# COMP9331 Assignment 1 Report

## The Program Design

**Version 1 – Build a simple client server model.**

As required, I built a messaging application, based on a client server model, using TCP.

**Server -** Firstly, I built a the server. Since the server will handle multiple clients, I implemented a multiple-thread style code. After creating the server socket and binding the IP address and port number, the server socket starts listening. Once it accepts a new connection from a client, a thread to handle the client will start. As for the thread part, a function named *user_thread* is defined to receive messages from each client and deal with messages accordingly. This function mainly achieve two functionalities, to decide whether the user could log in successfully or not based on its username and password and to make further processing on incoming massages. A flag named *login* and a while loop are used to control the login process. A function named *user_authentication* is used to check login information. If the combination of username and password exist as a key-value pair in the dictionary named *authentications* and the user is not online already, then it will send a message indicating successfully login. Otherwise, it will send a message to let the client know the process failed. For the handling messages part, a flag named *connection_state* and a while loop are used. In the loop, once receiving a new message it will split the message into a list by " ", and process according the first element of the list.

**Client -** Then I built the client part. Same as the server part, the input is firstly checked. Since the user name and password need to be authenticated by the server, I divide the client into two main tasks, one is logging in and another is receiving and sending messages. For the first part, a flag named *log_in* and a while loop are used to control the process. If the client received message from the server implying successfully logged in, then the flag will be set to True, and the loop will be ended. Otherwise, it will keep prompting the user to input username and password. For the messaging part, a flag named *connection_state* and a while loop will be used to control the process. While the flag is True, it will start a thread to call the function *receive_message* to handle incoming messages, and in the loop it will keep scanning the input and deal with the message to be sent accordingly. In the function *revceive_message,* every incoming message will be split by " " into a list, and the function will deal with it according to the first element of the message list, whether to print it or to do some further processing.

**Version 2 – Implement the functionalities as required in part one.**

The functionalities are implemented based on the simple client server model above which can handle multiple clients. In this version, the structure of the code is remained. In addition, some functions are defined for the functionalities and they are called in the *user_thread* function when needed.

**Server -** Some global variables:

- *online_user_dict* - To store all online user's information, in the format of: {'username': 'user address'}
- *login_time_dict:* To store the login time of all users, in the format of: {'username': [login time 1, login time 2, login time 3, …]}
- *blocked_dict:* To store the blacklist for each user, in the format of: {'username': ['username 1', 'username 2', 'username 3', …]}

    - *fail_dict:* To store the consecutive failed login attempt times for each user, in the format of: {'username': times}
    - *offline_message_dict:* To store all offline messages for each user, in the format of: {'username': ['message 1', 'message 2', 'message 3', …]}
- **User Authentication:** The login part has already been achieved in the previous version. For blocking users for specified interval after 3 unsuccessful attempts, a dictionary named *fail_dict* is created and a function named *check_failed_times* are used to implement this functionality. When a user just input an invalid password, the function will be called.
- **Message Forwarding & Offline Messaging:** A function called *send_message* will be called when the server receiving a message starting with "message". The function will firstly check if the receiver is valid, online and blocks the sender. If not, the message will be sent, otherwise, an error message will be sent to the sender. If the receiver's username is valid and the receiver doesn't block the sender but it's offline, the message will be append to the list in *offline_message_dict,* the key of which is the receiver's username. Every time a user logs in successfully, the server will check if there are offline messages in the dictionary. If so, it will send to the receiver and clear the list.
- **List of Online Users:** A function named *whoelse* will be called when the server receives a message of "whoelse" from a client. It will get the keys of *online_user_dict* excluding the username of the user requested for it, put them in a list, and then send it to the user.
- **Message Broadcast:** A function named *broadcast* will be called when the server receives a message starting with "broadcast" from a client to send broadcast messages to all users currently online. It will loop *online_user_dict*'s key, and if the sender isn't blocked, it will send the broadcast messages. Otherwise it will inform the sender the message can't be send because of being blocked.
- **Presence Broadcasts:** Once a user logged in successfully, the server will generate a message of "username logs in", call the function *broadcast*, and pass the message and the name of the user who just logged in to the function.
- **Online History:** A function named *whoelsesince* will be called when the server receives a message starting with "whoelsesince" followed by a time duration from a client to send a name list of users who logged in within the past *duration* of time. First, it will get the current time. If the *duration* is greater than when the server started running, it will send all the users who have logged in excluding the requester. Otherwise, it will loop *login_time_dict*, and check the last element of the login time list for each user. If the current time is less than the *duration* time, it will append the username to the list, excluding the requester. In the end, sort the list in a alphabetical order, and then send it to the requester. In addition, if the input cannot be converted to float, then it will send an error message to the requester.
- **Blacklisting:** A function named *block* will be called when the server receiving a message starting with 'block'. First it will check if the username is valid, if so the function will append the username to be blocked to the value of the key which is the username of the requester in *blocked_dict*. Otherwise it will send an error message to the requester. If a user requests to unblock another user, a function named *unblock* will be called. It will also check if the username is valid, then delete the username from the list which is the value of the requester's username key in *blocked_dict.*

- **Client -** The client part mainly prints the message it received and handles the logout functionality. Once a client sends a message of "logout" to the server, the socket will be closed, and the server will delete the username from *online_user_dict*.

**Version 3 (the final version) – Implement the peer-to-peer messaging functionality.**
The main idea of implementing this functionality is to create new sockets for the two users who will be involved in the p2p chat. Let's say user A wants to start a private chat with user B. User A sends a message "startprivate B" to the server. Once the server got the message, it will check if the username B is valid, online and B doesn't block A. If so, it will send back a request message to B, along with B's own address and A's username. After a 0.5 second sleep, it will send an approval to A, along with B's address and B's username. Once B received the request, it will create a new socket, bind the socket with the address pair it received, and start listen. In this way, B's new socket for the p2p messaging acts like a "server". When A received the approval from the server, it will create a new socket and connect to the address of B received from the server. In this way, A's new socket acts like a "client". Once B accepted A's connection, the key-value pair of {'A': 'socket of A'} will be added to a dictionary named *private_connections,* which stores the combination of username and its socket for each p2p messaging connection so that when multiple p2p connections established it could know the message is from which user and send the message to which socket. This step is vice versa for the user A. In addition, since user B acts as a "server", it will store its socket in a dictionary named *private_local_host*, the key of which is the username of another user in a p2p connection which is A in this case, and the value of which is its own socket. Once the p2p TCP connection established as above, user A and B will both start a thread and call the function named *private_messaging*, which is for handling messages received from p2p connections.

## The Application Layer Message Format

I took the command showed in the specifications of this assignment, so for the client part, the message could be divided into two or three parts. I took the first word of the command as a header, the server will process messages based on headers, such as "message", "broadcast", etc. For messages send back from the server, they contain two parts: a header which indicates the source or sender of the message, and the content of the message, since there won't be the case that a username contains a space. If a message is a notification or an error message, the sender will be set to "system".

## Possible Improvements

As described above, I finished all functionalities in the first part except the timeout, and the second part of p2p messaging is finished. Thus, the timeout functionality is a problem to be solved in the future.
I wrote lots of comments inside the code files, which may be helpful to understand and trace the code.