ITESM CAMPUS QUERÉTARO

Chat Server with Java Concurrency

Carlos Ayala Medina A01703682

Programming Languages Final Project

August – December 2021

Abstract

A chat server refers to a network-based application that allows real time transmission of messages between users. In this document we showcase an approach to develop this application using concurrency with Java, and the concepts needed to understand the solution and why it can bring value.

Context

The first chat system was developed in 1973, by Doug Brown and David R. Woolley. It was called "Talkomatic". It allowed up for up to five people to communicate. The way it worked was that every time a person typed a character, it will appear on all users' screens. [1]

This concept has evolved as the need for instant communication has grown, to the point that currently we have chat applications like WhatsApp, LINE and Telegram, that allow us to communicate easily with only our smartphones and an internet connection.

But even in this era of fast communication, there are places where having a solid internet connection is just impossible. In 2020 according to the INEGI [2], 1 in 3 Mexicans don't have internet access, but not because they don't need it, in fact most of the people without internet are in rural areas where the service is not available.

Solution

Here is the need of a local chat server, to be able to communicate while being on the same network even without an internet connection. Even with its limitations, a local server can allow people in the same building to chat between them. This can bring value to businesses in rural areas, as well as every place where a reliable internet connection can be hard to get.

First, let's start defining exactly can this application do, and what do we need to know to make it. This local chat server can communicate between two or more users that are connected on the same network. For example, WhatsApp, an app we all use, the only difference it`s that all users devices must be in the same network, it doesn't matter if there is no internet connection.

Now, to make this possible we use some things called **Sockets**. To understand the makings of the solution, we must know what a **socket** is.

Sockets

A **socket** is one endpoint of a two-way communication link between two programs running on the network.

With this, we are going to start to explain the logic of the application. Socket classes are used to represent the connection between a client program and a server program. So, to make a chat server that can have multiple users, we are going to have a server that will receive a message from one client, then the server will return the same message to all clients that are connected.

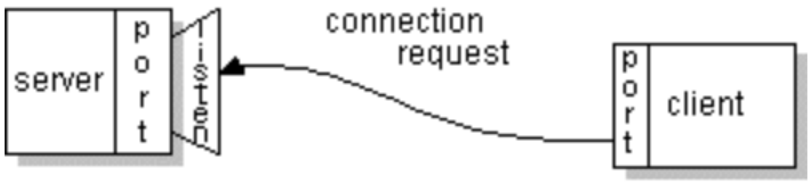

Figure 1: Example of Server and Client communication

Sockets work with Ports. This is the way they identify where the data is destined to go. In Figure 1, we can see that the server is always listening, which is what makes possible that a client can communicate. Every client will send a connection request to the server, if the ports match and everything works correctly, the connection will be made.

Concurrency

Now that we know what a socket is, and how we are going to use them to develop this application. We can start talking about the paradigm we are going to use, **concurrency**

Concurrency in this context, is the ability to have multiple computations running at the same time. For example, your computer can open multiple applications at the same time.

This does not mean that your computer is doing all the tasks at the same time. On the contrary, with concurrency what happens is that your CPU is making progress of one task at a time. But to make progress on multiple tasks to give the impression of running things at the same time, the CPU switches between tasks during their execution.

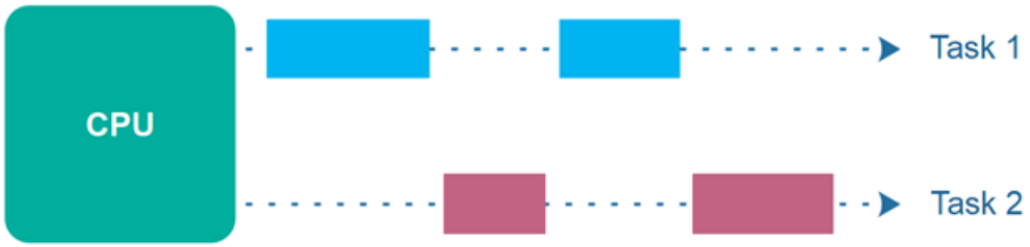

Figure 2: Example of concurrency

As we can see in Figure 2, the CPU appears to be running multiple tasks at the same time, but in fact is switching between them, we just don't notice because it's so fast.

