Program Design

The program is written in Java and is designed where both the server side and the client side use multi-threading. Multiple clients are able to login and execute commands with the client interaction in the same server. Likewise, multithread client socket allows client to receive and send data across the server simultaneously. Furthermore, multithreaded server socket and client socket allow peer to peer-to-peer messaging between clients.

Data Structure Design

The three main classes are the Server, Client and User. The Server handles requests from the clients and send appropriate responses back. Sever class also contains data about every user in the system. The Client class handles commands that are issued by the user for the client interaction and send requests and get responds the back from the server. Lastly, the User object contains all information about a certain user who registered for the message system. The User object includes users' username, password, login status, blocked users, offline messages, etc.

Application Layer Message Format

The application layer message format for communication between clients and the server is a Packet object that contains the message that is being passed through, the sender's name, receiver's name, and the type. The Packet object also comes especially usefully when users want to send messages to other users.

How the System Works

When the server is started, User objects are generated base on the login credential file for reference and usage. Then, the server runs continuously and waits for any new connection from the clients. If a client socket is accepted, a new client thread is made that enables clients to communicate with the server.

On the client side, once the program successfully started up with the correct arguments, the client then immediately starts the process of connecting with the server. Once the connection is successfully, the program then proceeds to ask users to login or register.

Once a user logged in successfully, the procedure is mostly similar: a client issue a command -> a packet is sent to the server -> the server handles the incoming request -> the server sends back a packet containing the response to the client.

When a client want to initialize private messaging with another user, the following procedure is executed: the client the sends request to the server -> the server sends a packet to the target user to ask permission -> the target user sends back the response to the server -> the server notify the user that request the private messaging -> if the target user accepted the invitation, the private messaging is enable once the peer-to-peer connection is setup successfully. Else, the server will notify the user. Once the private messaging started, the two users will be able to send private messages to each other -> the private messaging ends when a user logged out or when a user wants to stop the connection.

Design Trade-offs / Improvements

Currently, the peer-to-peer private messaging system does not work correctly, I suspect that something went wrong when I initiated the peer-to-peer connection. Thus, this needs further improvement. Initially, I did not use multi-threading for the client side which caused a problem when the server sends multiple responses back from other requests. Hence, I implemented the multi-threading for the client side.

One improvement should be considered is to store all saved date locally/remotely so that when a server is restarted, the server can obtain all saved information about users and not just their username and password.