CISC 335 - Programming Assignment 2
Project Report

By Joshua Neizer

How To Run The Software

In a CMD or Terminal window compile all of the java files. After compilation, run the Server.java file, this will allow you to run the Client.java files. If the client files are run first, then they will attempt to reconnect until a connection is made. Once connected, have one client enter the waiting room, and another the connecting room. You can connect to the waiting client via the connecting room, seen by a list of available clients. Once the client is requested, said client will then receive a message and can respond by either accepting or denying the request. If accepted, both clients will be connected and placed in a chat window. and you will be able to communicate with each other from there. Use the given instructed commands to navigate through the menus.

Program Overview

The program that I have created follows the assignment outline with a few additional quality of life improvements. For example, for security purposes, the clients do not hold any information of who they are connected to aside from just their connected client's nickname. All of the data is saved and routed via the server to allow a more centralized approach to network communication which helps improve network security and organization. Moreover, I decided to implement the use of menus and rooms throughout the program as I believe that it allows for a more user-friendly experience.

`Server.java`

`Server.java` is the server instance for the network, running as a continuous thread. It has many responsibilities, but in essence, it centralizes the network routes communication between clients. It gives the client an identification string so that it can be referenced over the network, it also saves all of the user data sent to the server to allow other clients to have access to said data without having to go to the desired client directly. Lastly, it allows the user to connect to any other user on the network without having to know the user's data itself allowing every user to remain private on the network while also being able to easily communicate.

The basic flow of the program is that when started, the server creates a socket, creates a stack of available client IDs, and maps that are used to reference user data. From there, it continuously waits for incoming packets from clients, either requesting to join the network and setting up their information with the network, attempting to establish a connection to another client, or requesting to send a message to their connected client.

In the first instance, the client will send all of their networking data to the server so that it can be used to reference and route messages to and from the client. They will also be requested for their nickname so they can be referenced as more than just their ID on the network, for added personalization. Lastly, clients can request to see all of the connected clients to the server, allowing them to see all active and online users. In the second instance, a client will make a request to the server to establish a connection with another online client. The server will validate the request with their information, and if verified, will send a connection request to the requested client. The client's response then is validated, and the result is sent back to the source client. If the client accepts the request, then the connection is established between the two. This is all done based on the user sending the requested ClientID. The server doesn't respond with any IP address and/or port numbers as that is all handled server side, instead, it sends whether the request was successful and the client's nickname if the request was granted. In the third instance, when a message is routed through the server, it is buffered on the server, as a message is only sent if the client accepts the message. The server can then act as a gatekeeper, buffering and discarding messages accordingly. This allows the client to only have to send a message once, rather than a message warning that they are going to send another message, then the message itself. Moreover, the server handles the response to requests, allowing the client to not have to decode messages, rather just getting the important information needed for communication.

The server is written in a way that allows dynamic communication for the client, being able to route a message if necessary, while also setting up a new client. Its main goal is to connect all of the users in a centralized topology, provide flexible communication, and allow client side processing to be as simple as possible.

`Client.java`

`Client.java` is responsible for all client side communication with the server and other clients. It is the interface for the user to allow them to communicate with the network. The Client is designed to be state based, allowing for interactive experience with the user, while also trying to keep the terminal as decluttered as possible. This is done through the use of the `clearscreen` function which, note, may not work on all terminals due to differing operating system's CLI programming languages.

When starting the client instance, the client attempts to connect with the server and will continue to do so until a connection can be established. Once connected, The user will be able to personalize their client instance through a nickname that will be used by other clients when referencing them. After the setup is completed there are four different states the client can be in; the menu screen, the waiting room, the connecting room, or the chat room.

