TCP CHATTING

CHATSAPP

REPORT

-A Chatting App Through the TCP Protocol-

Made by ***Email***

Enrique Hernández Noguera - ehernan8@uno.edu

INDEX

1. Introduction

During the development of this project, we proceeded to develop a Chatting App following a Client-Server approach using the TCP protocol, through socket programming in the Java programming language.
In order to give an easier use of the app, a GUI was also developed, giving the final user a better experience.
Finally, we gave the final product a funny (that may sound familiar) name, ChatsApp.

2. Features

2.1. Basic Features

The basic features of the project, which would stand for its MVP (Minimal Viable Product) relies on the communication infrastructure of the chatting app.
These features include, as already mentioned, a communication method between users using a client-server approach.
When a client is launched, and effectively joins a chat, it can send messages to the chat, which other connected users can see. This user can also chat some keywords, that will result in different outcomes:

- Bye Message: Whenever the user writes “Bye” into the chat, it will disconnect from it with a proper goodbye message.
- AllUsers Message: Whenever a user writes “AllUsers” into the chat, it will receive a list with all of the actually connected users in the chat, including himself, and the time at which each user joined it (if any users disconnects with a “Bye” message, it won’t appear in future “AllUsers” messages).

From the server’s side, the user that hosts (creates) a chat will have access to it as if it was a log for the chat, being able to see all of the users that joined/left the chat, as well as all of the messages sent to it (the “AllUsers” query message won’t appear, since it is something specific of each user).

2.1. Additional Features

The additional features regarding this project rely on the GUI applied.
These features include:

- Whenever the app is launched, an initial menu pops, which will allow the user to either create a chat or join, without having to execute different files to create/join a chat.
- Depending on the decommission that the user makes to whether to create/join a chat, the corresponding pop-ups will appear, asking the user about the data of the chat they want to create/join.

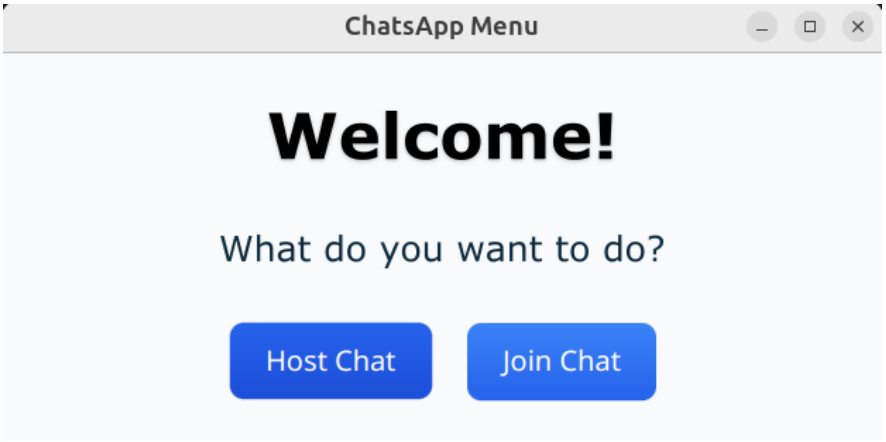
- Both the client and the server will have a chat area, where messages (which will be stylized depending on who sent them) will be deployed.
- On the server's side, there will be a "Close Chat" button, which will end the chat and send a corresponding message to every user connected to it when pressed.
- On the client's side, the structure of a modern chatting app has been developed. This allows the client to have a text box to input messages, as well as a "Send" button to deploy those messages into the chat.
- Whenever on the server side the "Exit Button" is pressed or on the client side a "Bye" message is sent, their corresponding connection to the actual chat will end, bringing them to the initial menu, where they can continue creating/joining new chats.

3. Testing

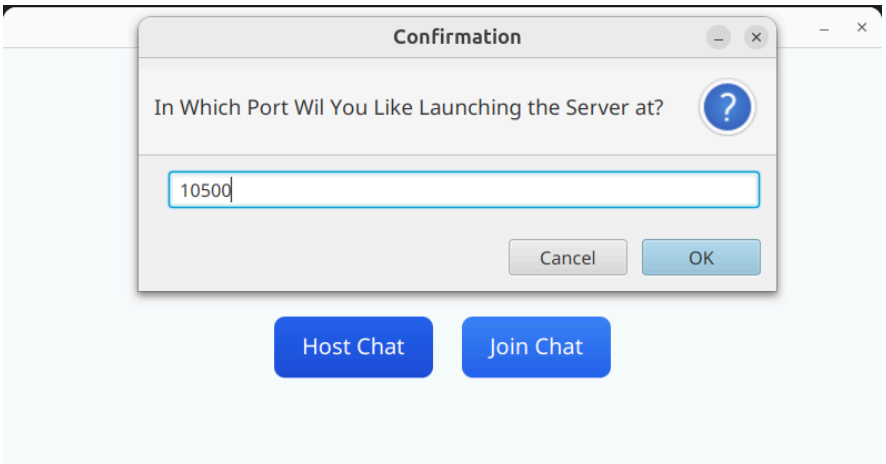
In this section we will show how the previously described features work through a series of explanation messages and screenshots about the app.

3.1. Create Chat

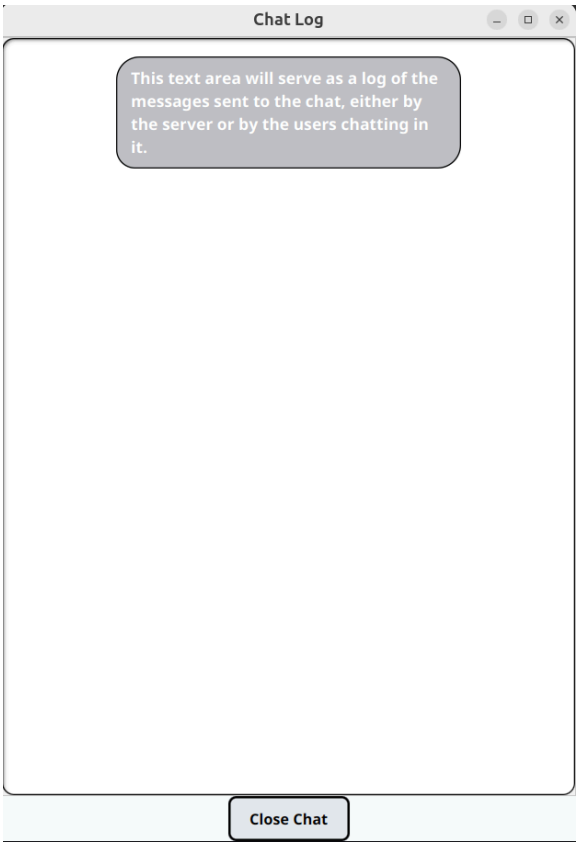
Let's show the steps to create a chat (launch a server).
First, execute the *Menu.java* class, this window will appear:



Now, click on the "Host Chat" button, you will be asked about which port to launch the server (create chat) on:

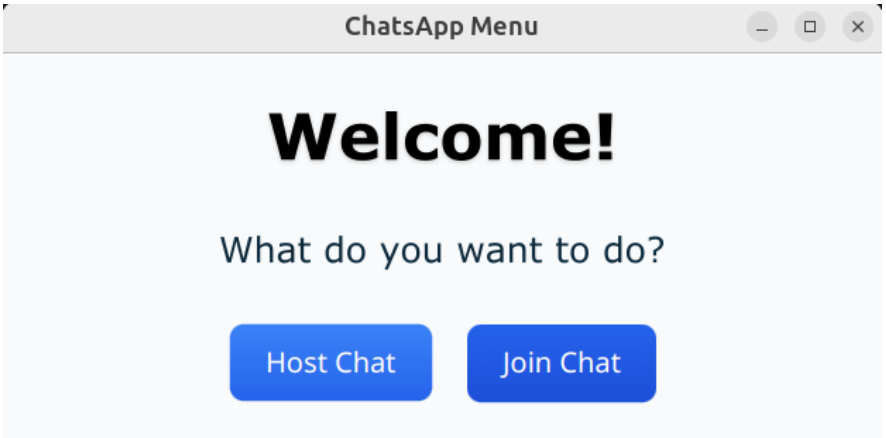


For this example, we will use the port number *10500*.
After pressing “OK”, the chat will be created, and this screen showing the chat’s log will appear:

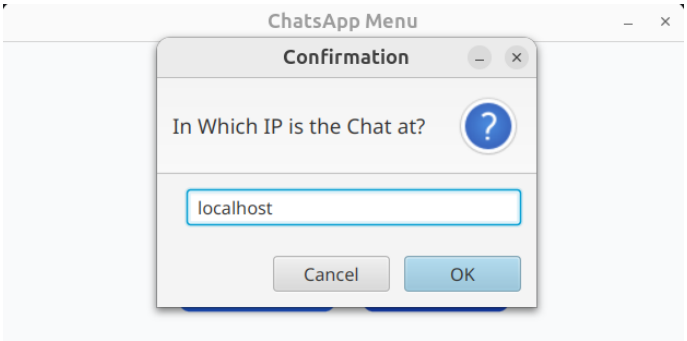


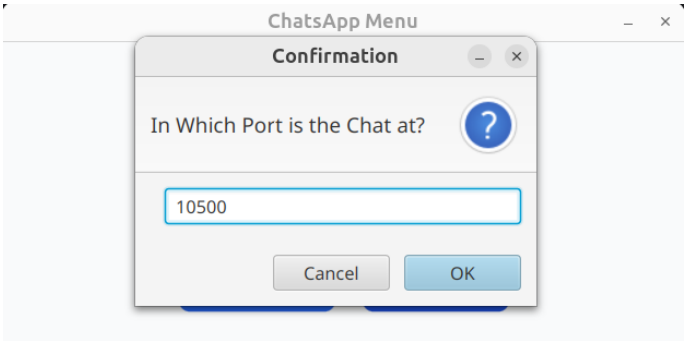
3.2. Join Chat

Now, let’s show how a user would join this chat.
Again in the menu scene, click the “Join Chat” button:

