Kyle Kaminski & Jeff Paulson

Professor Wang

Data Communication & Networking

May 3rd, 2020

ChatRoom Project Report

Introduction

For this project, we decided to implement a Java-programmed chat room. This project was written in Eclipse because it would be easier to interact with the console for input and output, and because it works better for native JFrames than other IDEs. Before we describe the project itself, we would like to explain UDP sockets, how to program them, how they work, and how it was possible to use them with our final project.

According to Keil.com, UDP socket programming “enable[s] simple IP communication user the user datagram protocol (UDP). The User Datagram Protocol (UDP) runs on top of the Internet Protocol (IP) and was developed for applications that do not require reliability, acknowledgment, or flow control features at the transport layer. This simple protocol provides transport layer addressing in the form of UDP ports and an optional checksum capability.” In short, UDP sockets are a form of internet communication where the speed of the transportation of data is prioritized over reliability. On the contrary, the TCP protocol is used when the reliability of the data being sent is higher priority than the speed that it gets there.

A classic example of this UDP vs. TCP usage is in online multiplayer games. Among others, games like Call of Duty or Battlefield use a UDP protocol because they care more about getting constant client updates to the server. This is so the game does not play frame-by-frame while it waits for every single packet from all players, which may be nearly impossible if the player count even exceeds a few players. Meanwhile, TCP may be used for games like Street Fighter or Smash Bros. This is not only because those games’ maximum players-per-game is around 2-4, but it is crucial for those games to run frame by frame, in order to work properly; otherwise inputs may be dropped.

It is important to distinguish between TCP and UDP when we would start our project to make sure we are using the right protocol for the assignment. In short, this project is about using as local socket to emulate communication through UDP. The main challenges we will run into is the GUI implementation. When we were creating the JFrame customization, some lines of code would work (ex. setVisible() and pack()), and some would not (ex. setBackgroundColor()). We decided to leave the default GUI as is and spend our time focusing on the more important aspects of the project.

Design & Implementation

To begin, we knew what classes we wanted to implement right off the bat. We created a Client and a Server class. The code was originally split into multiple different classes because we thought it would be easier to make an Object for every event that we wanted to make. As we started programming, we found out that it was easier to just put everything into the Client and Server classes. It was also easier and more efficient to not make objects for the messages, and just print them to all the clients’ GUIs using the *synchronized* keyword (which means it does the same thing for all of the Client objects at the same time). These classes may not be used by the time we finish the project, but our initial ideas from these classes will be implemented in some way, shape, or form in the final build. In terms of what we need to properly run this project, it needs to implement a specified port, as well as a “server” and “clients” to work. We also need a testbench to run it (ex. Eclipse running on a Windows 10 PC).

As mentioned above, we initially designed the components to all be separate objects. This was initially thought to be easier, but along the way we found out we can just merge all the components into two classes and to not make them into objects, but to just run the same code at the same time to all the Client objects’ GUIs. A new construct that we ran into when we were researching Java libraries was the *synchronized* keyword in Java, which we have never seen or used before. According to Baeldung.com, the Java *synchronized* keyword is used in a multi-threaded environment because, “a race condition occurs when two or more threads attempt to update mutable shared data at the same time. Java offers a mechanism to avoid race conditions by synchronizing thread access to shared data. A piece of logic marked with synchronized becomes a synchronized block, allowing only one thread to execute at any given time.” So, with the *synchronized* keyword, it is possible to perform the same exact piece of code at the same exact time for all users. An example of this is when, in our chat room, a user says “Hi” to all the other chat users. The time stamp printed to all the users is the exact same timestamp, to the millisecond, for everyone. This is because the process of printing to each client’s GUI is being performed as the same exact process.

In terms of general option, we do not have much on the front end, since we spent most of our time on the back end to get the program working. For the front end of the project, we use two mechanics: the console and a GUI. The console initially asks for input of a server name and a server welcome message. These messages are sent to the client side when new clients join the server. Next is the GUI. We decided to use GUIs because Java already has a JFrame class for GUIs, and they natively work very well in Eclipse. All you need is a line to initialize it, a few mandatory fields (like adding a text field and making it visible), and some customization. We also chose a GUI for client spin ups because Eclipse does not like to show multiple consoles for different programs running at the same time. We can create different console windows, but they all print the same thing. Also, they all update every time one of the programs running has an output, meaning they all cannot show different outputs anyway. Using a GUI, it is easier to give each user a distinct menu for their chatting purposes.

