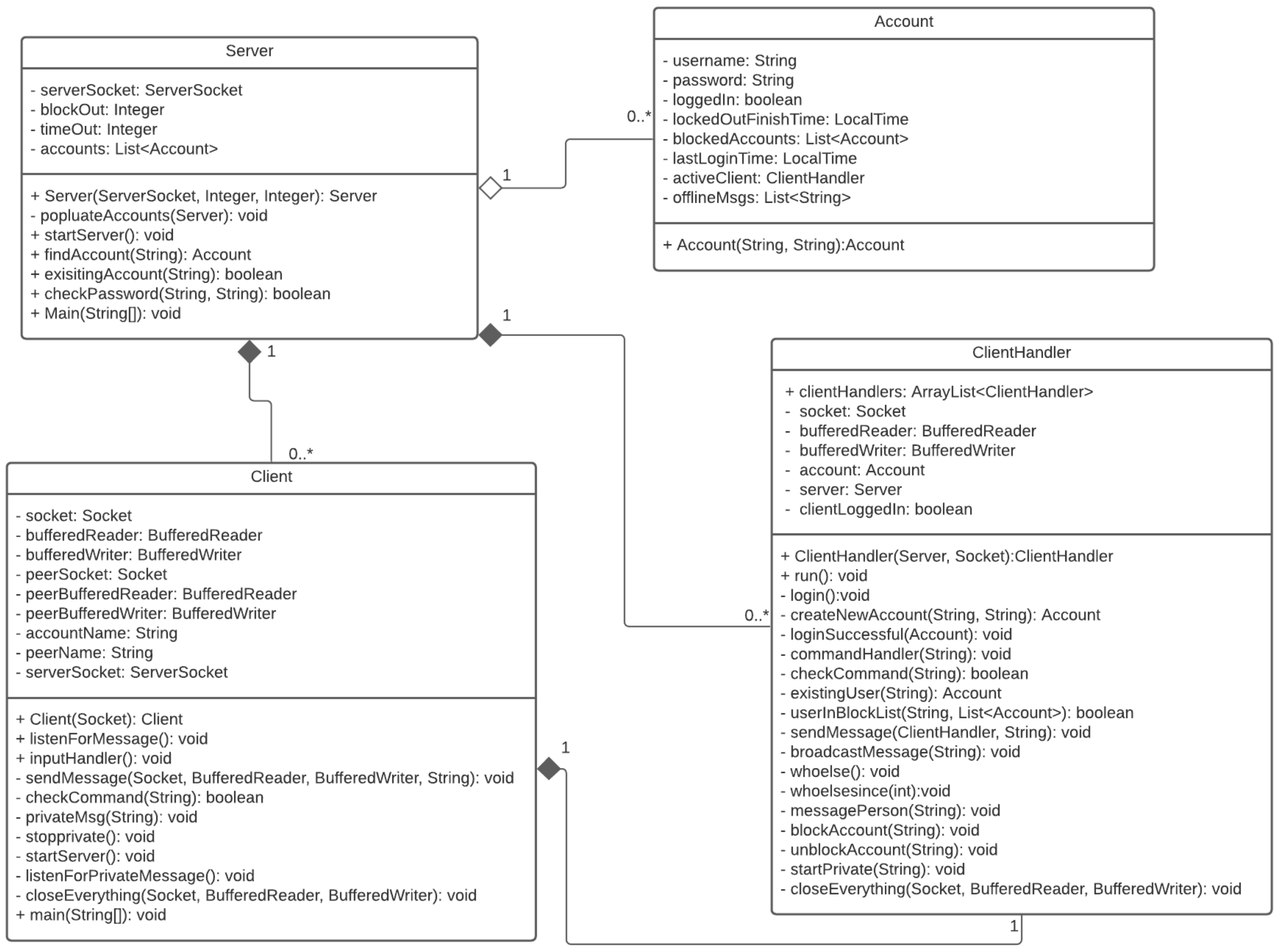
COMP 3331 Assignment Program Design Report

(By Marko Wong z530931)

