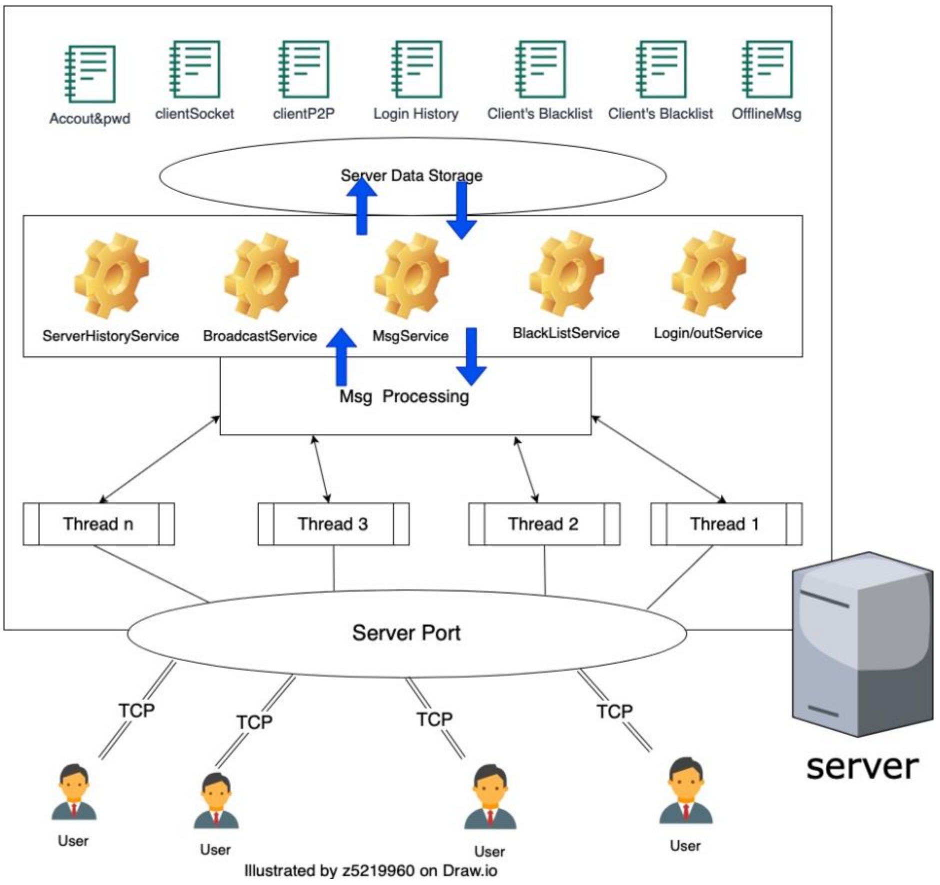
COMP9331/3331 Assignment Report

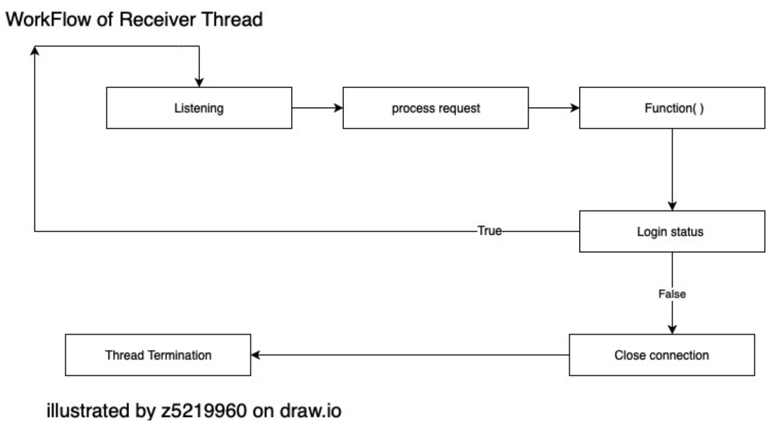
written by Lin, Heng-Chuan z5219960@unsw.edu.au
CSE student from Master of Information Technology (CSE grading criteria)

Server



Threading:

By creating 10 threads for connection awaiting, it would allow 10 (default) active users online in the same time. Once any client disconnected (i.e. specific thread would be terminated simultaneously), in order to accept the new connection, *Server* would periodically scan the status of threads to ensure every single thread ready to serve the upcoming clients.



