CISC 335 - Programming Assignment 1
Project Report

By Joshua Neizer

How To Run The Software

In a CMD or Terminal window, compile and run the TCP_Server.java file. In as many individual CMD or Terminal windows as desired, compile and run the TCP_Client.java file. Note that only three clients can connect to the server at a time. When the Client connects, follow the instructions on the Command-Line-Interface to interact with the server. If there are issues with connecting from the client to the server, the port may need to be changed from "5069" to any open port on your computer.

Program Overview

The program I created is one that follows the outline of the rubric with a few additional features that allows a more seamless and intuitive user interaction. There are three files involved in the project; TCP_Server.java, Client.java and TCP_Client.java. The program can further be divided into two layers, the server layer, which utilizes TCP_Server.java and Client.java, as well as the client layer, which utilizes TCP_Client.java.

TCP_Server

TCP_Server is the main of the server, it controls setting up socket connections, creating client connections, terminating client connections, starting client threads, and stopping client threads. It is not a static class, so it has a main that defines the maximum number of clients that are going to be connected to the server, as well as the socket number for the server. As the server is started, it creates an array of Client data, of the specified maximum number of clients, and then creates a server socket. After the socket has been established, it creates a thread for every client that attempts a connection. The server then waits and listens for a client to connect, once a connection has been established it creates the Client instance for the new connection and starts a thread. It repeats this process until all maximum clients have connected. Once the maximum number of accepted clients has been reached, the server repeatedly checks every client connection, once every 3 seconds to see if any connection has been terminated. If this is the case, the server suspends the thread, gets the connection establishment and termination time from the thread, and waits for a new connection to be established.

The purpose of each client being a thread is so that there can be concurrent communication between the client and the server. TCP connection requires a handshake between both nodes before all communication, so with a single-threaded process, each client would have to wait for

their turn to communicate with the server. Multi-threading makes every connection independent, and thus unaware of how many clients are allowed to connect.

All of the connection information is printed to the console so that the user maintaining the server can see all of the connection requests and messages being sent from the client and the server.

Client

Client is the thread created from TCP_Server that allows concurrent TCP connections. The client handles all client requests to the server. Once instantiated, Client creates all of the input and output streams for messaging to and from the server. When the Client thread starts, that's when the connection becomes "active", as indicated by its attribute. Once running, the client asks for the Client's preferred name so that it may be identified from the other instances, this is separate from the id which is used to number the instances. The Client instance will then continuously wait to receive a message from the client node and then will respond accordingly. There are three unique messages it's waiting for; "exit", "list", and "file". "exit" indicates to the server that they are terminating their connection, and so it shuts down all of the I/O streams as well as makes the connection inactive, making it available for a new connection. "list" lists all of the files that the client can request. "file" prepares the server to get a file request, then sends the requested file to the client in 1 KB chunks. Any other message will just result in the server sending an acknowledgement that it received a message from the client node. The Client object is essentially a response program to the client, outside of the initial setup, it does not send any unprompted messages. The Client also tracks when the connection is properly established and terminated for the Server to keep track of.

TCP_Client

TCP_Client is the client connection that shares a lot of the same internal logic with the Client, except that, outside of the initial setup, it doesn't wait for messages from the server that haven't been requested. When a connection is first attempted, it sets up a socket with the socket number and IP address stated in the TCP_Client's main. If the server is not active, the client is put into a "waiting room" where it checks if the server is active every 10 seconds. If the server is full, the client waits for the server until a connection can be established. Once the connection is established, the client enters their preferred name and is then allowed to interact with the server. The client is then put in a loop entering messages into the command line for the server to respond to. There are three unique messages the client can send; "exit", "list", and "file". They follow the same implementation as outlined previously. Any other message will result in the server sending an acknowledgement that it received a message from the client.

There are a few features that make it better for user interaction such as a few instructions once the connection is established to make it easier for the client to use the program. Another feature is that if the client asks for a file, they ask for the file number per the list to allow easier I/O. The user can only input an integer, they can cancel the request with an input of “0” and there is error handling if the requested file doesn’t exist.

Error Handling

Any failed socket instantiation results in the program being shut down because there may be severe issues at play. Other than that, all other errors are handled according to and ensure that the client can’t compromise the connection based on input. The Client program also keeps track of all errors on the server-side, and if 5 errors occur, there may be some issues with the connection that will most likely persist, so the connection is terminated.

Difficulties

During the development of the program, I encountered several difficulties ranging from misunderstanding the assignment itself, to misunderstanding how Java uses TCP connections. The first difficulty I faced was trying to be able to create the network as outlined in the assignment.

My first understanding of the objective was to create a “chat-room” of sorts where clients could connect to the server to join to communicate with each other. However, this did not work as intended due to the nature of TCP networks. What ended up happening is everyone having to wait for their turn to be able to speak because the server could only communicate with one client at a time. I resolved this difficulty by coming to the professor and confirming what the assignment outline meant.

The second difficulty I had faced during development, after I was able to clearly understand the program objective, was being able to thread multiple clients. The issue was because I tried to accept sockets from inside the thread, not inside the main server program. This resulted in every client being connected to the same thread, not being treated as its own thread. After researching how to multi-thread TCP connections, I was able to separate the connections that needed to belong to the thread compared to the main. For example, the server socket is shared amongst every thread, as well, a thread will only start when the server can make a connection with the client. It then passes that connection to the thread rather than the thread waiting for that connection.

The third difficulty I faced was the Client's threads on the server side being viewed as inactive by the server. The issue was simply just that the thread has to ensure the connection is established, set up all of its I/O streams, make an id, and get a list of files before it can declare itself an active connection. However, the server did not wait enough time before checking if it was active and viewed the connection as terminated. I resolved this by making the server wait 3 seconds before checking the termination of any clients. Allowing new connections enough time to establish.

The last difficulty I faced was the most challenging yet. The issue was when I was transferring files from the server to the client, the client was always waiting for the last 8 KB of the file. Due to the library and functions I was using, the program would not progress unless it received the last 8 KB. At first, it was unclear why this was occurring because I was having some difficulties figuring out how to send files in the first place. However, what was happening was the ReadBuffer used to read in string messages from the server, buffers an additional amount of data to ensure a better experience for the user. This was an issue as I used the ReadBuffer to read in the file size since other methods were not working. This leads to the ReadBuffer buffering the transmitted file instead of the DataInputStream buffering the file, as it should be doing. The solution I had was to send a message from the client to the server after the server sent the file size to ensure the client's ReadBuffer didn't buffer any file data.

Possible Improvements

An improvement I could make is to use the FileTransfer library, as it allows a more seamless transfer of files without the headaches that using DataInputStreams caused. Ideally, I would want to give the whole program a proper interface along with more features, which would provide the program more purpose. Although it is a great tool for helping me understand TCP networks and how they are applied, the program itself does not accomplish a lot. By allowing other clients to communicate with each other, allowing users to send files to the server or interacting more with the files on the server; the whole program could be more interactive. Moreover, a CLI is a great first step, but a GUI would be a lot better for this kind of user interaction GUIs are more intuitive than CLIs.