Enrique Posada Lozano
A01700711
ITESM Campus Qro

Multiple User Chat - App Report

Introduction:

Information is key in today's society. Whether it is through money transferences, messages, posts, or others, the interchange of data is what pushes today's society forward. That said, the rise of the internet has caused communication to expand beyond what's known, allowing communication from any part of the world. Although we now live in communicated society, the application developed for this project serves the purpose of understanding what's beneath the communication we use in our daily lives. However, "typical programming" is sequential, where a program follows each line of code written one by one, in order. So, in order to achieve something like a chat style communication between multiple users a Client-Server model is needed with the use of concurrency in order for to have processes that allow reading and writing between multiple users in a network. This project originally started as just a console application that guaranteed communication within a local network with local connected users, but now its purpose is to make it for the user, focusing on the visuals while maintaining a working backend with concurrency, as well as improving its functionalities.

Objective:

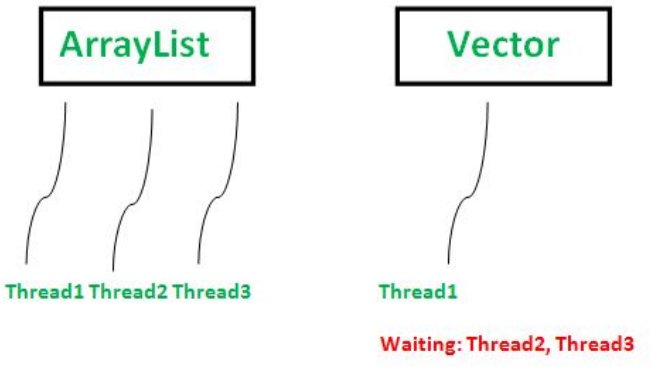
The objective of the project is to use concurrency through the implementation of a Client-Server application focused on a multi-user chat within a local network. In order to achieve communication in a network, sockets are required in order to establish communication between a server and client. The project makes use of sockets, which are endpoints between two or more computers.

Concurrency (Programming Paradigm):

Before we dive into concurrency, a brief introduction to the paradigm is adequate for anyone who reads this report. Concurrency is a programming paradigm that focuses on the ability of having multiple computations happening at the same time, but not necessarily simultaneously (parallel programming does this). Concurrency is everywhere nowadays, it enriches applications and allows doing many things at the same time. We can see this in mobile apps with processing information on servers, websites with multiple users and Graphical User Interfaces (*GUI*) for example video-games, among other usages. One of the models of concurrency is the *message-passing* model, which we'll focus on and explain in the next section.
On the server-side, the program works by listening for client connections and for each client that connects, a thread is created and assigned for each client, which are then stored in a Vector instead of an arraylist because they provide synchronization and thus, thread safety.

Enrique Posada Lozano
A01700711
ITESM Campus Qro

In the server, each client thread calls methods on the server such as broadcasting-messages, sending private messages and disconnecting, so a vector synchronizes the clients in order to avoid race conditions and have each perform an operation at a time. Plus, each method is synchronized in order to control when two or more threads want to either broadcast a message, send a private message or disconnect.