In the application we can think of every **thread** as a task, a **thread** is a segment in the execution in a program, in Java we can have multiple **threads** running concurrently.

Now that we understand the relevant information, we can finally start with the development of the application.

We are going to start by dividing the application in two parts, the server side, and the client side.

Server side

Server.java

In the server side we have two java classes, Server.java and ClientController.java. Each one has its own purpose in the application, first we are going to see what Server.java has and what it does.

```
public class Server {

    public Server(int port) throws IOException {

        JFrame frame = new JFrame();
        JPanel panel = new JPanel();

        panel.setBorder(new EmptyBorder(5, 5, 5, 5));
        panel.setLayout(new BorderLayout(0,0));

        frame.add(panel,BorderLayout.CENTER);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setTitle("Server");
        frame.setBounds(100, 100, 450, 300);

        JTextArea jTextArea = new JTextArea();
        DefaultCaret caret = (DefaultCaret)jTextArea.getCaret();
        caret.setUpdatePolicy(DefaultCaret.ALWAYS_UPDATE);
        jTextArea.setEnabled(false);
        JScrollPane sp = new JScrollPane(jTextArea);

        frame.add( sp );


        frame.setPreferredSize(new Dimension(400, 300));
        frame.pack();
        frame.setVisible(true);

        ServerSocket server;
        server = new ServerSocket(port);
       try {

            while(true) {
                jTextArea.append( "Waiting connections\n" );
                Socket socket = server.accept();
                jTextArea.append( "New user connected\n" );
                ClientController clientC = new ClientController(socket);
                Thread thread = new Thread(clientC);
                thread.start();
            }
       } catch (IOException e) {
           // TODO Auto-generated catch block
           server.close();
       }
    }

   public static void main(String[] args) throws IOException {
       int port = 9000;
       new Server(port);
   }

}
```

Figure 3: Server.java

GUI

The visual interface we have in the class Server.java is thanks to the library swing in Java.

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();

panel.setBorder(new EmptyBorder(5, 5, 5, 5));
panel.setLayout(new BorderLayout(0,0));

frame.add(panel,BorderLayout.CENTER);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setTitle("Server");
frame.setBounds(100, 100, 450, 300);

JTextArea jTextArea = new JTextArea();
DefaultCaret caret = (DefaultCaret)jTextArea.getCaret();
caret.setUpdatePolicy(DefaultCaret.ALWAYS_UPDATE);
jTextArea.setEnabled(false);
JScrollPane sp = new JScrollPane(jTextArea);

frame.add( sp );


frame.setPreferredSize(new Dimension(400, 300));
frame.pack();
frame.setVisible(true);
```

Figure 4: GUI part in Server.java

Here is the exact code utilized for the GUI, since this part is only for visual purposes, there is no need to have it the same way. But if you are interested, this code was made to replicate in some way a terminal. This with the idea of seeing the status of the server and its users.

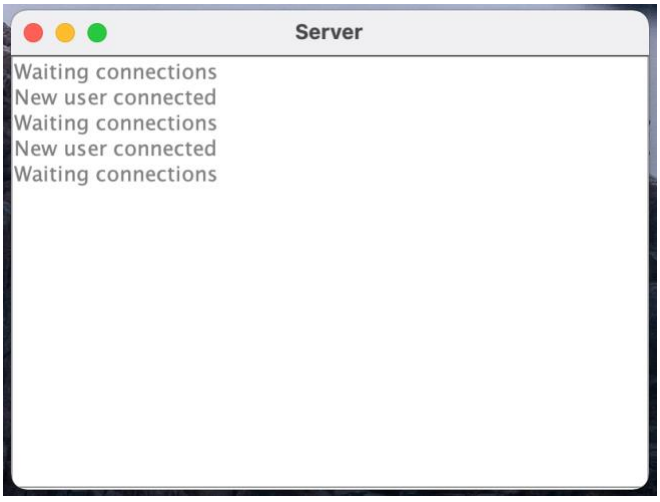

Figure 5: Server interface

As we can see in Figure 5, there is no way to interact with the interface. Its only purpose is to tell us if the client is running correctly. It will tell us if a new user is connected, and if everything is okay, it will tell us that is waiting for new connections, as that is its idle state.

Of course, there is really no need for having a GUI in the server side, we can print the status on the regular terminal and be done with it.

Server

Now we are getting to the part where the server happens.

