# COMP9331 Assignment

Jude Michael Lim - z3463557

My program was coded using Python3. I completed everything except the extension.

## Program Design

The program is entirely procedural programming that relies on multi-threading. Although it isn't recommended, my program relies heavily on global variables for the client side and server side implementation. This was done to emulate the way a server retrieves information from a database. Organising things this way allowed for much easier readability, and a way for multiple threads to keep track of program's state and act appropriately. In order to minimise confusion, each function lists the global variables they are using at the top of the function.

The server side starts off using 1 primary thread and 2 sub-threads as daemons. The primary thread is the one listening to any TCP connection requests. Stopping this thread is equivalent to shutting down the server. Whenever a TCP connection is made, the primary thread creates a new thread to handle the client requests. That client thread runs until the TCP connection closes. The 1st sub-thread thread maintains the state of the server. Every second it updates user information such as: who is online, time outs, login blocking, and more. The second sub-thread checks to see if the user is connected every 10 seconds by sending a "PING". This is to see whether or not an unexpected disconnection has occurred.

The client side has 3 sub threads as daemons and a primary thread. The primary thread, after creating the sub threads as daemons, keeps a lookout on whether or not the program has timed out, or logged out, whereby it well then close the program. The first sub thread created is the P2P listener, which functions just like the server's and creates new threads for each connection made. The second sub thread listens to the servers responses, and the third sub thread handles the user's input. The third sub thread calls the relevant functions in response to user input.

## Application layer message format

All messages transmitted between client and server, are strings. Each end first of all prefixes the message with a header of 4 bytes that indicate how long the message is going to be, followed by the message itself. This header itself is 4 bytes worth of characters, and so the maximum size of the proceeding message is "9999" characters. This is done in order to ensure subsequent messages are not read mistakenly. Each end then interprets the message and acts accordingly.

**Server to client** The server either relays messages from other users prefixed with the name of that user, or sends response codes to previous commands issued by the client. These response codes constitute the typical, "Login Success," "Login Failure," or "Timeout," statuses, which upon reading, the client can act accordingly. The client knows which codes to expect and react to, and when it receives anything else, it simply prints it to console.

**Client to server** The client will enter strings starting with the command they want followed by the arguments. The command will either be handled by the client itself or sent to the server. Any invalid commands will simply be rejected by the server and an error message will be sent back to the client.

## How system works

### Server initialisation

The server first of all creates all global variables, stores the command line arguments, and retrieves the credential files from the same directory its source code is stored. It then proceeds to create the threads. Upon a successful client login, the server immediately sends any queued messages sent to the user while they were offline.

### Client initialisation

The client declares all global variables, stores command lines arguments, initiates its p2p connection thread, and attempts to login to the server using a specialised login function. If the login fails, the program terminates. Upon a successful login, the client retrieves and retains its login details which it uses later for p2p connections, and also sends the port number to the server that its p2p connection thread is listening on. The client then proceeds the create the sending and receiving sub thread.

### Client commands

Client commands are all handled by the sending thread. If this thread detects a "logout" or a p2p specific command, it executes the relevant function. If not one of the 4 special commands are detected ("logout", "startprivate", "private", "stopprivate"), it simply sends the user input to the server.

### Server state management

All information regarding the client is stored in global variables. There is a dictionary ("login_addr") that contains all the online clients, which has the client username as a key, and the client socket, IP address, and peer connection port number as the mapping. If the client's user name is not in this dictionary, then they are treated as offline.

There are also, timeout, login blocking, user blocking, last logged in, and an offline messages dictionaries, which each thread can access and update whenever needed.

The timeout and login blocking dictionaries contain a value that is decremented each second. The last logged in value is updated every second with the current time, as long as the user is logged in. If the user has never logged in, then they will not be in the dictionary.

If the client isn't blocked from logging in, or has no offline messages queued up, or has not blocked anyone, then they simply will not be in the dictionary, and the relevant checks for server functions will not be performed on/for them.

#### Peer to Peer

The client asks the server for the IP address and port number that the peer that they want to connect to is using. The server was sent this port number as soon as any client logs in, and stores it in the relevant dictionary. The client then initiates a connection with the peer they want to in the same way that they connect with the server. Once a TCP connection between peers are made, each client has the ability to close it. The P2P connection is automatically closed when either peers logout.

## Design trade-offs and possible improvements

The way the header stores information is inefficent. I've decided to keep the entire message format as a string to keep things simple. The 4 bytes allow a maximum declaration of 9999 characters for the message, which is sufficient for the purposes of this assignment. However this can be easily improved with a short int, which only needs two bytes, and can store a value of 65,535.

The global variables are also bad design practice. In this assignment it was convenient to do things this way, and seemed like a logical option for multiple processes to share information with. I suppose a better way to manage this type of interaction is separating database functionality from the source code entirely. This would mean getting rid of the server state management thread and providing a set of functions to both handle the program needs and manage any user related information in the system.

My code also leaves all TCP connections on and listening for requests non stop until the socket is closed by the clients. I've noticed at times that it my CPU usage would be extremely high when a lot of clients are connected to my server, despite no messages being sent. I think leaving the listeners online while they're not being used is a waste of resources. I suppose a better way to handle this is to time out the sockets, and when a request is made, to reestablish the TCP connection again. This timeout would be separate to the server timeout count.

The way my program handles the commands on the client side can be improved. For example it should specifically filter out p2p specific commands instead of sending it to the server when not enough arguments are passed through. There are also a few repetitive cases that should be optimised.