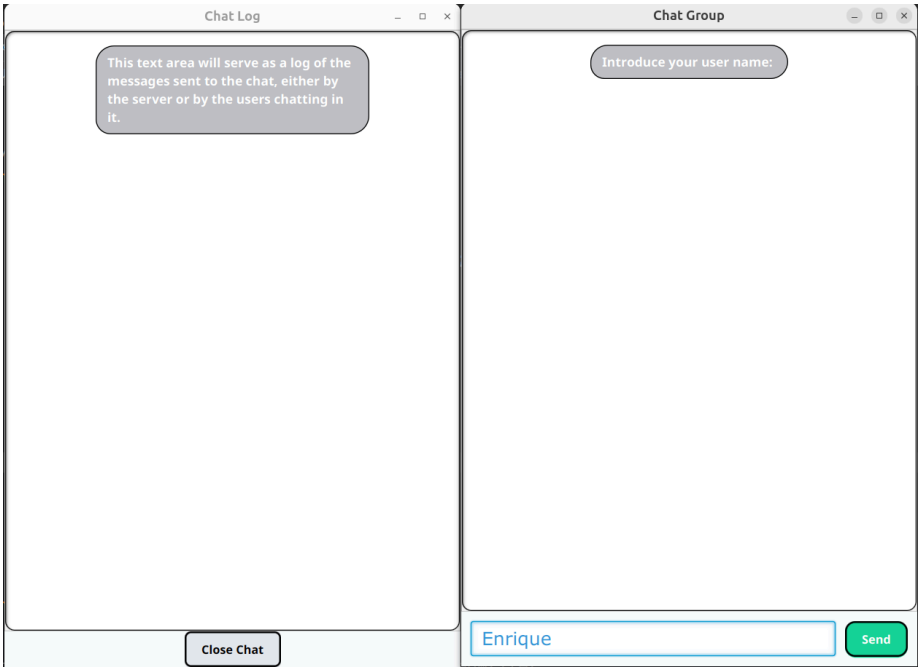


Afterwards, you will be asked to introduce the IP direction and port of the desired chat:

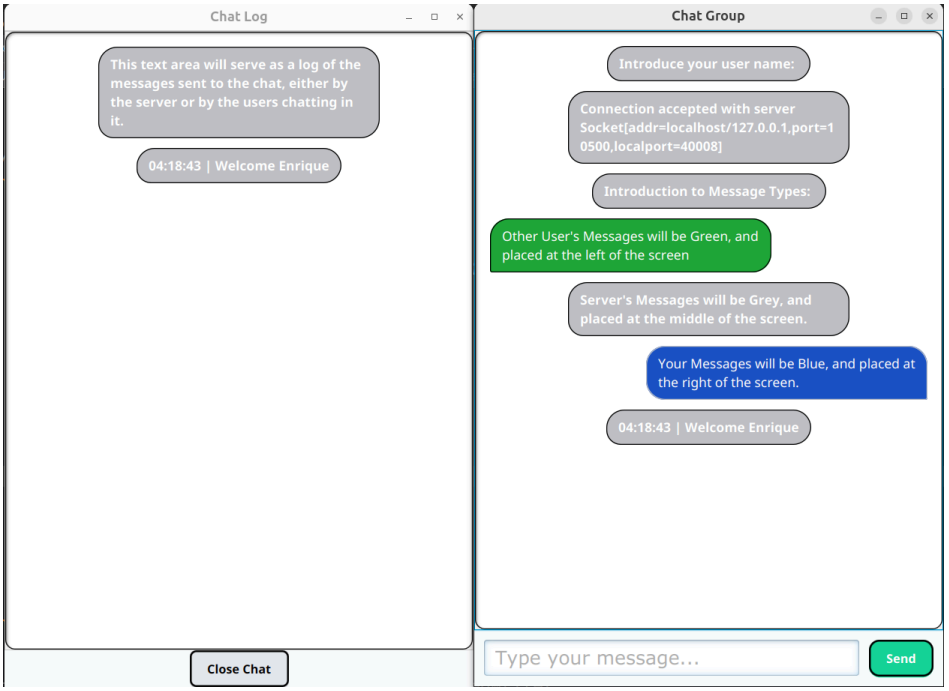




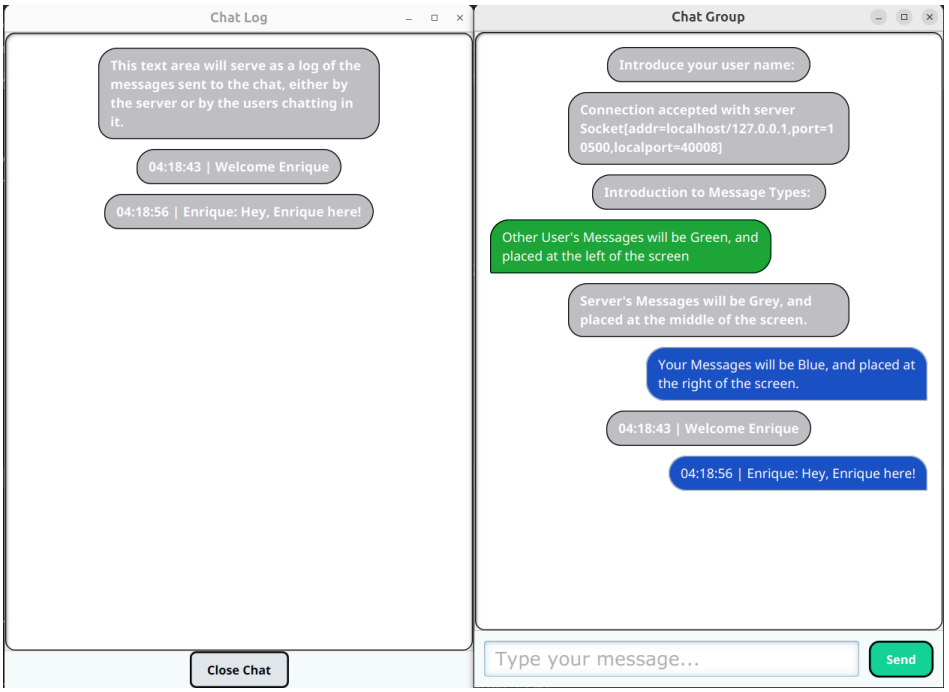
We will use localhost as IP and the port that we launched the server on previously.
After introducing the correct information, you will be asked to enter a username to be registered at the server.
For this first instance, we will introduce the name “Enrique”:



After pressing the “Send” button, we will have effectively joined the chat:

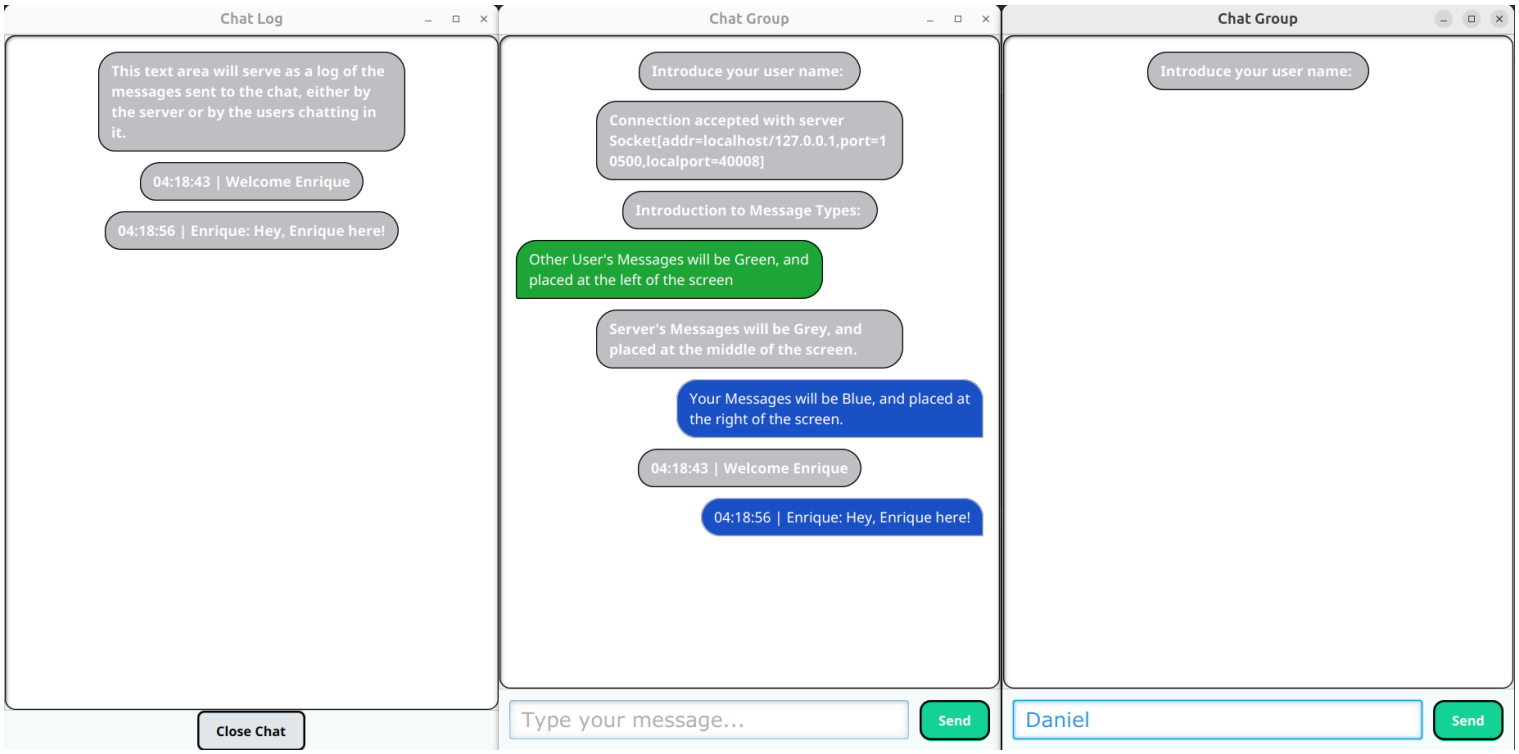


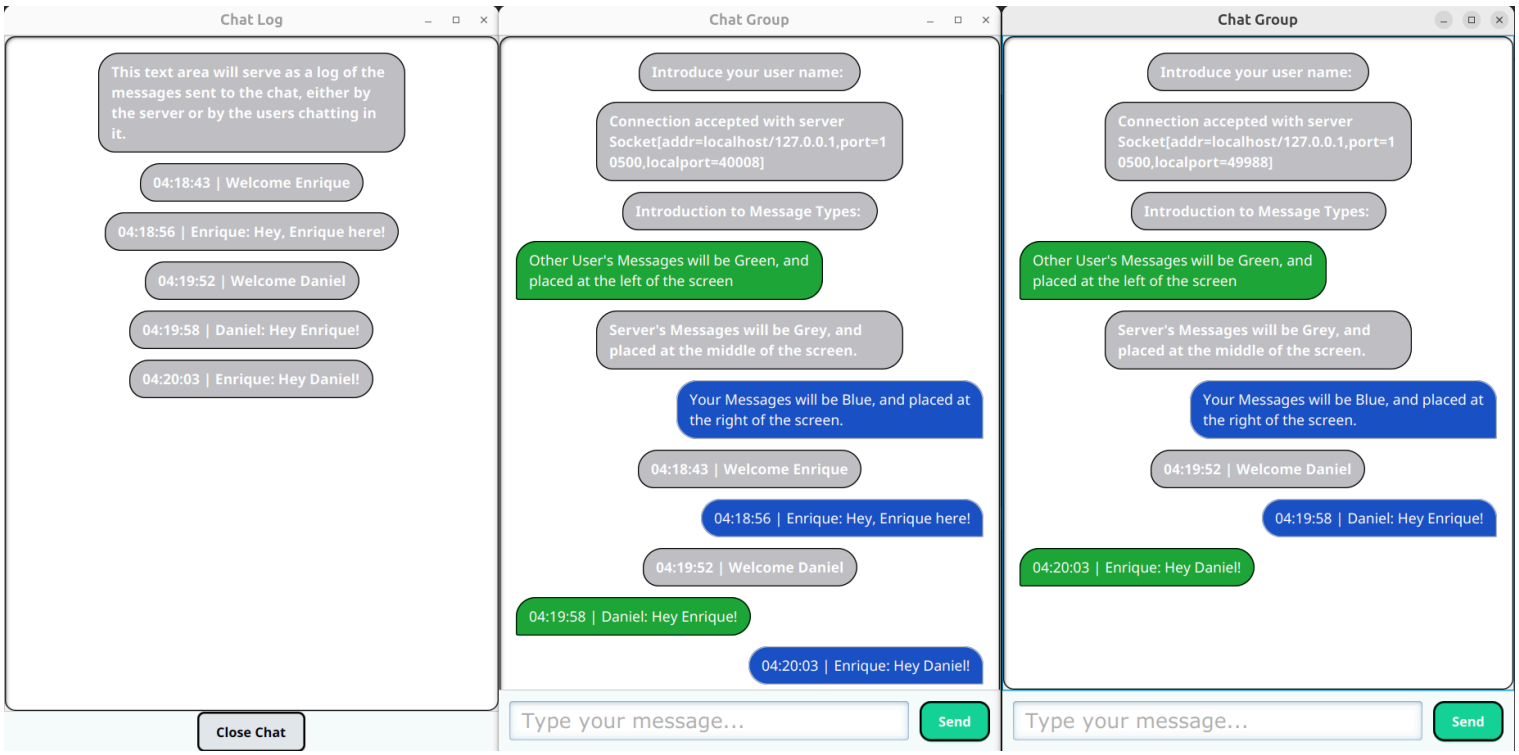
As we can see, we joined the server, got the introductory messages and received a welcome message.
We can also see that his messages appear in the chat:



3.3. Basic Communication

Now, let's test if the messages sent from Enrique and from a new user that we will join the chat, "Daniel", work properly:

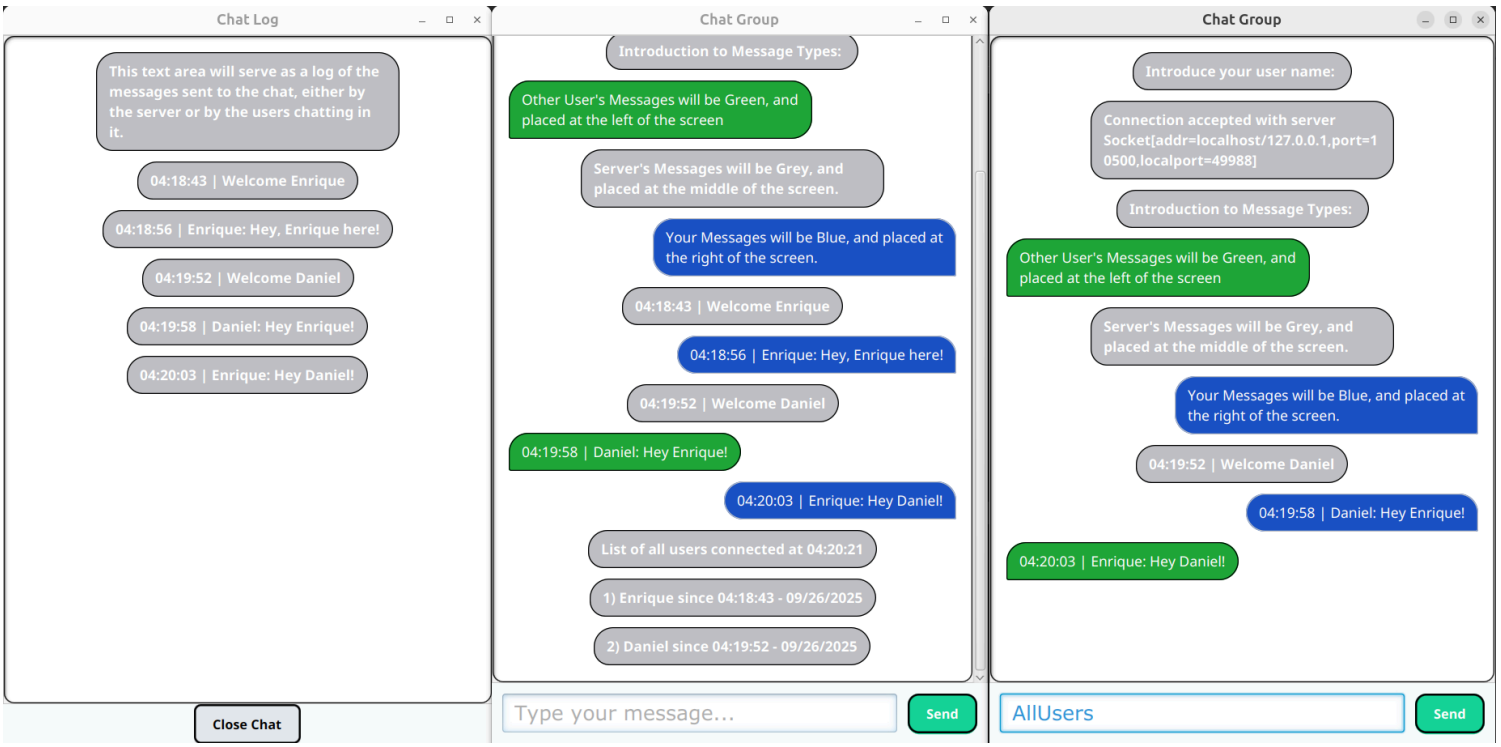




As we can see, the communication between the users works well.

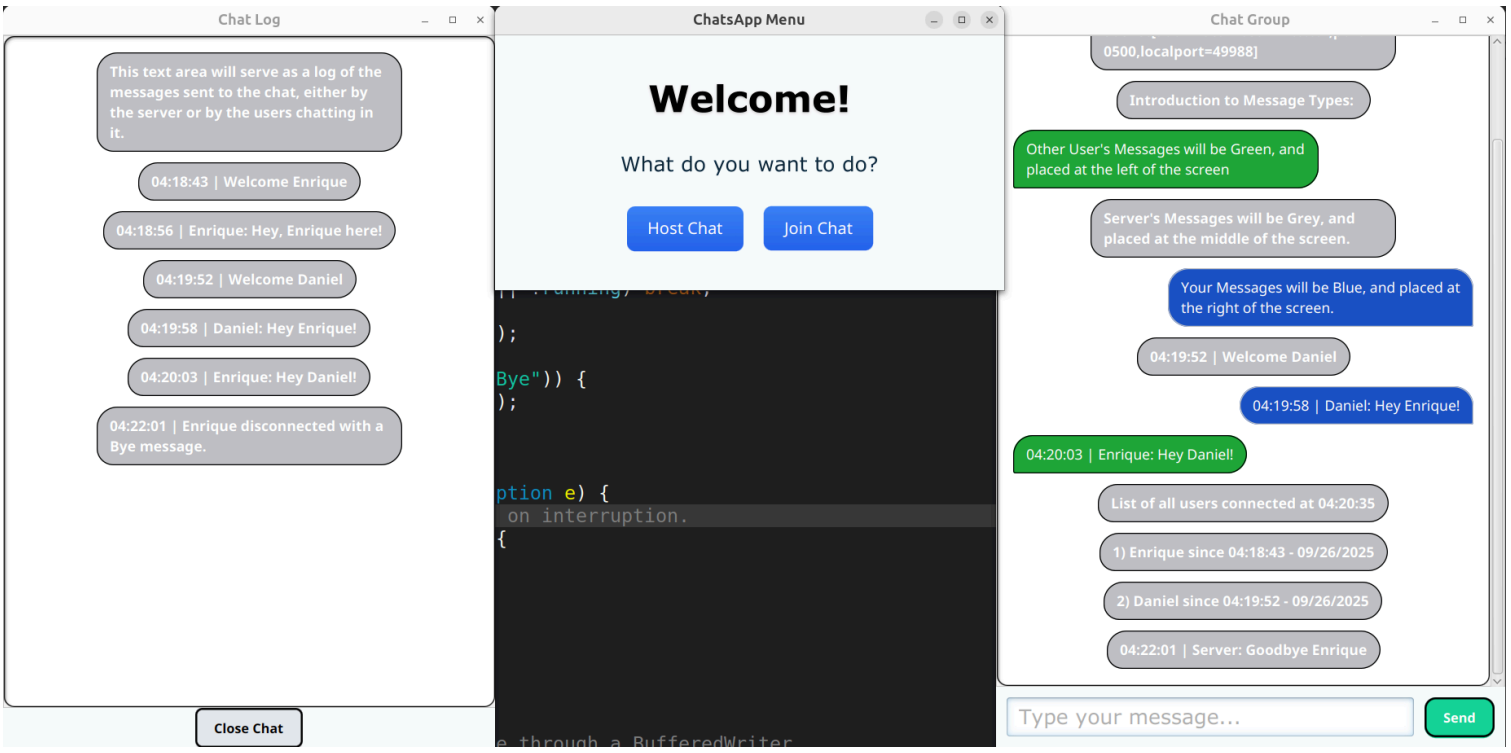
3.4. AllUsers and Bye Commands

Now, let's show how the AllUsers and Bye commands would work in this chat:



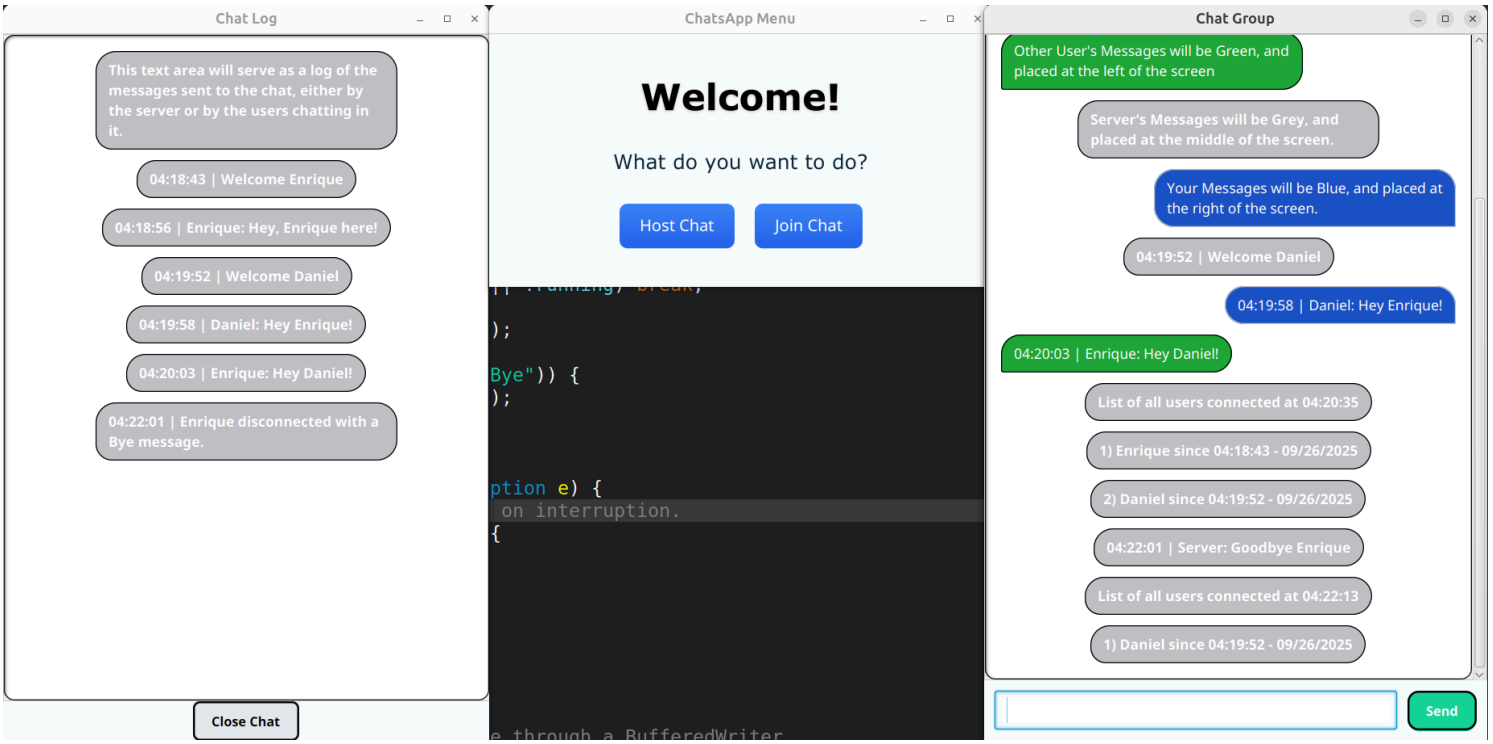
As we can see, both user's "AllUsers" commands worked well.
¿But what if we mix the usage of this command with the "Bye" command?

First, Enrique will disconnect using the "Bye" command:



Enrique left the chat, going back to the main menu, as expected.

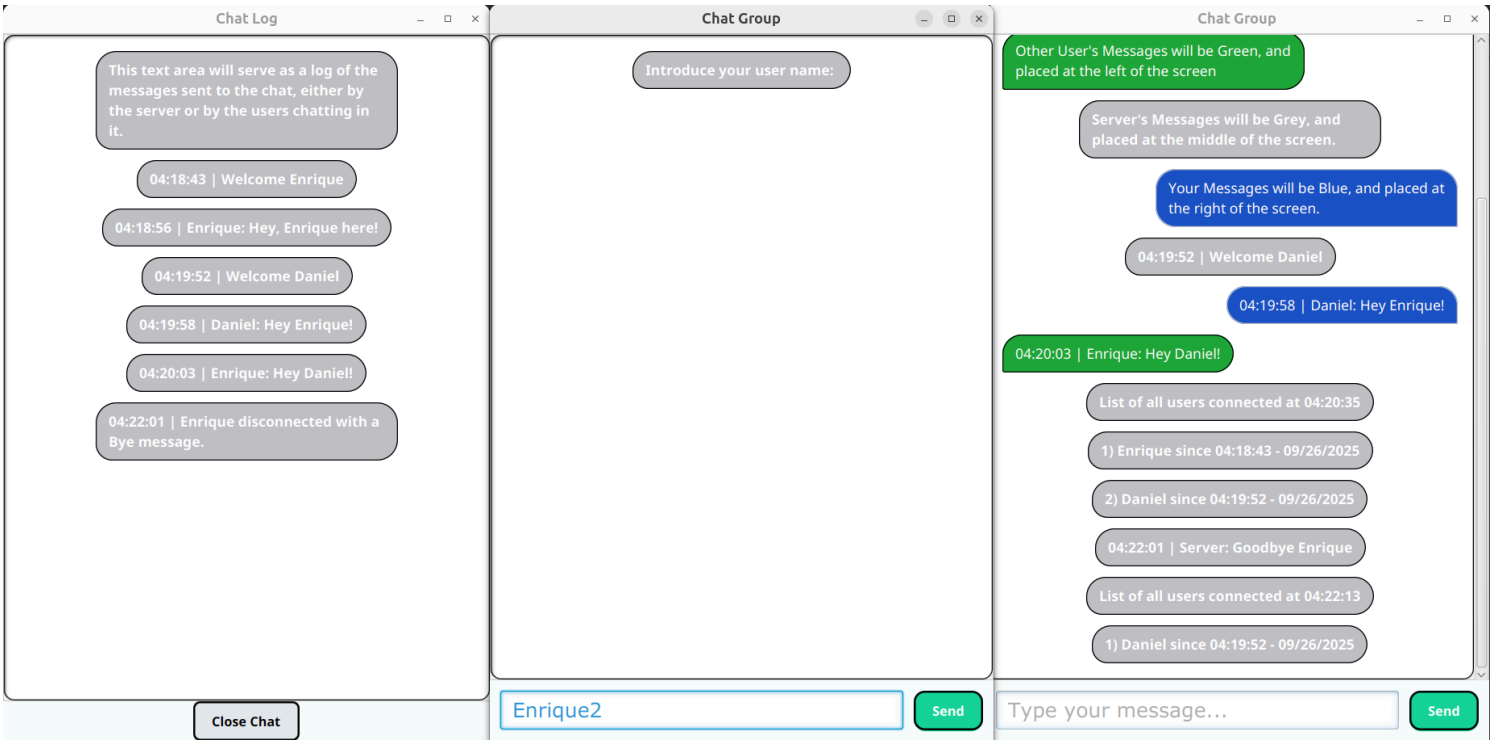
But what if Daniel now uses the “AllUsers” command?