```
 ServerSocket server;
 server = new ServerSocket(port);
try {

     while(true) {
         jTextArea.append( "Waiting connections\n" );
         Socket socket = server.accept();
         jTextArea.append( "New user connected\n" );
         ClientController clientC = new ClientController(socket);
         Thread thread = new Thread(clientC);
         thread.start();
     }
} catch (IOException e) {
    // TODO Auto-generated catch block
    server.close();
}
```

Figure 6: Server Socket part in Server.java

In Figure 6, we can see the creation of the ServerSocket, a class part of the java.net package, that implements a socket that servers can use to listen for and accept connections to clients.

After creating the ServerSocket with its respective port, we have an infinite while. Inside this loop we have an append to the JTextArea in the GUI, this is to print text inside the interface. After that we have a new socket, this will be the socket of a client once it sends his request.

This works with the line of code “server.accept()”, this method from the class ServerSocket listens for a connection and if everything is working correctly, it will accept it.

When the server accepts a connection, it will print that a new user connected on the interface, then we are going to create a new object from the class ClientController, we create it with the client socket. After that we can see the first part where we use concurrency, we made a new thread with the client controller object and let it run apart from our main thread.

```java
public static void main(String[] args) throws IOException {
    int port = 9000;
    new Server(port);
}
```

Figure 7: Main of Server.java

In Figure 7 we have the main, here we only have the port we are going to use for our server, we can change to almost any number, but it's recommended to use higher numbers in case other ports are being used for something else.

ClientController.java

Now that we finished explaining Server.java, we can go to the class ClientController. This class purpose it's to manage a list of all clients that connect to the server. In this class we have the method that makes possible sending messages to all other users in the application.

```java
public class ClientController implements Runnable {

    private String username;
    private Socket client;
    private BufferedReader reader;
    private BufferedWriter writer;
    public static ArrayList<ClientController> list = new ArrayList<>();

    public ClientController(Socket client) {
        // TODO Auto-generated constructor stub
        this.client=client;
        try {
            this.reader= new BufferedReader(new InputStreamReader(client.getInputStream()));
            this.writer= new BufferedWriter(new OutputStreamWriter(client.getOutputStream()));
            this.username = reader.readLine();
            list.add(this);
            send(username + " has connected");
        } catch (IOException e) {
            // TODO Auto-generated catch block
            disconnect(client, reader, writer);
        }
    }

    public void send(String msg) {
        for (int i=0;i<list.size();i++){
            if (list.get(i).username != username){
                try {
                    list.get(i).writer.write(msg);
                    list.get(i).writer.newLine();
                    list.get(i).writer.flush();
                } catch (IOException e) {
                    // TODO Auto-generated catch block
                    disconnect(client, reader, writer);
                }

            }
         }
    }
```

Figure 8: ClientController.java constructor and send method

We are going to start by explaining the constructor, besides having the client socket, we also create a BufferedReader and BufferedWriter. We can see that inside them we create an InputStreamReader and OutputStreamWriter with the get methods from the input and output streams. This is because the get methods return bytes instead of the text, so we can think of this as a conversion to a string.

Then we utilize the method readLine () that will get the first line where the username is, after that we add this to the Array List that is going to work as a list of all clients connected to the server. With that ready we use the method send () to send the message to all users that someone has connected.

Now we are going to see how the send () method works. We can see that it receives a String, which is the message that is going to be broadcasted to all clients. Then we see a for loop that goes through the list of clients, inside the loop there is an if to avoid sending the message to the one that send it. After checking which user not to send the message, we call three methods form the BufferedWriter class, on every client. First the write (String) method where we put the string with the message. Second, we have the newline (), to create al line separator between this and the next message. And lastly, the flush () method. This method is used to compel the bytes of buffered output to be written out to the main output stream.