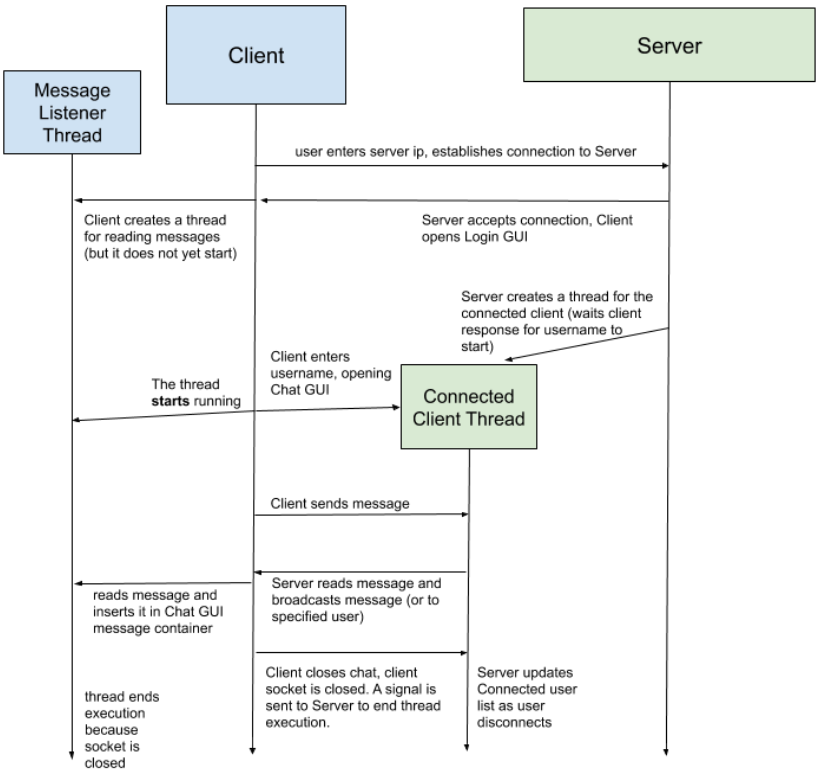


(ArrayList vs. Vector **Credit** : *GeeksforGeeks*)

On the client-side, basically what happens concurrent-wise is that a client runs a thread that listens for its socket input stream, and while the main program of the client just acts as the “back-end” of the client (being the one that holds methods for calling GUIs, sending messages to the server and handling disconnection from the server), the thread created listens for all messages received from the Server including the messages that the user sends to the Server. This last one is made as a way to confirm that the server does receive the clients messages and therefore, when a message sent by the own user appears on screen, it means the message was delivered to the server successfully. The previous implementation made use of two threads, one thread for writing and another for reading but now writing is event-driven, meaning that each time the user presses the send button (or closes the program), a signal/message is sent to the server indicating the desired operation to perform.

The image presented below illustrates the flow of the program between one client and the server. If a client enters the servers local ip address, the server accepts the connection, sending a response to the client requesting a username with the Login GUI, upon which the client sends its username, opens up the Chat GUI and has the message listener thread running for messages from the server. On the other hand, when the client sends its username the server creates a thread for that client and adds it to the pool (vector) of connected clients, where the thread listens for the messages sent by that user. At this point, the client can send message to everybody in the chat room, private message and can disconnect from the server. If the client disconnects, a signal is sent to the server indicating that the socket has been closed, and therefore the server updates its connected user list and closes the socket of the client without affecting other users.

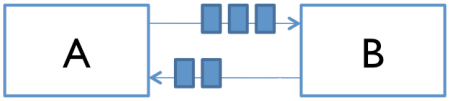
Enrique Posada Lozano
A01700711
ITESM Campus Qro



Model Design:

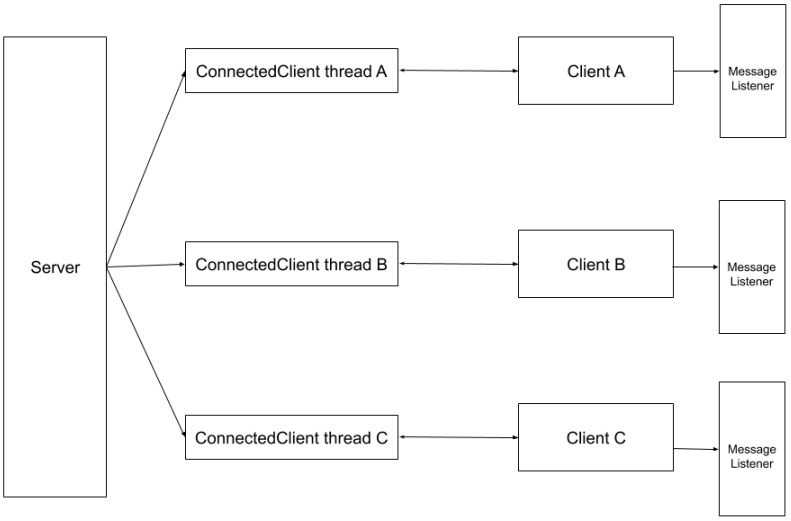
As previously mentioned, for this project a message passing model is used for communication between clients. A message passing model has threads interacting with each other by sending messages through a communication channel (with sockets in this case). This can be used for two computers in a network, a chat client-server or a server sending a webpage to a user that sends a request. The following image represents the model, which in our case A can represent the chat server and B the client.