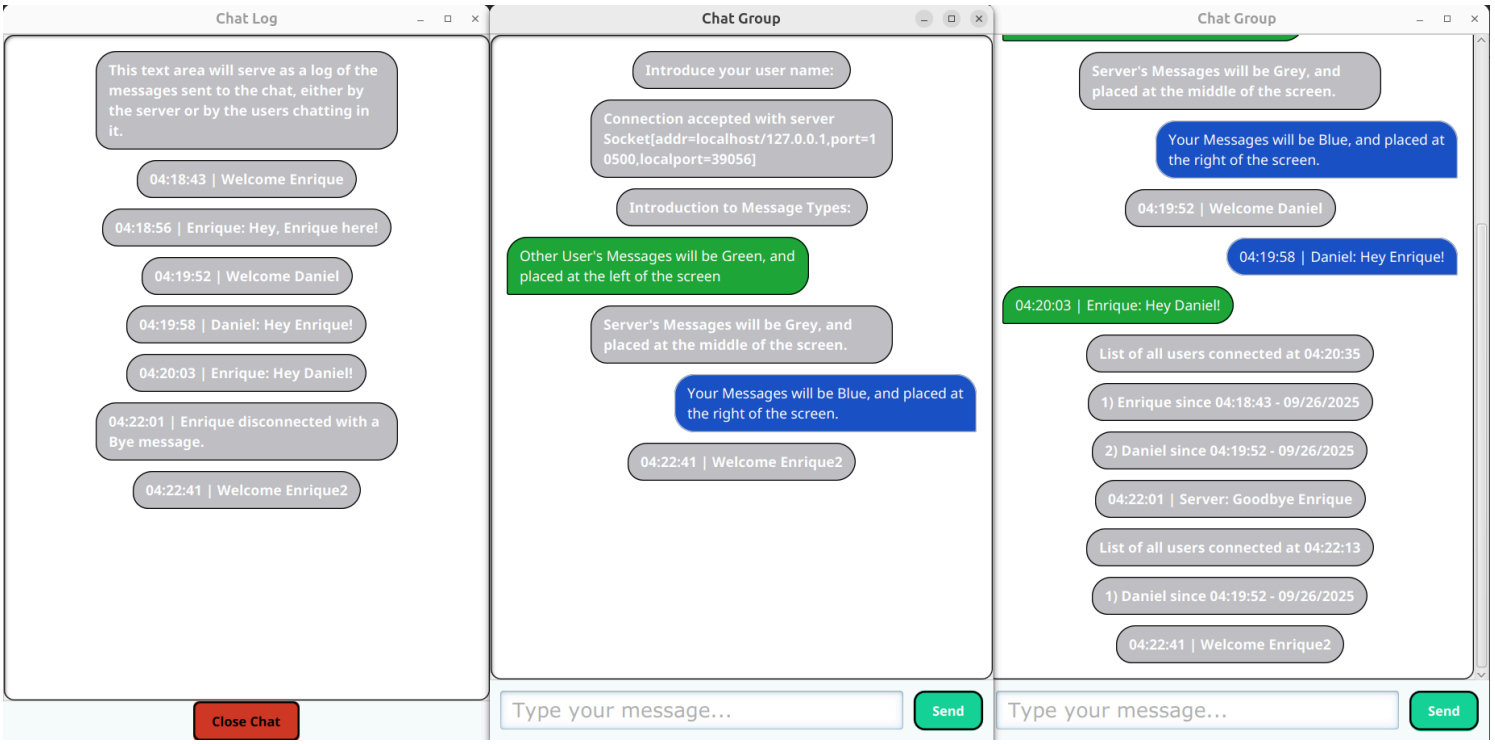


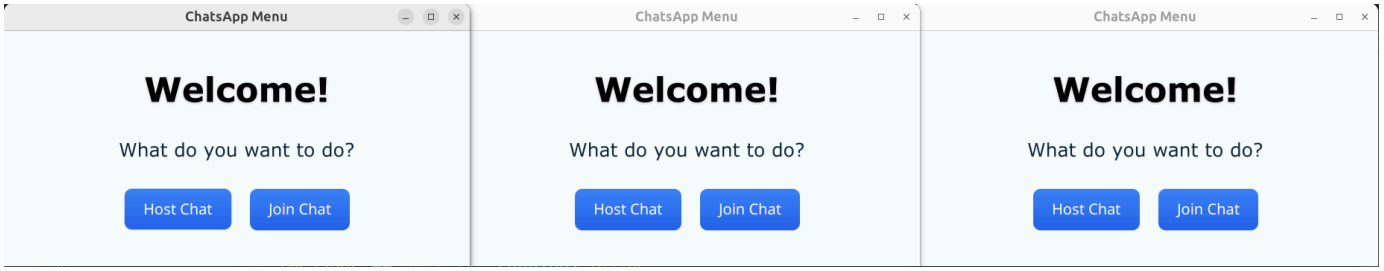
As we can see, he is the only one in the chat, again, as expected.

3.5. Close Chat Button

In order to check the “Close Chat” button, we will make another instance of Enrique to join the chat, and later press the button:



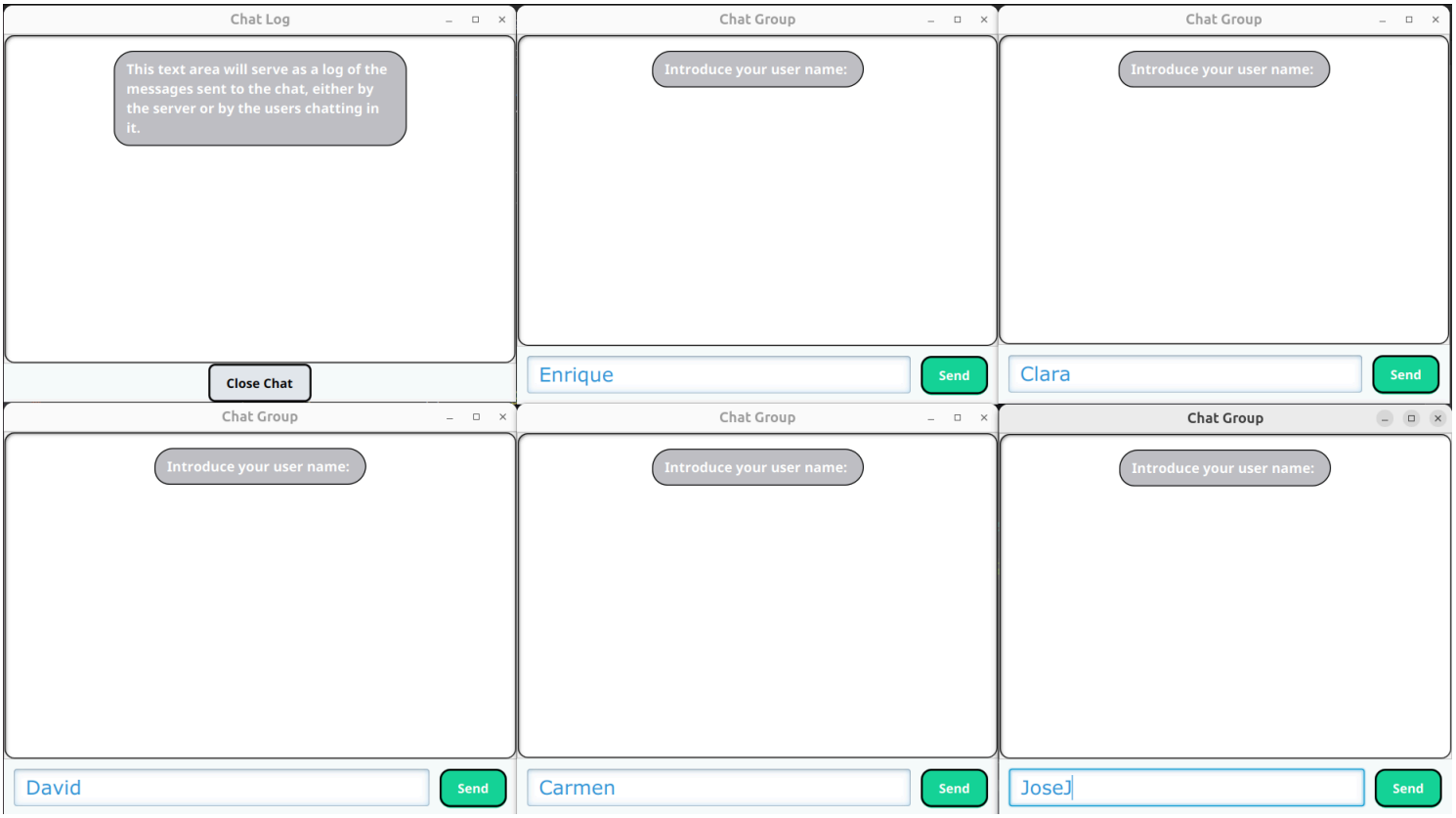


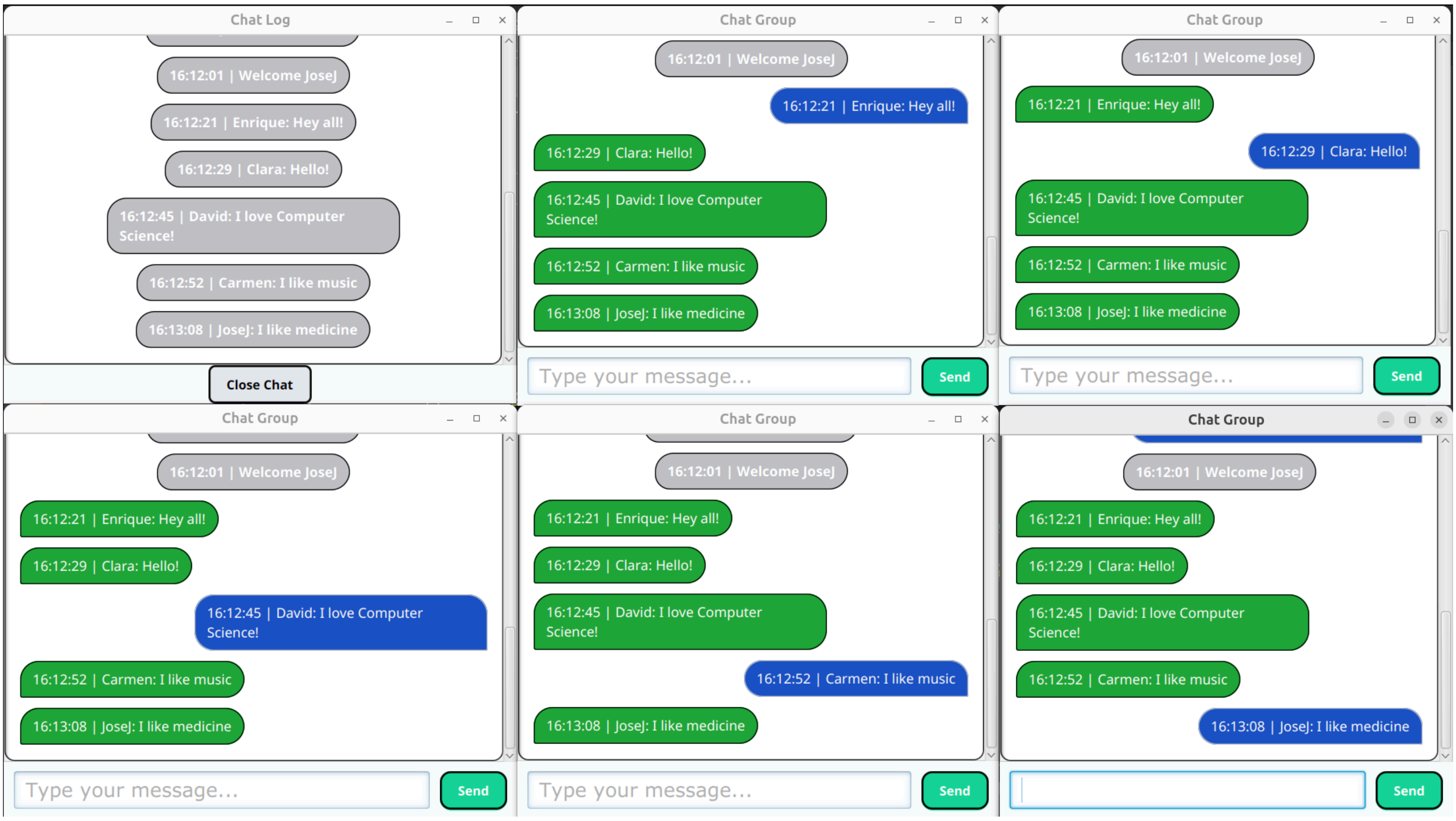


We can see that after the “Close Chat” button was pressed, the chat’s host as well as the clients connected got sent back to the main menu.