```
public void disconnect(Socket socket,BufferedReader reader, BufferedWriter writer) {
    // TODO Auto-generated method stub

    try {
        writer.close();
        reader.close();
        socket.close();
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

}

@Override
public void run() {
    // TODO Auto-generated method stub
    String msg;
    while(client.isConnected()) {
        try {
            msg=reader.readLine();
            send(msg);
        } catch (IOException e) {
            // TODO Auto-generated catch block
            disconnect(client, reader, writer);
        }

    }
```

Figure 9: ClientController.java disconnect and run method

Now as you probably noticed, on every catch we have in this class we call for the method disconnect (). In Figure 9 we can see that this method closes the Buffered Reader and Writer, as well as the Socket of the client. The theory is simple, if something happens, then close everything.

Then as you remember, in the Server class we make a thread with an object of the Client Controller class. So, Client Controller must implement Runnable, which comes with the method run (). We can think of this as the main of the thread. Here we are going to have a while loop that works only if the client socket is connected. And every time the Buffered Reader gets something from the

getInputStream () method in the client socket, we will call for the send () method to broadcast the message.

With this we finished the server side of the application.

Client side

In the client side of the application, we also have two classes, Window.java and Client.java.

Window.java

The Window class is the GUI where we can write our messages and receive them. This class also creates a new thread from a client object.

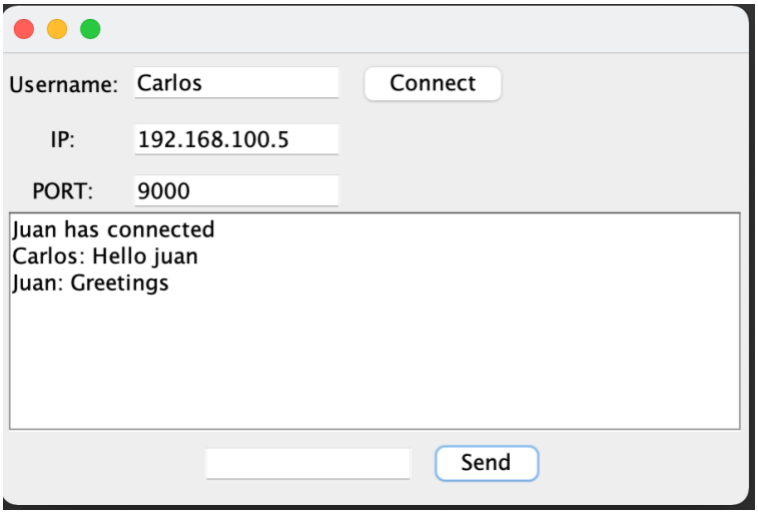

Figure 10: Window.java interface

We can see in Figure 10, the way the GUI looks. The user first must write a username and if the IP and Port are matching with the ones on the server then the user just must press “Connect”.

If other people connect while you are connected, then a message like the one in Figure 10 will appear. “Juan has connected”, we also can write a message and click “Send”.

This GUI is made the same way as the server one. So, we are not going to pay much attention to that since the code is long and not necessary to build the chat application. If you are interested in seeing exactly how it was made, the repository has all the code.

```
button.addActionListener(new ActionListener() {

    @Override
    public void actionPerformed(ActionEvent e) {
        if(textField.getText() != "" && textField_1.getText() != "" && textField_2.getText() != "") {
            username = textField.getText();
            ip = textField_1.getText();
            port = textField_2.getText();
            int aux = Integer.parseInt(port);
            runClient(aux);
        }

    }
});
```

Figure 11: ActionListener for the connect button

Here we are going to focus on the indispensable parts to make the chat run. In Figure 11 we can see the action listener method of the button "Connect". Here we take the values from the interface that the user can modify, then we call for the method run Client ().

