COMP9331 Computer Networks and Applications Assignment

Name: Wenke Yang

ZID: z5230655

Part I: Program design and implementation **[This program is written in Python 3.7]**

1. Client side

1.1 Data structures

(i). **conns**:

The data structure conns should contain all connections maintained by the client. This including the connections between the server and all private connections with other clients. When a client initiates a connection with another client and the connection is accepted, the username and its socket would be stored to conns and the connection state is True, indicating that this is an active connection. Same happens when the client has just accepted a connection from another client. Once a connection is confirmed to be closed, the username and all information along with this connection will be removed from list.

The data structure conns is a hashmap. The key is either the original server or the username of a connected client. The value of conns is another hashmap containing the corresponding socket and the connection state. The state is True for the connection is active, False for the connection is unavailable now but not officially closed(i.e. the connection is set up, but the other client is now offline).

(ii). **clientSockets**:

The data structure clientSockets should contain information about a socket that has connected to the client. As well as the corresponding username that this socket connects to, and IP address and port number that the socket is communicating with. There are some overlaps of data stored in clientSockets and conns but clientSockets can be used for searching the username by the socket, while conns can only used for searching the socket by username. The data structure clientSockets is a hashmap. The key is the socket, and value is another hashmap containing the corresponding IP address and port number, and the username.

1.2 Sockets

At the client side, I have divided the sockets need to be set up for all connections into three categories. The first category is one socket (clientSocket) that is used for communicating with the original server only. The second category is a socket (privServerSocket) that acting like another server at this client which is used to wait for private connections from other clients, the client sockets accepted would be stored for receiving data later. The third category is the sockets (privClientSocket) that will be created when the user at this client wishes to initiated a private connection with another client, one at a time.

1.3 Multi-threads

Three daemon threads are designed to be implemented at client side, each used for:

- recv_thread - receiving messages from clients and server (including private and public)
- priv_conn_thread - accepting private connections initiated by other clients
- input_thread - waiting and handling user inputs and send them to correct sockets (including initiate a connection)

All data structures mentioned above would be shared by all threads, the race condition is controlled by aquiring the locks whenever some data structures need modification and release the lock when finished.

2. Server side [The server was designed to keep running forever until force quit]

2.1 Data structures

(i). **credentials**: username and password information for authentication.

(ii). **clients**:

The data structure clients is used to store clients' usernames and their information, this including:

- login time - the time of the user's latest login
- last activated time - the time that the user send his/her last message actively to server
- blocked time - the time that the user was blocked by server due to 3 wrong attempts of password
- socket and address - the user's current socket and IP address, port number
- offline message - if the user is logged out, the messages sent by other users would be stored here
- blocked users - the list of users that this user has actively blocked
- blocked by - the list of users that has blocked this user
- status - the status of user, can be login, logout, blocked(3 wrong attempts), authen(authentication state).

(iii). **addr_to_user**

The data structure addr_to_user is the reverse mapping of client socket and address to username. This is also used to store a client socket that is in authentication status (a client has just connected but not yet logged in), so that it has a socket, an IP address and a port number, but an empty username.

2.2 Sockets

At the client side, one server socket would be used for waiting connections from clients. Each time a client socket accepted, the client socket and the corresponding user information would be stored to the data structures in 2.1.

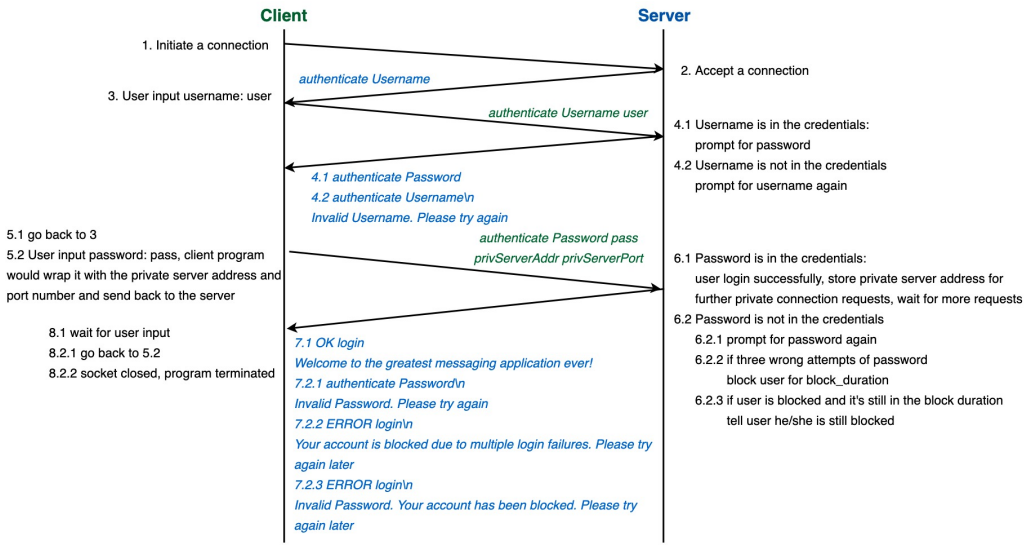
1.3 Multi-threads

Three daemon threads are designed to be implemented at server side, which are similar to client side. Also, lock is used for shared data structures. Each thread is used for:

- recv_thread - receiving messages from connected clients
- conn_thread - accepting connections initiated by clients
- send_thread - responsible for comparing the current time and last activate time of online users and sending the timeout message. Also, send offline message when a user is logged in and the offline message queue is not empty.