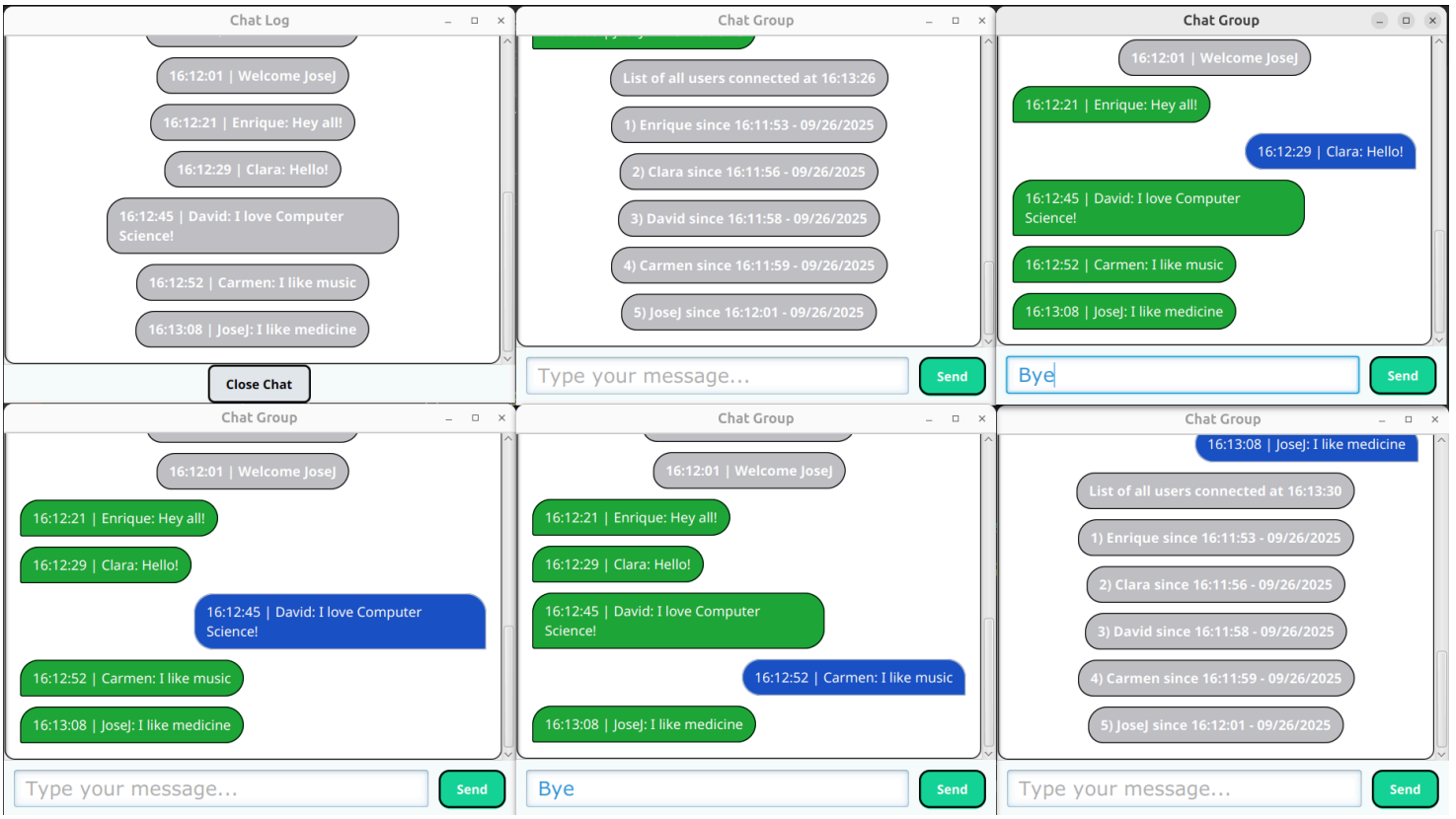
3.6. Larger Scale Testing

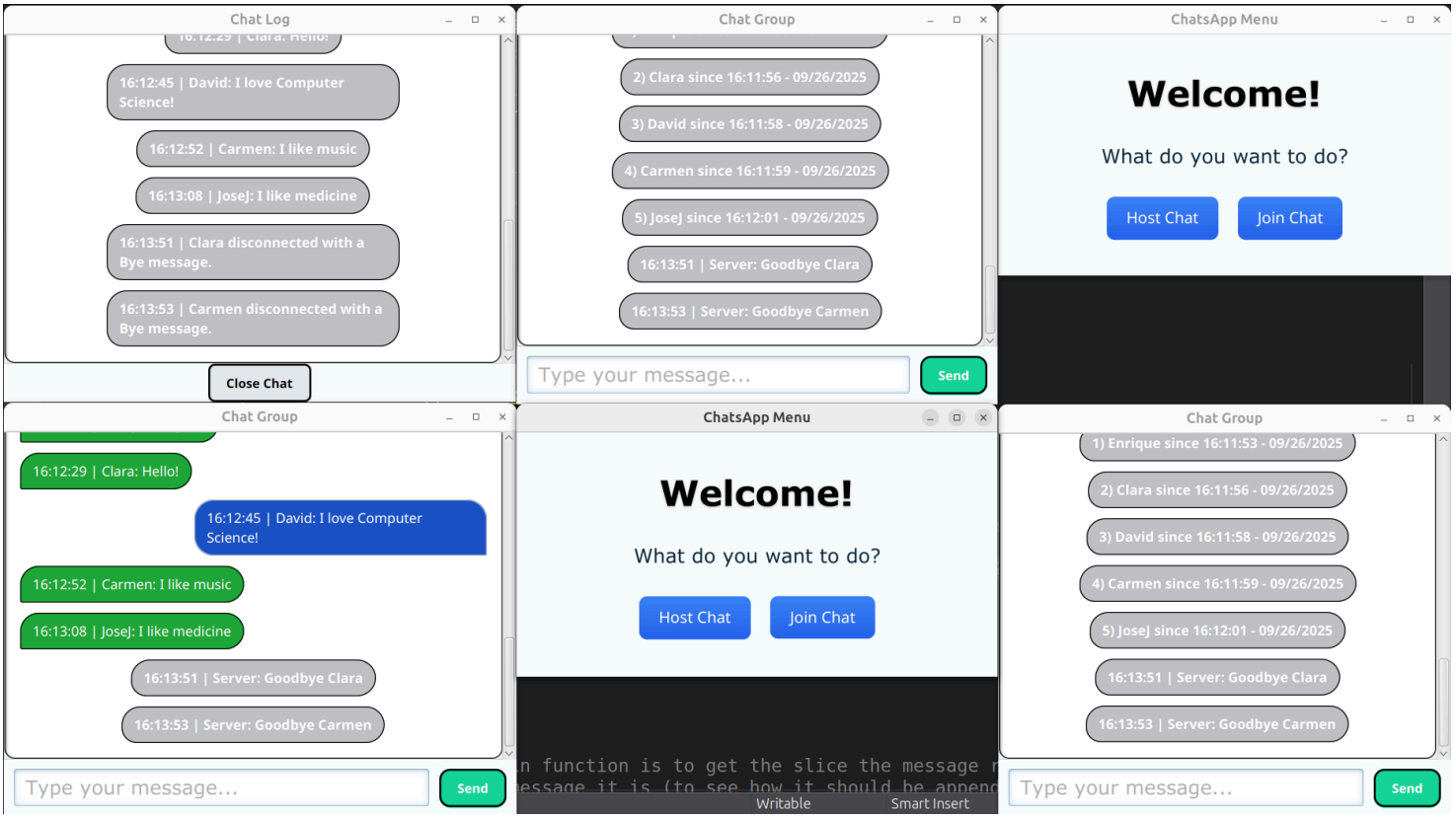
Finally, now that there is a basic understanding of how the app works, a larger scale chat with 5 users connected and interacting between each other will be shown, so that we can prove the app's reliability.