```
public void runClient(int port) {
    Window wn = this;

    new Thread(new Runnable() {
        Socket socket;
        @Override
        public void run() {
            // TODO Auto-generated method stub
            while(true) {
                try {
                    socket = new Socket(ip, port);
                    //socket = new Socket("localhost", 9000);
                    Client client = new Client(username, socket, wn);
                    client.receive();
                    client.sendMsg();
                } catch (IOException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }

            }

        }

    }).start();

}
```

Figure 12: run Client() method

```
public static void main(String[] args) {
    EventQueue.invokeLater(new Runnable() {
        public void run() {
            try {
                Window frame = new Window();
                frame.setVisible(true);

            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    });
}
```

Figure 13: Main of Window.java

In Figure 12 we see the method run client (). First, we see that we make a Window object of this instance. Then we create a new thread for the client. This is another time we use concurrency in the application. To understand better why we need another thread, we must look at Figure 13. There is the main of Window.java, we can see that is a different main than usual, since here we are creating a new thread. That is because for the GUI to be interactable we need to run it as a thread.

So, we have the thread of the window, and every time we make a connection to the server, we are going to create another thread with a client. This makes possible to close the client but not the window GUI and make another connection with the same window without having to run a new one. We are not going to implement that option since it's not necessary but thinking about it help us to understand how the threads are working.

In Figure 12 we can see an infinite while loop, where we create a new socket with the values we get from the interface. Then we create a client with the username, the socket, and the window we just created at the start of this method. After that we call two methods from the Client class, receive () and sendMsg (). Their names are self-explanatory, but we are going to see how they work on the Client class.

```
    btnNewButton.addActionListener(new ActionListener() {

        @Override
        public void actionPerformed(ActionEvent e) {
            msg = textField_3.getText();
            textField_3.setText("");

        }
    });
}

public String getMsg() {
    return msg;
}

public void setMsg(String msg) {
    this.msg = msg;
}
```

Figure 14: ActionListener for Send button, set and get method for msg

```
volatile static String msg;
```

Figure 15: Volatile variable msg

Before explaining the Client class, let's see Figure 14. Here we have what happens if we press the send button. There is nothing special, we just get the message, save it on the variable msg and then clear the box where we write it.

The important thing here is that as we can see in Figure 15, the variable msg is volatile. This mean that it can be accessed and modified by multiple threads. Using this type of variable help us to

reduce the risk of memory consistency errors, because any write to a volatile variable establishes a happens-before relationship with subsequent reads of the same variable.

Thanks to this, we can modify the msg variable between the Window thread (GUI) and the Client thread. That is also the reason why we have a setter and getter method for this variable.

Client.java

```
public class Client {

    public String username;
    private BufferedReader reader;
    private BufferedWriter writer;
    private Socket socket;
    public Window wn;

    public Client(String username, Socket socket, Window wn) {

        this.wn=wn;

        this.username=username;
        this.socket=socket;
        try {
            this.reader=new BufferedReader(new InputStreamReader(socket.getInputStream()));
            this.writer=new BufferedWriter(new OutputStreamWriter(socket.getOutputStream()));
        } catch (IOException e) {
            // TODO Auto-generated catch block
            disconnect(socket, reader, writer);
        }
    }

    public void sendMsg() {
        // TODO Auto-generated method stub
        try {
            writer.write(username);
            writer.newLine();
            writer.flush();
            while(socket.isConnected()) {
                if(wn.getMsg() != null) {
                    String msg = wn.getMsg();
                    writer.write(username + ": " +msg);
                    writer.newLine();
                    writer.flush();
                    wn.textArea_4.append(username+": "+msg+"\n");
                    wn.setMsg(null);
                }

            }
        } catch (IOException e) {
            // TODO Auto-generated catch block
            disconnect(socket, reader, writer);
        }


