COMP3331 Assignment Report

Basic Outline

Client side

Client side prompts login, sends details to server. It has a thread to constantly read in server data and accordingly output to its client terminal. It also has a thread to read in client connections upon initiation of private connections. User input is continuously read in a while loop and outputted to the server.

Server side

Does mostly everything. Use of hashmaps to store map of user name to Socket on server, login times, logout times, pending messages, peer to peer sockets etc. Has 5 main threads: ClientHandler, CommandHandler, BlockHandler, Timer and overarching Server thread. The timer is used to keep a count of server time, in use by the whoelsesince command. The clientHandler is instantiated for each client, meaning every client is run on a separate Thread. Blockhandler handles the timer for blocks, went to assert and deassert the block on specific users AND IP addresses. The commandHandler handles the interpretation of user commands (block, unblock, message etc.) There is also a logIn function which keeps track of the block counter (calls BlockThread when it reaches 3) and also a getCredentials function which stores the usernames and passwords from the text file onto the server. There is also a presence notification function used similarly to broadcast to notify unblocked users of others' logging in or out.

Application Layer message format

TCP connections are made between each client and the server. Server is initiated first to allow a welcomeSocket to open and accept connection on a certain port and IP. Client then connect to that same port and IP, and a data input and output stream is assigned to each client. For private connections, each client is assigned a socket acting as its server. The port for these sockets is given by the getPort() function of the original socket connected to the server. Each client that wishes to establish a TCP private connection with another client then receives this port and IP (which can be taken by the getHostAddress() function) and initates a connection from there to the requested the client. Since every client has a thread to receive incoming requests on an already opened welcomeSocket (initiated with the local port being the same as the remote port number of the socket connected to the server) the data can readily be transferred to and from clients via this client server combination. Again using data input and output streams, we can exchange private messages.

Design Decisions

It was decided early on that the client should do as little as possible. Hence it was decided not to use threads in the design of the client. It was only until discovering the task of P2P connections that it was realised this would become very difficult since every client needed to act as a server to listen for incoming client requests from the other clients. As a result, threads were added to the client. This resulted in an extremely high use of the CPU power when running 4-5 clients simultaneously.

From studying OOP, and observing the code, it can be seen that many improvements could be made with abstraction, encapsulation, inheritance and polymorphism, even though there is a lot of use of threads already.

Where the program doesn’t work

There are certain instances in the P2P section where the connection will only be established in 1 direction, meaning if the input is from user A saying startprivate B, the program will only register A sending to B and think that the private connection between B and A does not exist. Private messaging functionality is working fine, however. No segments were taken from anywhere on the internet. To write this code I made good use of the java help documents and the UDP threading example in the Assignment page.