Our chatroom server and client can be described in pseudocode. First is the Server class. This class must run before the Client class, so the Client class has something to connect to. Before anything else, we import the necessary IO classes, network classes, Execution classes (for executing multithreading), and Exception classes to suppress the exceptions in the console. We start with sets of created clients and sets of clients in the chat room. Then we create an object that allows our program to perform multithreading, which is required to use the chat room simultaneously with other users. A port is specified, and now we begin the main() method. The main() creates a multithreaded pool, connects to the specified port on our PC using UDP socketing, and executes the Handler() class. Our ClientHandler() class is nested in the Server class. The ClientHandler() class reads the socket we specified and tells the server to connect to that socket. The most important function we now run is, intentionally, called run() (required to use this name by the Handler() class) contains many different *try-catch* blocks, a *try-catch-finally* block, and even a *synchronized* keyword. The *try-catch* blocks try a block of code and catch any exceptions that may occur, in which it stops what it is doing to prevent the program from catching. This is used when we try to set up the client into the chat room, so the server does not crash because of a client-end mistake. The *try-catch-finally* block does the same thing, except when an Exception is found, it runs the *finally* block of code to keep the program going; we put a remove() method in here so if a client causes issues, we remove them to prevent further damage to the server-side of the program. Lastly, we use the *synchronized* keyword. This keyword makes sure all the code for all the clients is executed at the same time.

Next is the Client class. We imported the necessary JFrame classes, IO classes, customization classes, and Exception classes to suppress exceptions in the console that users create when they leave the chat room. The Client is given the default local address (127.0.0.1), then a JFrame object is created for them. Now we connect the Client to the Server. This is where the JFrame GUI becomes visible and prompts for input of a unique username. Once that is received, the Client is connected to the Server. The Client can send messages and receives messages from other users. When the Client wants to leave, they can X out of the window or type “/exit” in the chat (without “”). They are then removed from the chat room. Their window then becomes nonvisible again.

Testing

When talking about the server side of things, the Client receives a message (the message assigned at server startup) when they first join, and all other clients that were already connected are informed that there is a new client that joined. When a client leaves the room, not only does the console update to tell us that someone has left the room, but all clients receive the notifications in their message log. When a client sends a message, we can see that message pop up on all the GUIs of all the other clients’ logs as well. Also, when a specific client sends a message, their name is tied to it (it is sent within their own GUI instance), so all other users know which other client has sent the message. The Server and Clients also are notified of the current client count whenever a client joins or leaves. All these properties are handled by the Server to manage all the clients.

All the behaviors of the client side of the project are functional as well. When we click “Run As” on the Client class, we are greeted to a new GUI and a welcome message. When the client wants to leave the chat room, they can either X out of the window, or type “/exit” into the chat bar. In our code, this caused a NoSuchElementException, because when the server-side code would loop back to the beginning, it did not know why its client count is less, so it throws this error. We added a *try-catch* block to suppress this because the program still works as intended, it just throws an error every time. Next, when the client types a message, the server then receives the message. The server executes the code written for the Client object which allows it to broadcast to all the other clients in the chat room (thanks to that *synchronized* keyword). Next, we implemented a while loop with the condition of (true) for Client connection to the server. This means it executes forever until it is interrupted (in this case, when the Client object is destroyed and is garbage collected). This way, it will always stay connected, wait for messages, and will always be able to send messages. Therefore, each Client should do these two things at the same time, which fulfills the final behavior requirement: they all wait for user input and sending the message out, and they all wait to receive messages and print the message on the standard output.

Conclusion

When we started this project, we were a little hesitant at first because we both knew nothing on socket programming. After watching a multitude of sample videos on YouTube, how to implement this in our code, what the *synchronized* keyword was and how to implement it, we are very pleased with how this turned out. We are satisfied in being able to fulfill all the requirements. Before we started this project, there was a lot of mystery to what makes socket programming work. It seemed like a very simple concept but at the same time felt like a complex system to understand. Like other concepts in Data Communication and Networking, once we pushed through the initial level of understanding, socket programming became much less daunting. When we spent the time to learn the process and get to the root of functionality, implementation became possible. Overall, we had a lot of fun working on this project.

Group Description

Jeff- coding implementation, concept research, project paper review

Kyle- coding review, project paper, demo video production

Sources Used

Java multithreading:

https://www.tutorialspoint.com/java/java\_multithreading.htm https://www.geeksforgeeks.org/multithreading-in-java/

Java *synchronized* keyword:

https://www.baeldung.com/java-synchronized

Socket Programming:

https://www.keil.com/pack/doc/mw/Network/html/group\_\_net\_u\_d\_p\_\_\_func.html

Java Socketed Programming:

https://cs.lmu.edu/~ray/notes/javanetexamples/