Packet Message Processing:

The format of processed packet is simple. For instance, a packet contain text "LOGIN username password\r\n" is received. *Server* would probe the syntax until (included) first space which means

Server would check [LOGIN] and then it would recognize that this is LOGIN request then call the *Authentication* function to address it.

Login/out Service:

Including authentication, locks of multi-wrong-input, and logout function.

Message Service:

Processing the chat message toward either forwarding or offline storage.

Blacklist Service:

Storage of each client's blacklist. provide the result of query for supporting another function call.

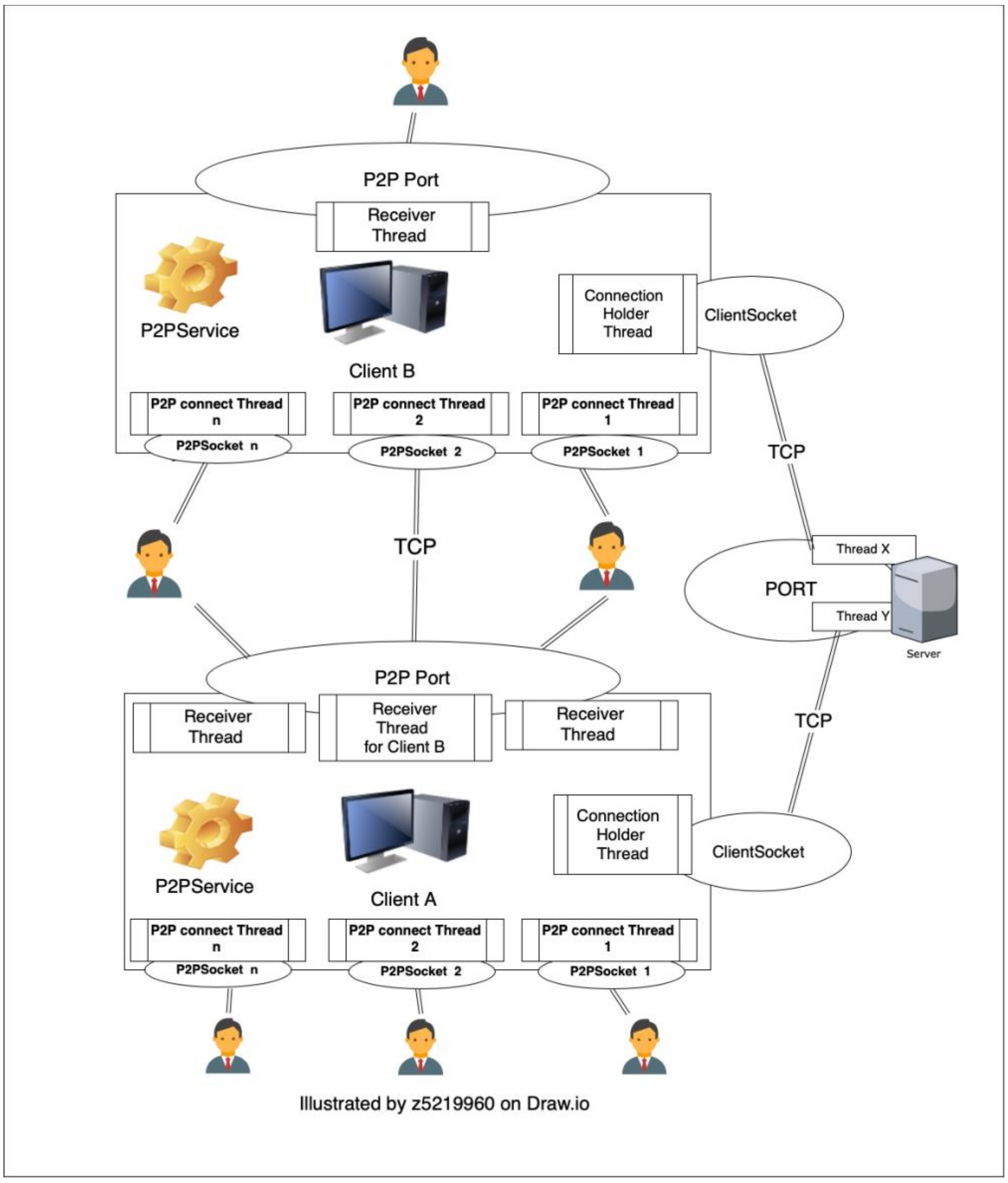
Broadcast Service:

Server would track the status of all online clients and broadcast them while their status changed.