First the menu, the menu is simply there to allow the user to choose whether they want to connect with another client, whether they want to be connected with, or whether they want to exit the program. The reason this is done is because of reasons that I will further get into in the next section, however, essentially it is because of the limitations of a CLI interface. In the connection room, the user can see the other online clients they can connect with. Once a client is requested, the user waits up to 10 seconds for the connection to be accepted, if the connection is accepted then they are brought to the chat room. If the connection is denied or the timer runs out, they may make another request or return to the menu. Within the waiting room, the user waits until they receive a connection request, at which they can accept or reject the connection. The user has the option, every 10 seconds, to continue waiting or return to the menu. Two clients in the connection room can connect with each other, however, it is not recommended to do so because of the limitations of the CLI interface, inputs can be interrupted and misinterpreted by the CLI scanner. The final room is the chat room where users take turns sending messages to each other. Every message has to be acknowledged and either accepted before being displayed. The user also has the option to exit the chat room and return to the main menu at any time to start a new connection.

The client is meant to be intuitive and easy to use, despite being only CLI based. There are some limitations with communication, however, for basic communication, the client provides a simple solution.

Difficulties

A majority of the difficulties within this assignment stemmed from not understanding the limitations of the I/O interface being used and being over-ambitious.

At first, the main difficulties I was having were the simple ones of not being able to get information to transmit between clients. This was resolved by simple debugging, going back to lecture notes, and Googling. They were just oversight and misinterpretations made when I was first implementing the UDP sockets and didn't take too much time to figure out.

However, the biggest issue I continuously was that I was treating the system the same as I would for other messaging systems like Facebook messenger. Due to this, I tried to have users be able to connect at any time and be able to send and receive messages continuously. Allowing users to receive requests at any state in the program, aside from the chat room, provided a majority of mistakes as my client source code started to become overly complicated to a point where I wasn't sure how communication worked. Eventually, through debugging I was able to resolve

most of these issues. One of my solutions was separating the waiting client and a connecting client via the connect and wait rooms. The scanner, I learned, doesn't like to be interrupted with a new call; messages over the network can be confused if two clients request to connect to each other at the same time, or if the client attempts to connect to a client that's already attempting to connect to another user. Edge cases such as these are what caused a majority of my difficulties. However, since I based the architecture on the other messaging systems that don't experience the same limitations, I was fixing issues that could have been resolved faster if I had simply redone the architecture from the start, rather than fixing what I had written and coming to that conclusion much later on. In hindsight, I should have restricted clients to communicate with users only in the waiting room, this would have made my code much simpler.

The second difficulty I was having, again, stemmed from overestimating what I could do with a CLI. I initially had threads running for sending and receiving messages within the chat room. Although technically functional, if you received a message while typing, your message was cut in two. It became hard to understand what you were really writing, and the cursor for writing would be placed in odd locations. The biggest shortcoming was when a client exited the room, the connected client would also leave, but if they had their input scanner running in a thread, this would conflict with the new scanner being opened to get the user's menu choice. I tried for hours to try to resolve this, as well as try to have fixed positions for incoming messages. After some time, I realized and accepted the limitations of using a CLI for this, and decided to use a turn based approach for communication. Although not when I had initially designed for, it makes much more sense after seeing how the alternative doesn't work in a CLI. Although a solution for my initial approach may be possible, I decided it was better to finish the project rather than spending more time searching for an answer that may not be there.

Although I had plenty of other problems when writing the software that cost me hours to fix, they all stemmed from my misunderstanding of the CLI's limitations. This misunderstanding resulted in a simple program turning into 400+ lines of code (without comments) for the client alone.

Possible Improvements

As mentioned before, I would like to be able to implement a feature in which the user can see who's in which room, and restrict users to only be able to connect with users in the waiting room. It wouldn't be too difficult to implement, just additional checks made on the server side, and an additional message sent to the server whenever changing rooms/states.

Another improvement I would like to have is to be able to eliminate the need for turn based communication between clients. I am not too sure how that could be implemented, but that would allow for a more robust and seamless communication experience.

The last potential improvement would be to implement the bonus, however, instead of being able to chat with multiple people separately, a user could join and leave chat rooms that hosted multiple clients.