CS 313 Project 1

Group 31: Multi-Client/Single-Server Chat Program

Keagan Selwyn Gill: 22804897@sun.ac.za

24/02/2024

Introduction/Overview

The goal of this project was to write a chat program based on the client-server model with the ability to add multiple clients to a centralized server, each with their own GUI, and can send messages to one another either on a local network or across different network connections via a VPN with LogMeIn Hamachi. The chat service required clients to give a unique username on startup. Clients could also message privately in the form of a whisper, with which messages are only sent to the specified user. All relevant messages sent, even ones sent by the same client, should be displayed on the client's GUI. A list of current online users could also be displayed with use of a specific command to send to the server. Java was used as the programming language and Java's built-in Sockets library for networking. To enable multiple client connections, we had to implement concurrency by having the main thread listen for client connections and spawn a new thread with a socket for each connected client. This approach allows the server to communicate with multiple clients simultaneously, maintaining a one-to-many relationship, and was achieved using Java's built-in 'concurrent' library. Both the server and any client disconnecting at any time should be graceful, and not disturb any other parts of the service. For the GUI I used Apache Netbeans 14 IDE as it provides a drag-and-drop GUI builder that allows developers to visually design their user interfaces without writing extensive code manually.

Unimplemented features

- GUI implementation of the online users list (online users can be displayed in the GUI via a command but there is not a graphical list of users that gets updated).
- No separate chat boxes for each client being whispered.

Additional features

Clients have the ability to change their username, the server will continuously request a unique username if the username provided has already been taken and this username will automatically be updated in the list of current online users.

Description of files

Source files:

Client class

- The Client class is responsible for connecting to the server and maintaining all input and output from the server. This class therefore holds the input and output streams and assigns these to the relevant threads on startup. The code generated by the Netbeans IDE which handles the GUI is also present in this class. This class also contains the code, that had to be manually implemented, which handles the operations that occur upon clicking the buttons on the GUI.

Server class

- The Server class keeps track of all clients connected. It is responsible for establishing new connections, and then creating and passing these on to the connection/client handler class (called ConnectionHandler). The server has a broadcast function, which allows it (and other clients) to send messages to all currently connected clients. It also stores administration information about users connected etc. and passes this on to all clients.
- A new ConnectionHandler is constructed for every client that connects to the server. The ConnectionHandler makes use of the server's broadcast function to send messages to all currently connected clients. Whispers are handled by the ConnectionHandler by sending only to the target of the whisper, instead of broadcasting to everyone by using the '/whisper' command. The ConnectionHandler also handles all the other commands that the clients can send to the server, which are '/name', '/users' and '/quit'.

Program description

The service comprises of two distinct parts, namely the server and the client. There is only one server for a given instance of the service, that handles multiple clients at a time. How the server does this is by having a connection/client handler thread handle each client separately and managing the connection/client handler threads instead of the clients themselves. The clients make use of sender and receiver threads (called out and in respectively, within the code) to contact the server, maintaining concurrency and allowing for server interaction. The program flow starts with a running server and at least one running client. The client(s) send messages via their sender (out) thread to the server. These are then either broadcasted to all users or sent to specific ones accordingly.

The clients then receive these messages using their receiver (in) thread and display them accordingly. Administration messages (such as users connecting/disconnecting) sent by the server are received by clients in the same way. Any part of the service disconnecting only breaks the chain of flow in the service and does not affect the components it is connected to. This means that a client that is connected to the server, disconnecting, does not affect the server, but rather just notifies it. Likewise, the server disconnecting/stopping, only stops messages from being sent using the server.

Experiments

Experiment 1

Question: Will the time taken to open a new socket change based on how many users are connected to the server?

Hypothesis: The time taken to open a new socket will not scale linearly. The time taken to connect per user/socket will increase as the number of sockets/connections we have to connect to increases.

Experimentation: We will test this by programmatically looping over the "create a new socket" code resulting in users being rapidly added until it reaches the upper limit, which we set for it. We achieved this with a while loop which takes in a parameter of i<**X** where **X** is our independent variable (**Number of connections**) and number of new sockets to be created. **The code and laptop** the testing is being conducted on remain unchanged in our experiment and are thus our control variables. Our dependant variables are **real time**, **user time**, **system time** and **time to connect per user**. Inside the while loop there is a try statement and catch statement. Inside the try statement we create a new socket and increment i until we reach X. We use the Linux "time" function to see how long our program runs for before terminating.

	Time to execute			
Number of connections	Real time	User time	System time	Time to connect per user (microseconds/connection)
1000	0m0.347s	0m0.375s	0m0.125s	347
2000	0m0.676s	0m0.344s	0m0.391s	338
3000	0m1.259s	0m0.297s	0m0.500s	420
6000	0m2.341s	0m0.516s	0m0.969s	390
9000	0m3.963s	0m1.188s	0m1.500s	440
12000	0m3.963s	0m1.750s	0m9.250s	1100
24000	1m31.464s	0m3.656s	1m15.516s	3811

*Formula is Real time/Number of connections (X)

Conducted on a laptop with an Intel Core i7-8550U CPU @ 1.80GHz 2.00 GHz and 18GB RAM.