Server History Service:

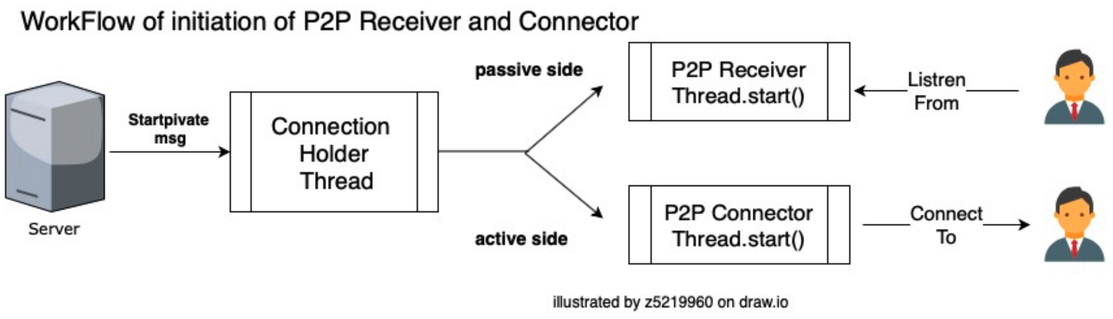
Record the time and status of each client and server itself for supporting other function call.

Client



Threading:

Threading implementation of client side is different from *Server*. There are 3 types of threading. For *Connection Holder Thread*, this is a unique thread for keeping the connection with *Server* behind the scene. On the other hand, in peer-to-peer connection, I consider that client itself is also a server accept request for other peers. *Receiver Threads* are built on the same port just similar with how the *server.py* is built. However, keep in mind that a single Receiver Thread only be set up when client received the request contain the name of another client from ***Server***. *P2P Connect Thread* works the similar way – it only be set up when client received the response contain the (IP, P2Pport#) of another client from ***Server***. Hence, this means that clients in a single Peer-to-Peer connection either connector(client-type) or receiver(server-type) thread is enable.


WorkFlow of initiation of P2P Receiver and Connector
Server
Startpivate msg
Connection Holder Thread
passive side
active side
P2P Receiver Thread.start()
P2P Connector Thread.start()
Listren From
Connect To
illustrated by z5219960 on draw.io

Peer-to-Peer Messaging:

provides the direct messaging between clients even in offline mode (after logout from server).

Feature implementation and Discussion:

I noticed that clients would trigger *timeout* from *Server* if they tend to chat in P2P way rather than use the forwarding service from *Server*. I found it quite annoying for triggering t*imeout*. Thus, after login to *Server* successfully, *client.py* would prompt a question about preventing *timeout* from *Server* while doing private messaging. Answer that question (Y or N) first, then you are good to use commands afterwards. If *PreventTimeout mode* is enable, every time when you use *private* command, *client.py* would send a pkt of an indication telling *Server* you're doing P2P instead of your private content to *Server*. Otherwise, *client*.py wouldn't sent any pkt to *Server* for *private*-relate commands (except *startprivate*)

other commands you might find useful:

extracommand	List all extra commands down blow on terminal
exit()	Exit the offline mode (only valid after logout)
listprviate	Get names of all Peers
whoami	Get name of your account

In offline mode:

You can still use the P2P messaging if you have established any P2P connection.
However, there's no implementation for reconnection to *Server*. (Actually, I came up with an ideal to remodify my *main()* into a *Connector Thread* to *Server* just like how I did for P2P connection.)

Finally, an implementation is applied for handling issues on both server.py and client.py about preventing crash when user terminated the program with Ctrl + C.
(Design: terminate thread while an entirely empty packet was received)

However, test the program with mercy, please use *logout* (from *Server)* and *exit()* to terminate the client.py. Thanks in advance ☺