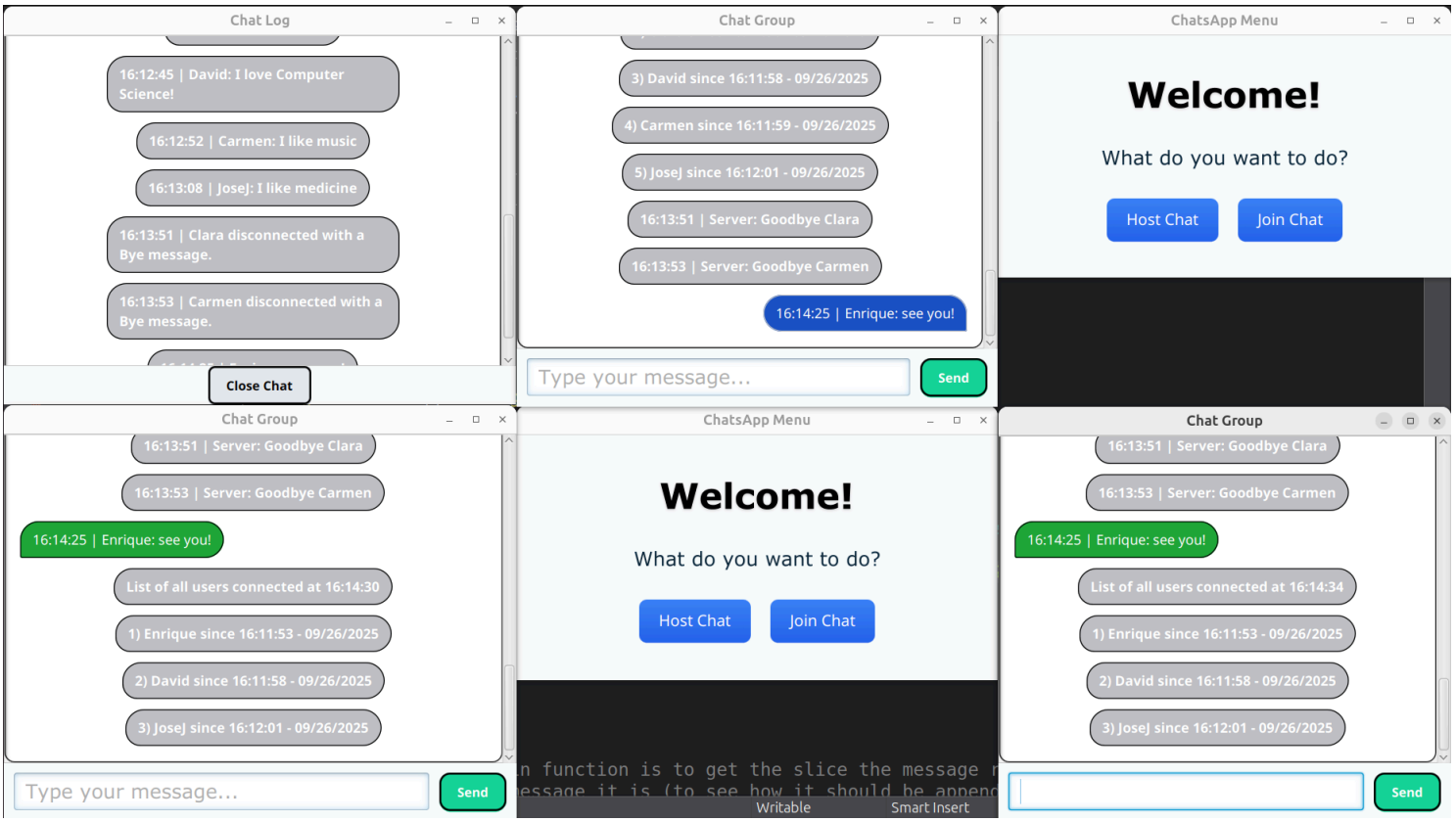


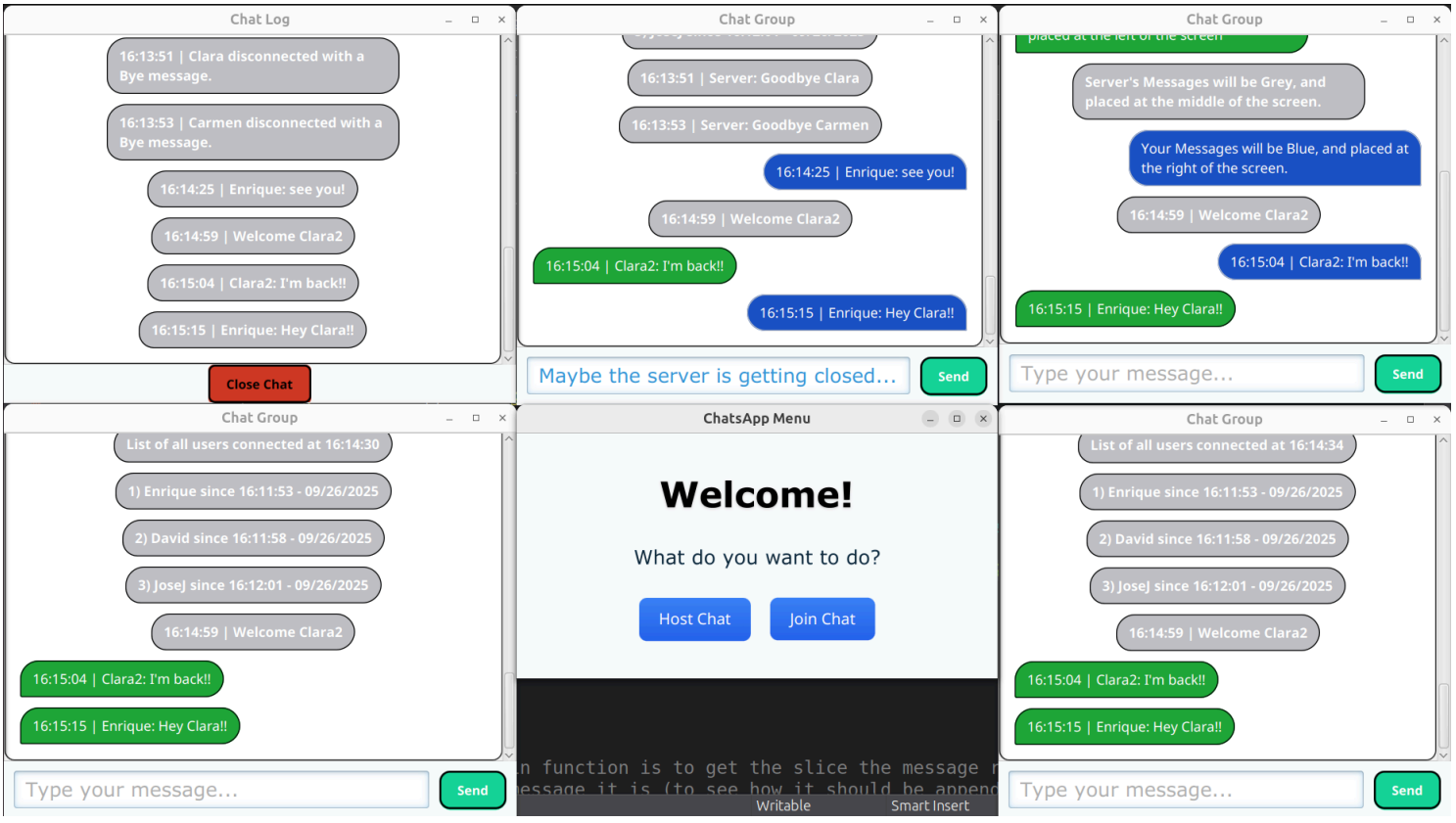

Chat Log
16:12:01 | Welcome JoseJ
16:12:21 | Enrique: Hey all!
16:12:29 | Clara: Hello!
16:12:45 | David: I love Computer Science!
16:12:52 | Carmen: I like music
16:13:08 | JoseJ: I like medicine
Close Chat
Chat Group
16:12:01 | Welcome JoseJ
16:12:21 | Enrique: Hey all!
16:12:29 | Clara: Hello!
16:12:45 | David: I love Computer Science!
16:12:52 | Carmen: I like music
16:13:08 | JoseJ: I like medicine
Type your message...
Send

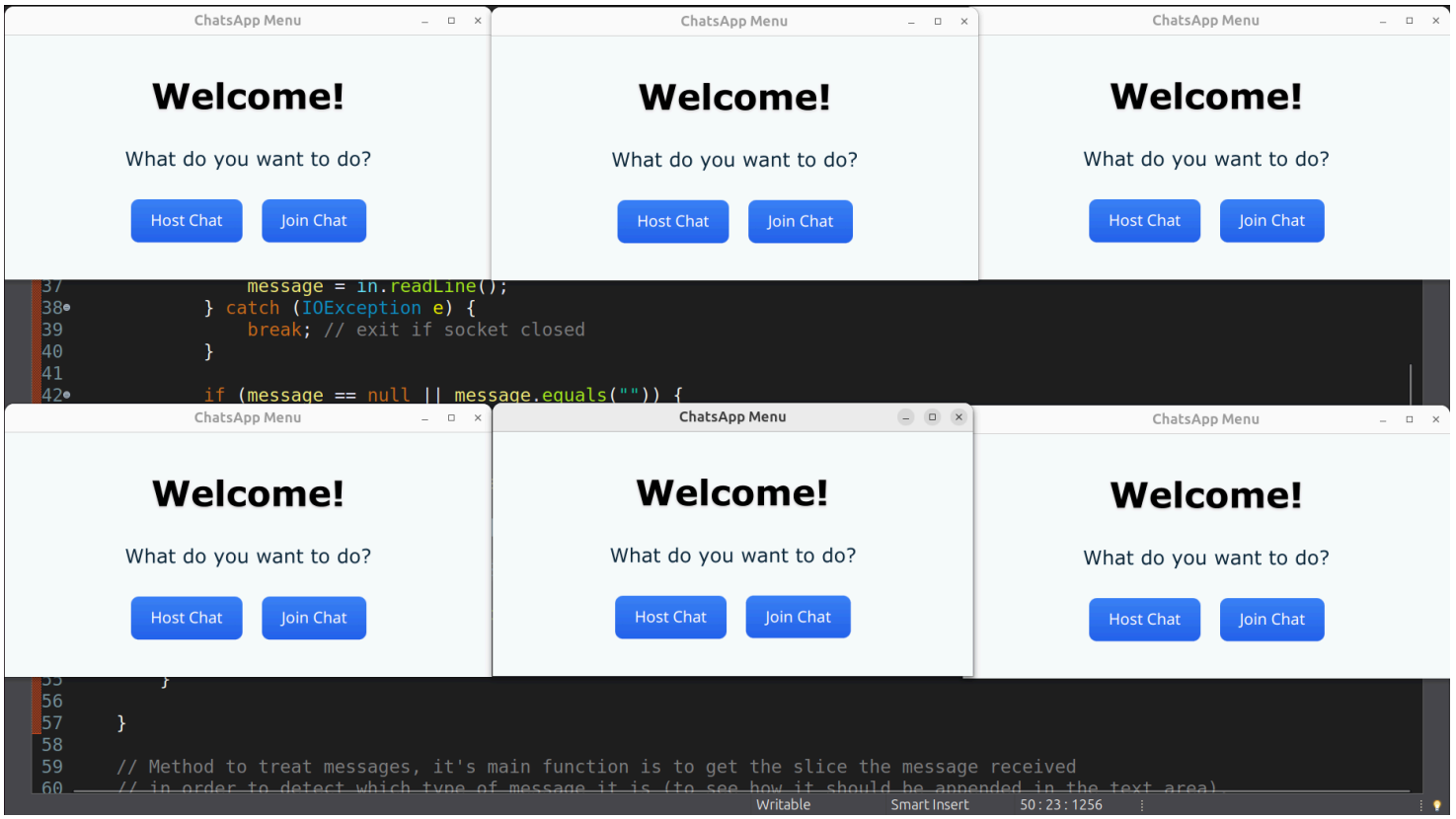

Chat Log
16:12:01 | Welcome JoseJ
16:12:21 | Enrique: Hey all!
16:12:29 | Clara: Hello!
16:12:45 | David: I love Computer Science!
16:12:52 | Carmen: I like music
16:13:08 | JoseJ: I like medicine
Close Chat
Chat Group
List of all users connected at 16:13:26
1) Enrique since 16:11:53 - 09/26/2025
2) Clara since 16:11:56 - 09/26/2025
3) David since 16:11:58 - 09/26/2025
4) Carmen since 16:11:59 - 09/26/2025
5) JoseJ since 16:12:01 - 09/26/2025
List of all users connected at 16:13:30
Type your message...
Send