Part II: Program execution cycle and message format

Stage 1: authentication *[Note: message is shown in italic text]*



Stage 2: Requests and response between client and server

Client A request	Server actions (information before '\n' is not shown to user)
message <user: B> <message: msg>	- B is online, send msg to B: RECV message \n msg; send to A: OK message \n - B is offline, append msg to B's offline message list - B has blocked A, send to A: ERROR message \n Your message could not be delivered as the recipient has blocked you - B is A, send to A: ERROR message \n Cannot send message to self - B is not found, send to A: ERROR message \n Error. Invalid user

Client A request	Server actions (information before '\n' is not shown to user)
broadcast <message: msg>	- send msg to all online users, if user is not A, and A is not in user's blocked user list: OK broadcasted \n msg - send to A: - if all delivered: OK broadcast \n - else, some one blocked A: ERROR broadcast \n Your message could not be delivered to some recipients
whoelse	- send online user list except A back to A: OK whoelse \n <online users join by '\n'> - if no other online user: OK whoelse \n No one else is online now.
whoelsesince <time:t>	- check login time for all users in clients hashmap, return list of users' login time > now - t: OK whoelsesince \n <users> - if no user's login time within t: OK whoelsesince \n No one else is logged in since t seconds ago
block <user: B>	- add B to A's blocked_users list, add A to B's blocked_by list, send to A: OK blocked \n B is blocked - if B is invalid/self/has already been blocked by A, send to A: ERROR block \n Error. Cannot block nonexistent user / Error. Cannot block self / Error. You have already blocked B
unblock <user: B>	- remove B from A's blocked_users list, remove A from B's blocked_by list, send to A: OK unblocked \n B is unblocked - if B is invalid/self/is not blocked by A, send to A: ERROR unblock \n Error. Cannot unblock nonexistent user / Error. Cannot unblock self / Error. B was not blocked
logout (when A receives OK logout message, it tells all its privately connected sockets that he/she is logged out. Then, stops all connections and exit the program)	- logout A by removing its socket from addr_to_user hashmap, change the status of A in clients list to logout, broadcast all online users A logged out except users in blocked_by list of A: OK broadcasted \n A logged out; send to A: OK logout \n. Then, close A 's socket. - **server has a send_thread checking the current time and last active time of A, if the given block time reached, server would do the same as above and send to A: TIMEOUT logout \n Timeout. You are logged out due to inactivity**

Stage 3: Peer to peer messaging (information before '\n' is not shown to user)

Client A request	Server actions	Client B actions
startprivate <user: B> (i). send startprivate B to server socket (iii). create a client socket, connect to B. If not able to connect, tell A: B is offline. If B is in the connection list, reply: B is already connected. (vi). reply username to B (viii). A can start messaging with B	(ii). if B is valid, send B's private server's address and port number to A if blocked by B/ B is self/ invalid user, return ERROR message	(iv). accept the connection from A, save the client socket of A to B's clientSockets hashmap and conns hashmap (v). ask A's username (vii). store A's username with its socket
private <user: B> <message: msg> (ii). send private B msg to client B's socket - if B is not in conns, show ERROR has not started private session with B,. - if B is in conns but status is not active, then B is offline after connection established, show ERROR B is offline now - if B is self, show ERROR can't send to self	— —	(i). B's recv_thread would waiting messages for each connected sockets in loop. (iii). Once it receives msg from A, msg would be shown to user and the program replies: OK private \n
stopprivate <user: B> (i).send stopprivate to B's socket (ii). pop B from each connection data structures.	— —	(ii). B received and reply OK stopprivate \n, and close the socket with A

[Note: Requests not shown above would receive: 'ERROR command \n Error. Invalid command' from the server]

Design tradeoffs and improvements

One of the tradeoffs is the time to sleep at both server and client, time.sleep is set to 0.1 to have quick response but increases the workload to both server and clients. Another tradeoff is for both server and client, two data structures stored socket and username duplicately but one has socket as key another has username as key, this increase the storage but speed up looking up. One of the improvements could be to put some checking works to clients to reduce the workload on server. For example, the validation of message formats.