COMP3331 Assignment Report
FONG CHING HIN STEPHEN z5191673

Program design

There are 3 files for the program: server.py, client.py and helper.py.
server.py consists of codes handling the interactions with the client, it's mainly divided into two parts: login_process and main_process
client.py consists of codes handling the interactions with the server, it's also divided into two parts: login_process and command_process
login_process() in server.py interacts with login_process() in client.py while main_process() in server.py interacts with command_process() in client.py
helper.py consists of some helper functions and a variable that can be used by either server.py or client.py to reduce code redundancy.
Since the client plays an important part in the application and there's a lot of information related to it, a class User() is created and functions inside can be used to extract information of the user or send/receive messages from it.

Application layer message format

All application layer message starts with "**WhatsApp**", here's a list of application layer messages and their corresponding description.

WhatsApp <username> logout
This message is sent from the server to the client to tell the client that it has been automatically logged out due to timeout, the client who receives this message will close any related sockets and terminates the program.
WhatsApp <usernameA> startprivate <userB_IP_address> <userB_private_accepting_port> <usernameB>
This message is sent from the server to the initiating side (<usernameA>) of the private messaging session, i.e. the client who initiate a private messaging session.
<usernameA> will then use the IP address and port number attached in the message to connect another client's connection socket and open a new thread to receive the message coming from <usernameB>.
WhatsApp <usernameA> allowprivate <usernameB>
This message is sent from the server to the accepting side (<usernameA>) of the private messaging session, i.e. the client who accepts another client's private messaging invitation.
<usernameA> will then open a new thread to accept a new connection from <usernameB> using the privateAcceptSocket and receiving the message coming from <usernameB>
WhatsApp stopprivate (1) with <username>
This message is sent from the server to the client (who typed the stopprivate command) to ask the client to close its private messaging session with <username> and deleted all related information about the session.
WhatsApp stopprivate (2) with <username>
This is basically the same as the above command, the only difference is that message is sent to the passive side of the connection, i.e. the client who didn't type the stopprivate

command. The message causes a prompt telling the private connection has been discontinued to be shown on the client

WhatsApp sent private command

This message is sent from the client application to the server to tell the server that a private messaging related command is sent, so that the server can restart the timer for timeout.

How the system works

In the server side, firstly, all necessary variables and server socket is initialized and start listening to connection. Then there will be an infinite loop keep accepting client's connection. Upon receiving every connection from the client, the server creates a separate User() object for this client and saves the connection socket and its IP address into the object. After that the server creates a dedicated thread for this user in which the thread executes the login_process() and main_process(). login_process() consists of all logics to authenticate the user, main_process() is used after the user has logged in to response to all the commands sent by the client and contact the client upon events like timeout or starting a private session with another client.

In the client side, firstly, all necessary variables and two sockets are created, one is for connecting to the server, one is to act as a server for later commencing private messaging session with other clients.

Then it enters the login_process() which helps the client to send its login details to the server, after logged in, a new thread will be opened to receive messages from the server and it enters the command_process() where the client can start typing commands and use the newly created receiving thread to receive response from the server.

For private messaging, suppose user A initiates a private connection with user B, A sends a message to the server to tell the server that he wishes to start a private session with B. If everything's fine (e.g. the client is not blocked by another user) then the server notices the B to accept a connection using its own private server socket, after that, the server sends the IP address and port number of A so that A can connect to B's server socket.

Upon B receiving A's connection request, B creates a new thread to receive messages sent by A, and upon A connects to B's server socket, A creates a new thread to receive messages sent by B. A and B uses the same thread to send message to each other as sending the command to the server.

When one of the clients in a private session want to stop the private session, it sends "stopprivate" command to the server, the server will then send application layer message: **WhatsApp stopprivate (1)/(2) with <username>** to both clients involved in the connection, clients will then stop the sockets.

Design tradeoffs considered and made

The application layer messages could be designed more complicated to contain more information, however, since this is a small application, it's not necessary to design a protocol as robust as HTTP, therefore, for the sake of simplicity, I limit each application layer message in one line.

Also, in the server program, active P2P sessions are stored in the format {A:[B,C]}, which implies A initiated a private connection to B and C, this saves a bit of space but require more work on searching during private chatting. I pick space efficiency over search efficiency because normal messaging usually occurs more often than private messaging, therefore, searching won't be required a lot, saving active sessions using less space would be more ideal.

Possible improvements

This application doesn't handle the case when the server or the client program clashes unexpectedly, errors and exceptions are thrown upon this kind of events. To make the application more robust, I can add some try/except/final statements to handle these cases. Moreover, this application assumes user logout in a standard way, i.e. by typing the logout command, then the server will perform some operations to keep everything in a consistent state (like deleting user-related data, removing the user from the onlineUsers list). However, if the user quits the program by typing Ctrl + C, all these operations won't be performed, therefore, all states remain the same as if the user hasn't logged out yet. To solve that problem, I can add try/except statement around my code to update the server's state whenever a client program clashes unexpectedly.

Possible extensions

One possible extension would be adding GUI (Tkinter will be a great choice) for this application . Now sending commands to the server would be done by pressing buttons on the application screen. For example, there will be an input block to allow user enters the message and a <send> button to allow the user to send the message.

We can also allow creating a group for users so that users inside a group can receive the messages sent by other groupmates, messages are invisible from users outside the group. This can be realised by creating a dictionary with GroupName: listOfUsers mapping. Each group has a leader, who can remove or add users from the group, this can also be saved in a dictionary {groupName1: leader1, groupName2: leader2}, afterall, hash table is highly efficient for accessing data.

We could also allow each user to maintain its own profile, for example, there are commands that allow the user to change its username, password, home town, email address, etc. Other users can view a user's information using a predefined command.

Final Notes

These codes were tested comprehensively so they should work under typical usage scenarios, however I can't guarantee there won't be race condition issues, please try not to do concurrent stuff too quickly.