COMP3331 Assignment Report
by z5205292

Python version: Both my client and server code use Python 3.7 as on the CSE machines.

Program design: As per the spec, the program is designed in two files, being a client and a server. Whilst there are countless ways in which this could be designed to work, I thought it made the most sense to have the server deal with as much as possible, whilst the client should mainly be responsible for sending whatever the user inputs to the server, and printing whatever the server responds with.

To use the program, the server must first be started, followed by the client. Once this is done, the client should ask the user to input their username, and once done this should be sent to be verified by the server. If successful, a password prompt will be seen by the client. If not successful, an appropriate error message will be seen by the client. This pattern is used throughout the entirety of the program, with the client not being responsible for anything else other than receiving messages from the server, and inputting commands for the server to interpret and do something with.

When the server receives a message from a client, its job is to interpret this data and create an appropriate response for the client. It was an intentional design choice such that the client is not involved in any of this process, rather simply waits for the response for the server and displays it to the user.

In terms of the server, it is split up into some main method to illustrate the main stages of connecting to the server. Once the welcoming socket accepts a new client connection, it is sent to the ‘validateUsername’ method. As the name suggests, this method is responsible for asking the client for their username, waiting for a response, and then determining if this is a valid username. If successful, the ‘validatePassword’ method is called, which works in a similar way. If a valid password for this username is given, the ‘home’ method is executed.

Since all that can be done before logging in is entering your username followed by your password, it made sense to split these stages of the login process into separate methods, as it not only makes the code cleaner but also demonstrates how it is a linear progression at this stage.

The home method is designed as an infinite loop, which simply waits for a response from the client, determines what type of command they executed, does some error checking for this command and then sends a response message to the client. This was done as a user can use any combination of commands in any order whenever they want, so having a linear sequence of progression doesn’t make sense after logging in.

When adding the private messaging functionality, the design of the program had to be altered slightly, giving the client a listening socket for any new incoming client connections, alongside ensuring the client was given some control over what to do with messages it receives. In doing so, the importance of using separate client threads for sending and receiving messages is highlighted, as without it a client would not be able to receive messages from the server and another client simultaneously. Similarly, the server must use threads so that it can receive messages from multiple clients simultaneously.

Application layer message format: In order to keep things as simple as possible, all messages sent from the client to the server were sent unmodified. In other words, whatever the user typed in their terminal was sent to the server.

Once the server received a message from a client, it would then use a combination of string comparison and regular expressions to determine what the user wanted to do. Once matching a command, the server could extract any useful data using matching groups in regex, and once done any error-checking could send back the message to be displayed in the client's terminal. Since the returned message is already in the correct format to be read, the client can simply print this message as is.

I believe using this message format made it a lot easier to implement all of the different error messages and different edge cases, as the client can simply print whatever is sent to it rather than having to format the server's message before doing so.

However, some commands involved sending additional data within the messages which would remain unseen by the user. For example, since the client program needs to know its own username for later use, the server may send a message in the form password: {} msg: {}, where the first field contains the username and the second field contains the message to display in the client's terminal. I used this format for all of these cases, as it allowed me to first extract the necessary data, and once done to simply print whatever remained after the 'msg:' part of the message. I used this format for the 'startprivate' and 'stopprivate' commands as well.

How my system works: Starting with the server, after setting up some initial variables, the main method is called, which creates a welcoming socket and then enters an infinite loop. In this loop, it waits for any client's connecting to this socket, and when one does it creates a new thread to handle this. This thread starts with the 'validateUsername' method, which asks the user for their username and waits for a response. Upon receiving a response, it checks this is valid, and either calls this method again if incorrect or calls the 'validatePassword' method.

In this method, for a maximum of three times in a row, the server asks the client for their password. If invalid, the loop continues, and it the limit is reached, the connection to the client socket is closed and the client terminal will hence close. However, if correct, the 'home' method is called. This begins by settings up some initial variables that are needed for all new users who log in. Once done, an infinite loop begins, which simply waits until a message is received from the client.

Once a message is received from the client, a list of if statements are executed to determine what command the user entered. If none match, it will reach the end of the method and simply let the client know this is an invalid command, followed by restarting the loop. If an exception occurs, which could happen because the user logged out or a timeout, an except clause is used to close the connection to the client socket and handle anything that needs to be done when a user exits. Otherwise, if one of the if statements finds a match for the command, then depending on the command it will get some information and send appropriate messages to the desired client(s).

The client starts in a similar way, assigning any variables and then entering the main method. It first sets up a socket to connect to the server, and for this socket creates both a sending and receiving thread. Additionally, another thread exists which creates a welcoming socket for any private client connections, and then sits in an infinite loop similar to how the server welcoming socket works. Upon receiving a new client connection, a thread to receive messages is made and it continues listening.

The sending thread for the client simply enters an infinite loop, waits for the user's input and then does either one of two things. If it is a private message, it sends it to the intended client (given it is a valid private message). Otherwise, it sends this message to the server. The receiving thread also enters an infinite loop, and waits to receive a message from the other socket. It checks for some special case messages, such as starting and stopping a private connection, but otherwise it simply prints this message to display in the client terminal. If an exception is raised for some reason, any sockets connected to either the server or other clients are closed and the program ends.

Design trade-offs: To keep the program simple, one trade-off I made was abstracting most of the functionality to the server side, resulting in the client having much less information about the messaging application in general. As a result of this, some tasks such as private messaging another client without using the server became a lot more challenging and required added steps to be added. This was because the client didn't know their own username, and I had to add extra complexity to the client side to account for this.

Additionally, another trade off I considered was using a single threaded client to keep it as simple as possible, which would have resulted in the client's ability to both send and receive messages simultaneously much more difficult. However, I ended up going against this decision, as the added simplicity to the code was not worth the various issues that would arise from trying to implement this successfully.

Possible improvements/extensions: One possible extension that could be made to the program is having messaging groups containing various users. To realise this, first a command to create a group would need to be implemented. Next, a user in a group (at this point this is simply the owner) should have the ability to add a user to a group through an add user command. Note they shouldn't be able to add users who have blocked them. Once there are users in a group, messaging the group would work in a very similar way to how broadcasting works, except that rather than messaging all online users, it only messages users who are in the group. Additional commands that could improve this would be changing the name of the group, adding admins, leaving the group and deleting the group.

A possible improvement to be made to my program could be handling more edge cases that aren't included in the spec. Whilst a lot of these are very unlikely to be seen by most users, having the program crash or behave unexpectedly is not ideal. Another improvement would be that a user should not be able to private message another user if they become blocked, even if an existing private connection already exists. Whilst the spec says to ignore this, I believe adding this to my program would definitely improve it, as if a user blocks someone they definitely don't want to be messaged privately by them.