(Message Passing model **Credit** : MIT)



As it can be seen in the image below, the design of this application consists of a multi-threaded server that listens for connections and assigns a thread for each connection (user) received. From there, each client has a connection to the server and are able of sending messages between each other, having each connected client make use of methods

in the server in order for the server to distribute messages in either a one-to-one message or a broadcast message to all connected users.



Results and Observations:

As mentioned before, this project acts as an improved version of a chat app developed first in console. In the first implementation, a client would have two threads, one that would listen for messages and one that would take care of scanning user input and sending it to the server. This has now changed to have a thread that listens just as before, but writing is now event-driven. Implementing a GUI allowed the implementation of things such text labels or text-areas and so in order for each message to be sent, a button needs to clicked so instead of scanning lines, if the user input is not empty, then when the send button event is triggered then the content is sent out to the server.

Another thing was that the previous implementation made use of array-lists, however this was changed to vectors in order to synchronize threads, meaning that one thread can access a vectors methods at a time, providing thread-safety (no-race conditions).

One of the main problems with this new implementation was having a GUI handling the message listener thread during the login. When a client logged in, if the user disconnected then the thread would run into errors because its input stream could not read a message due to having its socket close and not being interrupted, crashing the server. Therefore, the thread starts running right after the client provides a username, since both the Server's thread for the client and the message listener of the client will start as the user enters the chat-room itself.

However the main problem with this version was the input stream (as just mentioned), not only at login but many times it would be unable to hear messages because a user

disconnected and the socket would not adequately be closed. This was all solved by following the message passing model where one would signal the other, meaning in this case that when a client disconnects, a message is passed to the connected client thread, which knows its the exit signal and then calls a method for handling disconnection, closing its socket and removing the user from connected user list.

An important thing to note in this project was the misconception of static variables in a multithreaded program. I originally had the idea that static variables where not affected since each thread would mind its own, but using static variables also affects all threads, meaning not using them allows them to work with each instance/thread of the server. This problem originally made all of my users have the same values, so not using static variables in thew connected client class solved this issue.

Another problem was wanting the application to be compiled by the user in order to work, this was fixed by making a version with the java version in which the project was developed and another for java RunTime version 1.8 and using jars instead of the compiled version of the code.

Last but not least, not using synchronized methods for disconnecting clients just allows them to have race conditions for disconnecting or sending messages, which would all crash the server.

At the end the program was successfully tested with various users in both a virtual machine as well as the local machine since there was not another available computer that I could make use of to test the program on.

Program Usage:

In order to run either the Server or Client, a minimum version of Java RunTime 1.8 or newer is needed for it to run. The program has two executables, one for the Server and another for the Client in a jar format. On a console open either the Client or Server jar, the Server when running will handle connections and users. Meanwhile, for the client you will be greeted with a welcome message and a request for the server's local ip address in order to connect. If a wrong ip is entered or the server has not been run yet, it will let you know. When a connection to the server has been established, a login interface is presented to the user and upon entering a username , redirects to the chat room where conversations can be initiated. Inside the chat-room, in order to send a private message from one user to another connected in the chat-room, the user must enter the character ‘@’ followed by the username, a space and then the message to send to the user. From there, the server will either send the message and if it fails then the server sends another response where it failed to find the username provided in the list of connected users. When a user closes the window, that user sends a signal to the server to disconnect and the server updates the list of connected users and removes the user.

Enrique Posada Lozano
A01700711
ITESM Campus Qro

Conclusions:

As you've seen by now, concurrency opens up new opportunities for the development of new applications, and in this case communication between different people. The same principle is applied for example to videogames, since communication between a server and client allow things like multiplayer to happen, among other things. Furthermore, more improvements can be made in this project as now that connectivity is handles between computers and a GUI is implemented, there are endless areas of opportunity such as having clients maybe start a game between one another, encrypting messages sent between clients and modifying the interface so it can have colors and maybe be able of sending images or files at one point.

References:

- MIT. (n.d.). Reading 17: Concurrency. Retrieved from https://web.mit.edu/6.005/www/fa14/classes/17-concurrency/
- GeeksforGeeks. (2018, December 11). Vector vs ArrayList in Java. Retrieved from https://www.geeksforgeeks.org/vector-vs-arraylist-java/