Program design


Server
- serverSocket: ServerSocket
- blockOut: Integer
- timeOut: Integer
- accounts: List<Account>
+ Server(ServerSocket, Integer, Integer): Server
- popluateAccounts(Server): void
+ startServer(): void
+ findAccount(String): Account
+ exisitingAccount(String): boolean
+ checkPassword(String, String): boolean
+ Main(String[]): void
Account
- username: String
- password: String
- loggedIn: boolean
- lockedOutFinishTime: LocalTime
- blockedAccounts: List<Account>
- lastLoginTime: LocalTime
- activeClient: ClientHandler
- offlineMsgs: List<String>
+ Account(String, String):Account
ClientHandler
+ clientHandlers: ArrayList<ClientHandler>
- socket: Socket
- bufferedReader: BufferedReader
- bufferedWriter: BufferedWriter
- account: Account
- server: Server
- clientLoggedIn: boolean
+ ClientHandler(Server, Socket):ClientHandler
+ run(): void
- login():void
- createNewAccount(String, String): Account
- loginSuccessful(Account): void
- commandHandler(String): void
- checkCommand(String): boolean
- existingUser(String): Account
- userInBlockList(String, List<Account>): boolean
- sendMessage(ClientHandler, String): void
- broadcastMessage(String): void
- whoelse(): void
- whoelsesince(int):void
- messagePerson(String): void
- blockAccount(String): void
- unblockAccount(String): void
- startPrivate(String): void
- closeEverything(Socket, BufferedReader, BufferedWriter): void
Client
- socket: Socket
- bufferedReader: BufferedReader
- bufferedWriter: BufferedWriter
- peerSocket: Socket
- peerBufferedReader: BufferedReader
- peerBufferedWriter: BufferedWriter
- accountName: String
- peerName: String
- serverSocket: ServerSocket
+ Client(Socket): Client
+ listenForMessage(): void
+ inputHandler(): void
- sendMessage(Socket, BufferedReader, BufferedWriter, String): void
- checkCommand(String): boolean
- privateMsg(String): void
- stopprivate(): void
- startServer(): void
- listenForPrivateMessage(): void
- closeEverything(Socket, BufferedReader, BufferedWriter): void
+ main(String[]): void
0..*
1
1
0..*
1
0..*
1
1

Link to diagram: https://lucid.app/documents/view/753ae4e7-0b68-4008-844d-4728f8e6c807

The program consists of 4 classes.

1. Server: Responsible for running the Server and accepting any new clients
2. Client: Responsible for running the Client and sending command to Server
3. ClientHandler: Part of Server, responsible for handling commands from Client
4. Account: Responsible for storing data about user accounts

Data structure design

All the data about users are stored in the Account class and the data about active Clients are stored in the ClientHandler class in the arraylist “clientHandlers”.

Application layer message format
There is no strict message format, however when users send messages their message begins with their account name whereas Server messages are just plain messages with no format. This is enough to stop users trying to prose as the Server and altering the program as all their messages are distinguished by their messages beginning with their username.

How does the system work?
An overview of the system is the server starts and waits for clients to connect via a TCP connection and then the server will control the flow of messages exchanged between clients.

A more in depth view is as follows: firstly, when the Server is started it will get three inputs from the command line arguments: the server port, block out time and timeout duration. Using the server port it will create a welcoming socket and populate the server accounts with all the entries in a file called “credentials.txt”. After all the setup it will start the server and wait for clients to connect via a TCP connection. When a client connects to the server, the server will accept this connection in its’ welcoming port and will create a new thread with it’s own socket to communicate with this client in the ClientHandler class. In the ClientHandler class it will detect this client is not logged in yet and will send the appropriate prompt for the client to login or create a new account. In the event the client fails to enter the correct password 3 times, then the ClientHander will lockout the account the client was attempting to login into for the Block out duration. When the Client has successfully logged in the timer for the time out will begin. Everytime the client sends a command to the ClientHander, the ClientHander will check if the command is valid. If the command is valid the command is processed accordingly. Otherwise the time out timer will continue until there is a valid command sent by the client. In the event the client didn’t send a valid command before the time out timer ran out the client program will be sent a time out message and the client program will terminate.

For the Peer to Peer messaging the logic is as follows. Given there is two client logged in, when a client-A send the command “startprivate client-B” to the server, the server will send a request message to client-B and wait for a response from client-B. If client-B accepts this request, Client-B will send a message accepting this request to the server to which the server will then send a message to Client-B to start up it’s Socket to an upcoming connection. The server will also send a message to Client-A containing Client-B socket. Client-A using the received socket will connect to the socket via a TCP connection. Upon a successful connection both ends will remember the other client output and input stream from the socket. Both clients will also start up another thread to continuously listen for private messages. From here both clients can send messages without the need of going through the server. However each valid private message will send a valid command message server to maintain the time out timer. When a client in the private connection wants to disconnect or when either client logs out, both clients will try to send a message to the other client about the closure of the connection and then close the socket and all its streams.

Design trade-offs considered and made

The main Design trade-off made was deciding to remove the welcoming port for peer to peer or private messaging. The reason behind this was to simplify the program as the specification said there will only be one private messaging session for each client. Hence it will be a lot simpler to remove the welcoming port and just use the port as the peer to peer socket.

Limitation of Program

- When the client disconnects mid way into logging in the entries are made null fields.
    - Since the specification stated “we will ONLY assume that users will exit by explicitly using the logout command” meaning the client will never disconnect mid way when logging in. Hence this limitation should never happen.
- Only supports 1 peer to peer connection at a time.
    - Since the specification stated “we will initiate at most one p2p messaging session from each client at any given time” which led to this limitation.
- When waiting for the other peer to respond to the private messaging request, the peer who made the request cannot issue other commands until the other peer responds.
    - Since the specification stated “Setting up a TCP connection between the two will require certain information about B to be communicated to A via the server. ...We will leave the design to you.” Since this is part of the TCP connection process I decided on this limitation.

Acknowledgement

The basic structure of the TCP Server and Client is based on the video by Wittcode instead of the provided starter code.