    }
```

Figure 16: Client.java, constructor and sendMsg method

Now we are finally on the last class of the application, the Client class. In Figure 16 we can see the constructor that receives the values we get from the user on the interface. Same as with the Client Controller class, we have a Buffered Reader and Buffered Writer, almost everything in the constructor is the same except for the Window object that we are going to use to manipulate the volatile variable msg.

The sendMsg () method works this way. First, we write the username with the BufferedWriter, we also use the new Line () and flush () method to send it correctly. This is for the Client Controller

class to know which user send the message. After that we have a while loop that works if the socket is connected. Here we check if the variable msg is empty, if not we do the same process to send a message with the BufferedWriter, then we append the message on the GUI, so we don't have to get it back from the server. Finally, we set msg as null. Of course, this is only possible because msg is a volatile variable.

```
public void disconnect(Socket socket,BufferedReader reader, BufferedWriter writer) {
    // TODO Auto-generated method stub

    try {
        writer.close();
        reader.close();
        socket.close();
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

}

public void receive() {
    // TODO Auto-generated method stub
    new Thread(new Runnable() {

        @Override
        public void run() {
            // TODO Auto-generated method stub
            String msg;
            while(socket.isConnected()) {
                try {
                    msg=reader.readLine();
                    wn.textArea_4.append(msg+"\n");
                } catch (IOException e) {
                    // TODO Auto-generated catch block
                    disconnect(socket, reader, writer);
                }

            }

        }

    }).start();

