COMP3331 Assignment Report Kellen Liew(z5312656) 19/11/21

Code Used: Python Version 3.7

Program Design

The base of the program's design is based off of Wei Song's template found in the assignment folder. It offered a good starting point which I then used to implement the rest of the assignment in the order of the tips.

The server listens for clients, and when it does make a connection, it opens a clientthread class to handle clients in parallel with one another. Each client thread then receives and responds to the inputs from the client application.

Clients are a bit more complicated. The client app requires multiple different processes to be running in parallel with another. We need a msgthread class to be constantly listening to the server and either printing the messages or passing them on to the cmdThread, which handles user inputs. In addition we also have a timer thread, which handles timeout logouts. This resulted in a one to many cardinality between server clientThreads and client owned threads, which required application layer formatting whenever a server communicated with clients as discussed later.

Whilst it would've been nice to fold some of these threads together, the nature of the input and recv lines meant that combining these threads would make them get in the way of each other. Threads also often required references to each other, making them more coupled and increasing development difficulty, especially in implementing new threads for p2p messaging functionality. Another drawback to be considered for these threads were the number of while loops that needed to be closed for a user to logout properly.

For p2p functionality in the client app, we added two additional sockets for the client class, one would act like a server socket listening for new connections. Whilst another would act like a client socket which given a server socket port could then connect to another client. This added complexity as in a p2p connection, a client app could either be acting server or acting client and thus required different methods for both possibilities when using the connection. In future iteration, I could try to unify some of these methods if possible.

Data Structure Design

The usage of data structures were important in allowing the server to reliably and efficiently retrieve information, returning it to users or using it to decide whether certain actions should proceed or not. I identified the following information and represented them with their corresponding data structures.

- lockedUsers: Dictionary: Key-> Username, Values-> Time of account locking (referred in spec as ‘blocked’ accounts)
- userLastOnline: Dictionary: Key-> Username, Values-> Time of last logout
- userSockets: Dictionary: Key->Username, Values-> Sockets used in user’s clientThread
- mailbox: Dictionary: Key->Username, Values->An array of offline messages sent whilst user was logged out
- Blocks: Dictionary: Key->Username, Values->An array of usernames that the user has blocked
- Dms: A list of private messaging pairs. The pairs are represented by an array with three elements, the first 2 indexes are the two userNames involved in the private message, the third index contains the status of the connection (invite, in use)

Application Layer Message Format

When communicating to the server, there wasn’t much of a need to use a format in converting and passing user inputs towards the server. This is because all of the users inputs already have, consisting of a command first, followed by arguments that may be relevant to the command, all are separated by spaces. We simply REGEX out the command, and confirming it’s there can use the string split function to extract out all the arguments.

There was however a need to introduce message formatting for server to client communication. This is because of the aforementioned one to many cardinality, but also because different messages from the server require different responses. Some message are meant to be printed in the client app when received, whereas some are meant to invoke an input from the user. As such, I started with 2 formats in the first part of the assignment, adding a third in implementing private messaging. They all contain a header consisting of 3 letters separated from the rest of the message by a space, hence they are easy to remove by taking out the first four characters

- MSG - Tells the receiver(msgThread) to print the message
- CMD - Tells the receiver to pass on the message to the command thread, in this case msgThread contains an instance of cmdThread that it can pass to
- DMS - Tells the receiver to implement special actions regarding private messaging, helps keep the private messaging component seperate

How the System Works

Login - This is separate from the main loop that the server and clients run, this is possible because the number of options at this stage are quite limited, the client can’t call any other

commands and can only enter a username and password. As such, the process resembles a kind of sequence. It allows for both registering and logins. If the user arrives at a dead end, for example having an invalid username/password, the login function can simply print an error message and call itself, in the case of locked accounts, after a period of waiting. Servers use the userSockets dictionary as a reference for currently online users as well as the lockedList as a reference to deny certain users access.

Locking - After 3 incorrect attempts, the server assigns a username a locked status by adding it to the lockedList dictionary with it's lock time. Clients trying to login to a username that is locked get trapped in a while loop until they quit or they wait out the lock period

Logout - We first send a logout message to the server, which responds with a confirmation message. This allows the server to update information like userSockets and last logouts. Upon receiving confirmation, we end the threads and close the connection

Timeout - When a client is logged in, a local timeout thread is started which checks the time from last input. It has a 0.5 second sleep between each check to not be too intrusive. The timer is reset from each loop of the cmdThread which handles all user inputs, including private p2p messaging, so a user messaging directly and not with the server won't be timed out.

Whoelse and whoelsesince - These use the clientSockets dictionary, since the server already keeps a list of active client sockets it can reference. Whoelsesince also adds the userLastOnline dictionary which adds last logouts to include offline users who are within the specified time.

Messaging, Broadcast and presence notifications - Messaging uses the standard communication channel, broadcast utilises the same mechanics but for loops it through userSockets and presence notification is a broadcast that runs after a user login/logout.

Offline messaging - We do a check on each message to see if the user is online. If not, we add the message to the username key in the mailbox dictionary. All users in credentials.txt before server start have an entry and signups also initialise an entry with the mailbox.

Blacklist messaging - The server has a dictionary to manage user blocks. We didn't need to do too much refactoring to add this in. Some actions require a check to see if the block dict entry contains the sender. The server also has a method that removes all blocked users from a list for a specific user.

P2P messaging - Through the use of more threads and sockets we got it to work like regular broadcasting. See program design. Thanks for Reading!!!