COMP3331 Assignment

Description

The message system designed implements 1 server in which several clients are able to log in and issue commands allowing for client interaction. Both the server and client program use multi-threading to both allow the server to handle multiple concurrent clients, client communication with the server and also to allow for peer to peer messaging of clients.

The system was written in Java and in the design, uses 3 main classes for most of the operations; the Server, Client and User class. While the server and client class are mostly self-explanatory with their name, the User class is what allows the server to handle the client's commands. The User object contains all the information about a certain user that is registered through the credentials.txt file. This includes their username, password, online status, blocked users etc. which is referred to by the server to process client requests. There are other classes involved in operation including 2 classes for peer to peer messaging however it is these 3 that form the foundation for the message system.

Application Layer Message

The application layer message format for communication between clients and the server is basic in the form of a Packet object which contains 2 main fields, and 2 optional subfields. The main fields are a packet header in the form of a simple string called "type" the message body with the variable "payload". The 2 subfields are the for the destination and source usernames of the packet however these are loosely used and are not always required for the server to be able to process the packet. The packet type is what allows both the client and the server to process the payload of the packet. There are 11 different types outlined below with their names denoting their usage with respect to the assignment specification. For the types that are not outlined in the spec, WELCOMEPORT is for getting the welcome socket port number from a client for peer to peer messaging and to send it to the server, and EXIT is for exiting the program. LOGOUT logs the user out of the system; however, the client would then be prompted again for a username and password; EXIT would completely exit the program.

- LOGIN
- WELCOMEPORT
- MESSAGE
- BROADCAST
- WHOELSE
- WHOELSESINCE
- BLOCK
- UNBLOCK
- STARTPRIVATE
- LOGOUT
- EXIT

System Operation

Start-up

When the server is started, a list of User objects is generated for reference and usage. These essentially are accounts for the message system which can only be accessed by using one of these accounts. After the user list is made, the server then continuously runs a welcome socket, waiting for any client programs to connect to it. If a client socket is accepted, a new thread is made (ClientHandler) and communication for that client is handled there.

On the client side, once the program is called with the correct arguments, the process of connecting to the server happens immediately and the client is prompted to log in to the system. In addition, a separate thread with a welcome socket is setup for peer to peer connection.

Login

For the client, the program asks for a username and password in which the client has to enter into the terminal. At this time there are no other commands the client can issue. Once a username and password has been entered, LOGIN packet with the details is sent to the server for processing. The server checks for the respective account based on username and checks if the login is valid. The server then sends back a LOGIN packet to the client containing the status code of the login attempt. Other than SUCCESS for a successful login, this code would also provide the reason for an unsuccessful login including if the user provided the wrong password, the account is locked, the user is already logged in from another terminal etc.

Once successfully logged in, both the server and client perform certain initialisation actions. The client program then sends a WELCOMEPORT packet to the server containing the welcome port number of the client. This number is then registered to the client's thread to be accessed when another user wants to start private messaging them. On the server side, the thread is registered with the client's username and their User object from the server. This becomes their unique identification which allows for the server to determine messaging, blocking, broadcasting destinations.

General Use

For all other commands, the procedure is mostly similar: A client issues a command, the command becomes the header for the packet with trailing information becoming the payload -> The packet is sent to the server. -> The server then refers to the user list to check and implement the command etc. blocking another user. -> The server sends back a packet with the same header (or a SERVER header which simply prints the payload to the terminal) detailing the outcome of the command whether it would be a simple notification ("yoda is blocked") or an error message.

Private Messaging

Private messaging is unique as it only goes through the server once and then the peer to peer connection is made for users to communicate directly to each other. For private messaging to start, a client first issues a startprivate command to the server. The server finds the respective user and returns their welcome socket port number to the requester if they have not blocked them. Once the packet is received, the client program then attempts to setup a connection with the user.

The process for manually connecting to a client and a client connecting to you are complementary. The initiator makes a socket to the target and then sends a message containing their username. The target accepts the socket and waits for that one message to arrive. Each client then starts another thread with that specific socket for communication. That message is used to register the destination

of the socket. This allows threads on both sides to know their respective destination user. This is used to identify the specific to allow private messaging to the correct person.

Regarding stopprivate, if a stopprivate command is issued and the user IS privately connected to that user, a private message containing “stopprivate” is sent. The other client thread identifies the request and then sends a “stopprivate” back. For a client thread, once a “stopprivate” is received, the thread begins to close. (Note: not been personally tested -> both clients sending a stopprivate at exactly the same time to each other).

Design Trade-offs/Improvements/Extensions

One major design trade-off is the format of the packet. Initially, the packet format was with a String header type and Object payload. However, when writing the system, for majority of the time, the payload was to be processed as a string. Some other format included passing integers, arrays including the InetAddress for private messaging. However the proportion of use cases for other types was not significant enough and therefore, the payload was set to a String. This was made for much cleaner code as previously, to string split a payload, it first had to be casted increasing the code size. A drawback of this would be the restriction of file transfer. For this specific assignment, while string payloads worked well, for further extension and improvement, changing the payload type to a more general form factor would improve with flexibility.

Another consideration was the responsibilities of the server. Initially, the server only maintained its threads and user referencing was done with a UserList class. If the a request was made to the server, the server would then refer to the UserList for processing e.g. block yoda -> server would perform UserList.block(blocker, yoda). However during implementation, it was found that most times, the server would simply pass on the parameters and perform nothing else than the function e.g. Block yoda -> server.block(user, yoda) -> would translate to return UserList.block(user.yoda). This happened to often and it was considered redundant. As a result, the user list is not generated and kept in the server class and not outsourced. While this increases the responsibilities of the server, this is acceptable for this spec and now the server focuses on two things only: connecting clients and processing users.

For general improvements, most processes in both the client and server could possibly be refactored into functions. Currently, both sides are processing commands through a large switch block that has several lines of code underneath each case. It is most likely possible to refactor the code to instead call functions for a certain procedure instead of performing each step manually. Another improvement can include the process of communication. The packet object used works well for the assignment, but a more robust format would be preferred both for improvement, more precise processing and possible extension implementation.

As it is set up now, several extensions can be made to the system. One such extension can include the adding and removing accounts in the server. Right now, the accounts are generated through the credentials.txt file, however it is possible to write addUser() and removeUser() functions that are called to modify the active user list. On a tangent from this, these functions could either be added through commands in the server terminal, or through an administrator account. This administrator could be implemented through the client program and another field to the user object would be added to check if the user is a registered administrator. From this, extra commands can be processed e.g. adduser obiwan highground a check for admin status is done for the command to be processed. Additional user management can possibly be added including changing username and password. All of this can be implemented by adding more cases to the switch statement and altering the credentials.txt if the changes are desired after turning off the server.