}
```

Figure 17: Client.java, disconnect and receive method

In Figure 17 we can see a familiar method, disconnect (). It the same as the one in Client Controller, we close everything if something happens. Then we have the receive () method, one that also represents the paradigm of concurrency.

Here we create a new thread that if the socket is connected, will listen for new messages. Thanks to the readLine () method, only when a message appears the method will continue. Then same as in the sendMsg () method, we append the new messages on the GUI.

Results

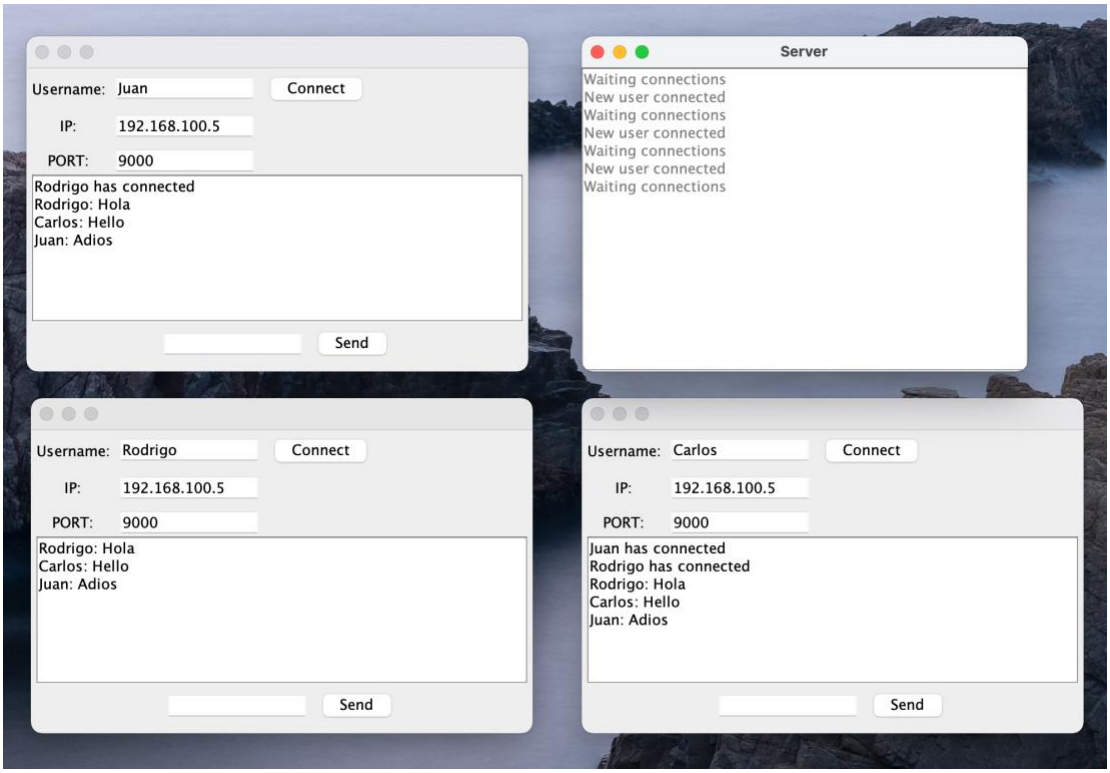

Figure 18: Test 1

After the development of this application, we can test how it works. In Figure 18 we can see three instances of Window connecting to the server and communicating between them.

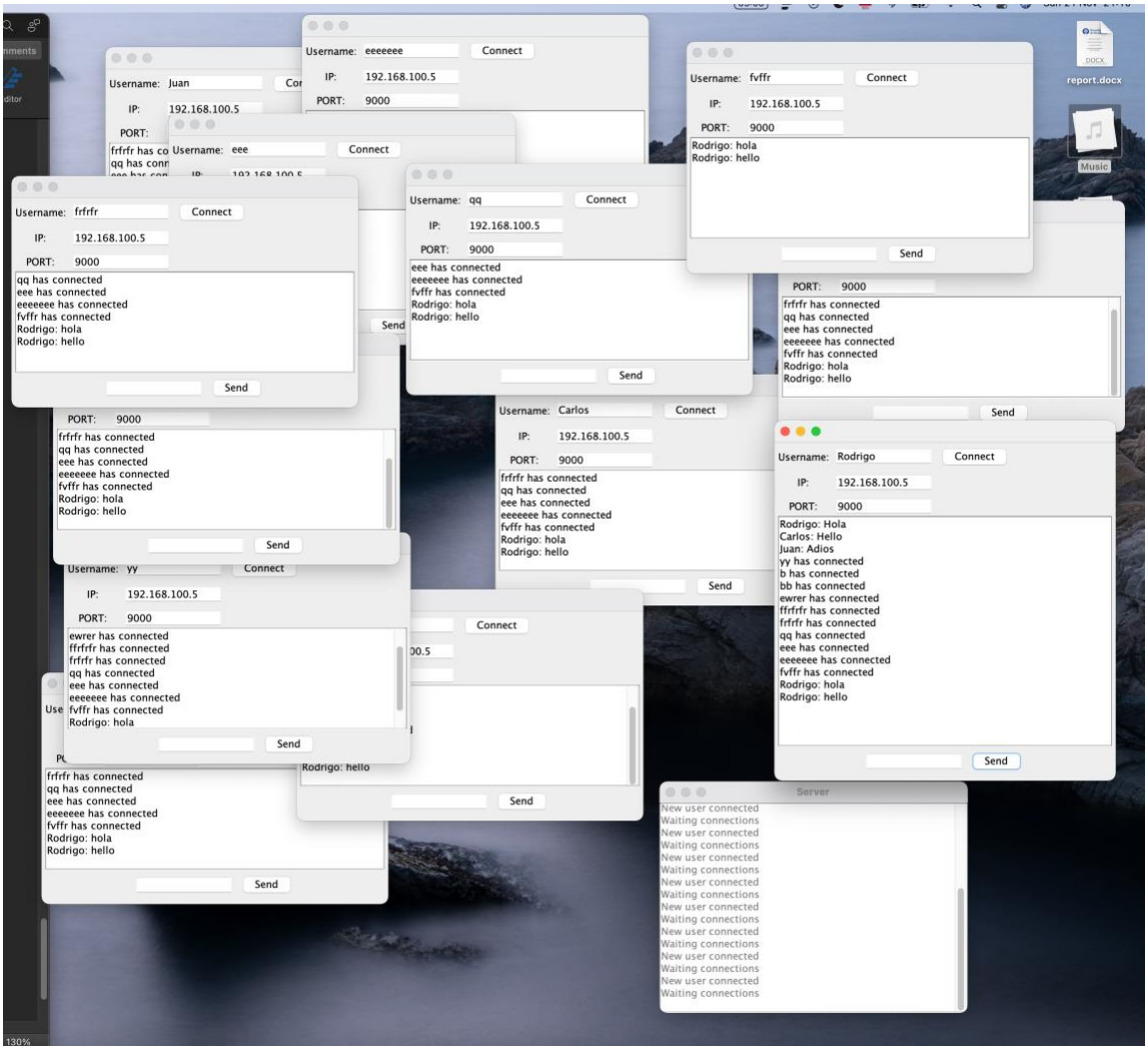

Figure 19: Test 2, 15 users

In Figure 15, we test the application with 15 instances of Window, it also works correctly.