Results: We can conclude that the time per connection up until 9000 connections stays roughly the same, at an average of 387 microseconds, however steadily increasing as number of connections increases. After that, our time to connect per connection drastically increases and does so nearly fourfold by the time we reach 24000 connections. This shows that as the number of sockets open increases the time taken to make new connections increases, gradually at first but eventually rather steep. We therefore accept our hypothesis.

Experiment 2

Question: Is there a max number of users that the server will accept before it crashes? How does this compare between two computers, where one has better hardware than the other?

Hypothesis: There is a maximum number of users that can connect to the server before the program crashes, and the number of connections accepted until failure will be significantly more on the computer with better hardware.

Experimentation: Our dependent variable will be the code. The independent variable will be the two laptops. We will test this by running code that loops over the create a new socket code resulting in users being rapidly added until the program crashes. We achieved this with a while loop which takes in a parameter of true. Inside the while loop is a try statement and a catch statement. Inside the try statement we create a new socket and increment connection count. In the catch statement we print out the connection count managed before the program crashes.

Laptop 1 has an Intel Core i7-8550U CPU @ 1.80GHz 2.00 GHz and 18GB RAM.

Laptop 2 has an Intel Core i5-8250 CPU @1.60GHz 1.8GHz and 12GB RAM.

Test number	**Laptop 1's number of connections accepted until failure**	**Laptop 2's number of connections accepted until failure**
1	29732	5182
2	29461	7784
3	29635	810
4	29545	8134
5	29602	2486
6	29446	3382
Average result	29570	4629
Standard deviation	108.97	2685.67

Results: We can conclude that there is a max number of clients for which the server can accept before crashing. We can also draw from our data that the laptop with the better hardware is able to handle more connections. The higher-end laptop also had a smaller standard deviation and so seems to run the program with more stability. Therefore, we accept our hypothesis.

Experiment 3

Question: Will the server still be stable upon any client termination, and will clients be stable upon server termination?

Hypothesis: The server will not be disrupted upon any client termination and clients will not be disrupted upon server termination.

Experimentation: I will run an instance of the Server class and 3 instances of the Client class each connected to the server with unique usernames. I will then terminate one client by using the '/quit' command which was programmed to be the stable way to terminate a client since it closes the input and output streams as well as the socket. If there is no disruption on the server side, i.e., no crashing and errors displayed on the terminal, I will then move on to terminating the next client and repeating the process until all 3 clients have been terminated without any disruption to the server. Finally, I will have the same setup, i.e., an instance of the Server class and 3 instances of the Client class each connected to the server with unique usernames. I will then terminate the server and determine whether either of the clients have been disrupted, i.e., crash with errors displayed on the terminal.

Results: The initial client termination caused no server disruption. The next two client terminations produced the same result. I then tested the second phase of experimentation, where I terminated the server. Neither of the clients were disrupted upon the termination of the server. From these results, I can accept my hypothesis and conclude that the server and client are stable.

Issues encountered

At first, we experienced some concurrency issues with the server, when multiple users connected to the server at once. These were quickly solved using java concurrency structures. The major issues after that were to do with client and server disconnection. These issues were centred around clients or the server crashing on errors when the other disconnected. These were all solved by the final submission however.

Design

I designed the Client GUI in such a way that the top portion of the window is for the user to connect to the server with a username, host address and port; and the bottom portion of the window is where the user can send and receive messages once he/she has connected to the server.

I designed the program in such a way that the client can enter the following commands into the message box in order to send to the server and perform the corresponding tasks:

1) "/users" displays the connected users to the client that called it and to the terminal of the server.
2) "/name <newName>" changes the users' username (as long as it is unique) and broadcasts the change.
3) "/whisper <Recipient> <Message>" sends a private message to a specified user.
4) "/quit" triggers the shutdown function which closes all input and output streams as well as all socket connections.

Compilation and execution

Since this project was built using the Apache Netbeans 14 IDE, it cannot be compiled and run from the command-line/terminal. The project must be first opened and then compiled and ran from within the Apache Netbeans IDE. To do so, follow these steps:

1) You must have the Apache Netbeans 14 IDE installed on your computer.
2) Open the IDE.
3) Move your cursor to the top left of the application and click on "File" and then "Open Project" (or use the keyboard shortcut Ctrl+Shift+O).
4) Navigate to the directory of your desired project that you want to open.
5) Select the project folder that contains the folders generated by Netbeans upon creating a project, namely "nbproject" and "src". In this case the project folder is called "Chatroom" and it should have a little coffee-cup-on-a-saucer icon to the left of it indicating that it is a Java project. Once selected click on "Open Project" at the bottom right of the window.
6) To run the project, first click on the Server.java file to open it, click on any line of code in the file (so that Netbeans knows that this file is the file you want to run), then either left click to open a drop-down list and click on "Run File" or use the keyboard shortcut Shift+F6. Now the built-in terminal will be displayed and you should see the text, "Server running on port 4000".
7) Now to run a client, first click on the Client.java file to open it, click on any line of code in the file, then either left click to open a drop-down list and click on "Run File" or use the keyboard shortcut Shift+F6. Then a GUI window for the client should open.

Libraries used

- java.io.DataInputStream;
- java.io.DataOutputStream;
- java.io.IOException;
- java.net.ServerSocket;
- java.net.Socket;
- java.util.ArrayList;
- java.util.concurrent.ExecutorService;
- java.util.concurrent.Executors;