How it Works

This project contains 5 files. Server.py, client.py, logger.py, user.py & group.py & uses python version 3.7

First, start the server by executing the command:
python3 server.py <server_port> <number of attempts>

Next, only when the server is successfully started, run the client file.
python3 server.py <ip_address> <server_port> <client port>

The user will be prompted to login with a username and password. Upon successful login, the user will be presented with the following commands as displayed below:

Command Name	**Command Function**
/msgto	Private message: Launch a private chat with another active user and send private messages.
/activeuser	Display active users.
/creategroup	Create Group: Build a group chat room for multiple users and send messages in the group chat.
/joingroup	Join Group: Join an existing group chat.
/groupmsg	Group Chat Message: Send a message to a specific group, and all the users in the group will receive the message.
/logout	Log out.
/p2pvideo	P2P Video: Send a video file to another active user directly via UDP socket (for CSE Students only).

Design

Multiple clients can connect concurrently to a single server.

Most of the code logic is concentrated in the server.py file. Exceptions include checking if the file exists on the client side before requesting for p2p sharing, login, etc.

Created User class to store user info:

- Client socket
- Username
- Password
- Ip
- Port
- Is active
- Is locked (to determine if account is locked)

- Lock countdown
- Num of attempts left
- Login time

Created Group class to store group chat info:

- Group name
- Group members
- Group message count

I have used the sample starter code provided on the COMP3331 forum which used threads to allow clients to run concurrently.

The code logic for each section of the assignment (eg. /logout, /login) is written in separate methods for readability and organization. Thus, the run method of the thread only contains the if – else statements that check for which specific function has been requested by the client. This avoids massive chunks of code logic concentrated in one portion of the file.

Application layer
Server is persistent. TCP connection remains open after sending a response.

Enables concurrent requests and responses with the help of python threads. Client sending message is occurring concurrently to receiving a response from the server.

Pipelining is used.

Moreover, the response client receives from the server is handled by python Threading. This means that the sending and receiving data to and from the server is independent. Thus, it enables the added functionality of receiving a response from the server in cases such as a group message being sent out even without the client needing to enter an input.

Trade offs

- In order to allow the client to receive messages from the server even without first entering an input (eg. Cases of receiving a group message or p2p message), the receive message and send message of the client were coded as concurrent executions. Thus, the client could receive a response from the server while pending a user input. However, due to this concurrent implementation, the user prompt message would always be displayed right after the previous input was entered, even before the response from the server was achieved. Thus, it seemed like there was an error in implementation since the message for user input prompt is displayed twice before the response from the server.
- Thus, to resolve this unique interaction, a delay of 0.01s was introduced between user input to give some time for the server to reply. While this works majority of the time, it will potentially encounter the issue once again if the server's user database is larger and more processing time is required to execute the user input commands, leading to the server response time exceeding the time delay of 0.01s
- A different response was needed for each of the different cases of each functionality of the messenger app. Consequently, there was a lot of repeated code when sending and receiving code in server.py
- In the P2P video sending, I initially attempted to send a message from client to server to notify the end of video file. However, due to possible packet loss in transmission, the termination string was potentially not transmitted, leaving the server socket open

indefinitely. Thus, I resorted to using a timeout to terminate the connection after a period of 10s.

Possible Improvements

1) Client socket, client_address is stored both in the clientThread class and user class. Perhaps the storing of the client socket is not necessary in the ClientThread since one of the attributes of client is an instance of user. This will assist in reducing redundancy in code
2) Currently, the attributes of user are access directly by ClientThread. While this does not affect its functionality, it is a breach of law of demeter (principle of least knowledge in object oriented programming) and introduces a potentially tightly coupled class. To improve the code, eg. Instead of calling self.user.get_username(), I could have created a method in clientThread to perform this function.

Possible extensions to the program

1) Include a change password functionality. /changepassword
   - User prompted to enter their existing password before being able to change to a new password
2) Include a leave group chat functionality. /leavegroup
   - User inputs the name of the group chat they want to leave as the 2nd argument
3) Include a delete message functionality. /deletegroupmsg (and/or) /deletep2pmsg
4) Create broadcast group. /createbroadcast
   - Only the owner of the group can send messages to the group chat. All other members can only receive messages.

References

https://realpython.com/intro-to-python-threading/

https://realpython.com/python-sockets/

https://docs.python.org/3/library/datetime.html

https://pymotw.com/2/socket/tcp.html

https://pythontic.com/modules/socket/udp-client-server-example