```
public static void main(String[] args) throws IOException {
    int port = 9000;
    new Server(port);
}
```

Figure 20: Where to change the port

```java
public void runClient(int port) {
    Window wn = this;

    new Thread(new Runnable() {
        Socket socket;
        @Override
        public void run() {
            // TODO Auto-generated method stub
            while(true) {
                try {
                    socket = new Socket(ip, port);
                    //socket = new Socket("localhost", 9000);
                    Client client = new Client(username, socket, wn);
                    client.receive();
                    client.sendMsg();
                } catch (IOException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }

            }

        }

    }).start();

}
```

Figure 21: Where to change the ip

To test it we only need to open a server and multiple windows. If you plan to download and test this project, be sure to have a recent version of Eclipse for Java or install Java on your computer. Then to make the test if you want, you can change the port on the server main (Figure 20) and in the GUI for the users. Also, if you are testing all in the same computer you can comment the line where we create the socket with the variables and uncomment the next line where it uses localhost (Figure 21). If you want to test it with different computers in the same network, you must get the IP from the computer where you are running the server and use that one in the GUI for the client.

Conclusion

A chat server has many uses, and while the application can prove to be useful on some situations, I believe that using concurrency to develop this program was the more useful part. This project has helped me to understand more of the paradigm and be aware of how common it is used on other programs. Developing a project while being conscious of the paradigm and the issues that can appear such as memory consistency errors, and finding solutions to these problems, has proved to be an excellent practice to get better at working with a paradigm in mind.

References:

[1] Jones, Steve. "PLATO". Encyclopedia Britannica, 23 Nov. 2015, https://www.britannica.com/topic/PLATO-education-system. Accessed 18 November 2021

[2] Ana Luisa Gutiérrez. (2021). 1 de cada 3 mexicanos no se conectaron a internet en 2020: Inegi. 18/11/2021, de El Financiero Sitio web: https://www.elfinanciero.com.mx/empresas/2021/06/22/en-la-era-de-la-conectividad-1-de-cada-3-mexicanos-no-navegaron-por-internet-en-2020-inegi/

References used for the development of the application:

https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

https://docs.oracle.com/javase/tutorial/essential/concurrency/atomic.html

https://www.geeksforgeeks.org/bufferedoutputstream-flush-method-in-java-with-examples/

https://www.javatpoint.com/java-socket-getinputstream-method

https://www.geeksforgeeks.org/difference-between-concurrency-and-parallelism/

http://tutorials.jenkov.com/java-concurrency/concurrency-vs-parallelism.html

https://docs.oracle.com/javase/tutorial/networking/sockets/definition.html

https://www.ibm.com/docs/en/i/7.2?topic=communications-socket-programming

https://stackoverflow.com/questions/13582395/sharing-a-variable-between-multiple-different-threads

https://stackoverflow.com/questions/5921175/how-to-set-jpanels-width-and-height

https://stackoverflow.com/questions/1052473/scrollbars-in-jtextarea

https://coderanch.com/t/536458/java/Socket-connection-program-works-localhost

http://isp.vsi.ru/library/Java/JExpSol/ch37.htm