Chat Log
16:12:01 | Welcome JoseJ
16:12:21 | Enrique: Hey all!
16:12:29 | Clara: Hello!
16:12:45 | David: I love Computer Science!
16:12:52 | Carmen: I like music
16:13:08 | JoseJ: I like medicine
Close Chat
Chat Group
List of all users connected at 16:13:26
1) Enrique since 16:11:53 - 09/26/2025
2) Clara since 16:11:56 - 09/26/2025
3) David since 16:11:58 - 09/26/2025
4) Carmen since 16:11:59 - 09/26/2025
5) JoseJ since 16:12:01 - 09/26/2025
Type your message...
Send
Chat Group
16:12:01 | Welcome JoseJ
16:12:21 | Enrique: Hey all!
16:12:29 | Clara: Hello!
16:12:45 | David: I love Computer Science!
16:12:52 | Carmen: I like music
16:13:08 | JoseJ: I like medicine
Bye
Send
Chat Group
16:12:01 | Welcome JoseJ
16:12:21 | Enrique: Hey all!
16:12:29 | Clara: Hello!
16:12:45 | David: I love Computer Science!
16:12:52 | Carmen: I like music
16:13:08 | JoseJ: I like medicine
Type your message...
Send
Chat Group
16:12:01 | Welcome JoseJ
16:12:21 | Enrique: Hey all!
16:12:29 | Clara: Hello!
16:12:45 | David: I love Computer Science!
16:12:52 | Carmen: I like music
16:13:08 | JoseJ: I like medicine
Bye
Send
Chat Group
16:13:08 | JoseJ: I like medicine
List of all users connected at 16:13:30
1) Enrique since 16:11:53 - 09/26/2025
2) Clara since 16:11:56 - 09/26/2025
3) David since 16:11:58 - 09/26/2025
4) Carmen since 16:11:59 - 09/26/2025
5) JoseJ since 16:12:01 - 09/26/2025
Type your message...
Send


Chat Log
16:12:45 | David: I love Computer Science!
16:12:52 | Carmen: I like music
16:13:08 | JoseJ: I like medicine
16:13:51 | Clara disconnected with a Bye message.
16:13:53 | Carmen disconnected with a Bye message.
Close Chat
Chat Group
2) Clara since 16:11:56 - 09/26/2025
3) David since 16:11:58 - 09/26/2025
4) Carmen since 16:11:59 - 09/26/2025
5) JoseJ since 16:12:01 - 09/26/2025
16:13:51 | Server: Goodbye Clara
16:13:53 | Server: Goodbye Carmen
Type your message...
Send
ChatsApp Menu
Welcome!
What do you want to do?
Host Chat
Join Chat
Chat Group
16:12:29 | Clara: Hello!
16:12:45 | David: I love Computer Science!
16:12:52 | Carmen: I like music
16:13:08 | JoseJ: I like medicine
16:13:51 | Server: Goodbye Clara
16:13:53 | Server: Goodbye Carmen
Type your message...
Send
ChatsApp Menu
Welcome!
What do you want to do?
Host Chat
Join Chat
n function is to get the slice the message
Writable
Smart Insert
Chat Group
1) Enrique since 16:11:53 - 09/26/2025
2) Clara since 16:11:56 - 09/26/2025
3) David since 16:11:58 - 09/26/2025
4) Carmen since 16:11:59 - 09/26/2025
5) JoseJ since 16:12:01 - 09/26/2025
16:13:51 | Server: Goodbye Clara
16:13:53 | Server: Goodbye Carmen
Type your message...
Send


Chat Log
16:12:45 | David: I love Computer Science!
16:12:52 | Carmen: I like music
16:13:08 | JoseJ: I like medicine
16:13:51 | Clara disconnected with a Bye message.
16:13:53 | Carmen disconnected with a Bye message.
Close Chat
Chat Group
3) David since 16:11:58 - 09/26/2025
4) Carmen since 16:11:59 - 09/26/2025
5) JoseJ since 16:12:01 - 09/26/2025
16:13:51 | Server: Goodbye Clara
16:13:53 | Server: Goodbye Carmen
16:14:25 | Enrique: see you!
Type your message...
Send
ChatsApp Menu
Welcome!
What do you want to do?
Host Chat
Join Chat
Chat Group
16:13:51 | Server: Goodbye Clara
16:13:53 | Server: Goodbye Carmen
16:14:25 | Enrique: see you!
List of all users connected at 16:14:30
1) Enrique since 16:11:53 - 09/26/2025
2) David since 16:11:58 - 09/26/2025
3) JoseJ since 16:12:01 - 09/26/2025
Type your message...
Send
ChatsApp Menu
Welcome!
What do you want to do?
Host Chat
Join Chat
Writable
Smart Insert
Chat Group
16:13:51 | Server: Goodbye Clara
16:13:53 | Server: Goodbye Carmen
16:14:25 | Enrique: see you!
List of all users connected at 16:14:34
1) Enrique since 16:11:53 - 09/26/2025
2) David since 16:11:58 - 09/26/2025
3) JoseJ since 16:12:01 - 09/26/2025
Send


Chat Log
16:13:51 | Clara disconnected with a Bye message.
16:13:53 | Carmen disconnected with a Bye message.
16:14:25 | Enrique: see you!
16:14:59 | Welcome Clara2
16:15:04 | Clara2: I'm back!!
16:15:15 | Enrique: Hey Clara!!
Close Chat
Chat Group
16:13:51 | Server: Goodbye Clara
16:13:53 | Server: Goodbye Carmen
16:14:25 | Enrique: see you!
16:14:59 | Welcome Clara2
16:15:04 | Clara2: I'm back!!
16:15:15 | Enrique: Hey Clara!!
Maybe the server is getting closed...
Send
Chat Group
Server's Messages will be Grey, and placed at the middle of the screen.
Your Messages will be Blue, and placed at the right of the screen.
16:14:59 | Welcome Clara2
16:15:04 | Clara2: I'm back!!
16:15:15 | Enrique: Hey Clara!!
Type your message...
Send
Chat Group
List of all users connected at 16:14:30
1) Enrique since 16:11:53 - 09/26/2025
2) David since 16:11:58 - 09/26/2025
3) JoseJ since 16:12:01 - 09/26/2025
16:14:59 | Welcome Clara2
16:15:04 | Clara2: I'm back!!
16:15:15 | Enrique: Hey Clara!!
Type your message...
Send
ChatsApp Menu
Welcome!
What do you want to do?
Host Chat
Join Chat
Writable
Smart Insert
Chat Group
List of all users connected at 16:14:34
1) Enrique since 16:11:53 - 09/26/2025
2) David since 16:11:58 - 09/26/2025
3) JoseJ since 16:12:01 - 09/26/2025
16:14:59 | Welcome Clara2
16:15:04 | Clara2: I'm back!!
16:15:15 | Enrique: Hey Clara!!
Type your message...
Send



4. Code

The code inside the project is completely commented and perfectly understandable. However, we are going to explain its structure and the logic behind it in this section.

4.1. Main Logic

An initial explanation of how the GUI handles the different threads and how these interact with each other is of great importance for the understanding of future explanations.

- Main Class - Menu:

The Menu class is, as you may imagine, the class that, when executed, shows the main app's menu that was shown in the previous section. From this menu, you can either create a chat or join one; but both actions will lead to the ChatUI class being launched.

- Chat Modelling Class - ChatUI:

This class is used to model the chat's host view as well as a chat's member view. When launched, depending on the type of view to model, it will run different threads to either model the client's view (run ClientSender thread) or the host's view (ServerAttender thread), which we will explain now.

- Host’s Structure - ServerAttender Thread:

When a ServerAttender thread is launched, the chat starts working, and users can join it.
Its main purpose is to accept connections from new users and, whenever one joins, its connection is accepted, and the ServerAttender thread launches a ServerThread thread, which will be in charge of that specific user’s communication with the chat.
As a summary, we have a main thread (ServerAttender), which’s function is to wait for new users to join the chat; whenever one sends a request, this thread accepts it, creating a specific thread (ServerThread) to be in charge of the user. After this, the main thread will continue waiting for new users to join the chat.

- Internal Host’s Thread - ServerThread Thread:

The main function of this thread is to listen to the messages that its user sends, and depending on the one received, treat them accordingly. We will see its main methods in future sections.
This approach of having a main thread and an additional thread in the server in charge of each active user in it was developed focused on a better experience for the user. If we only had one thread in charge of the server, it would be too busy attending new users' join requests and hearing messages from all of the different users already connected and treating them accordingly. Therefore, this multi-threaded approach is best suited to provide a great user experience, as well as a better server performance.

- Main Client Thread - ClientSender Thread:

This thread is the one launched by the main menu once the join chat button is pressed. Its main function is to listen to the incoming messages from its user, and send them to the corresponding ServerThread.
It is also in charge of launching the “ClientReceiver” thread, which will explain now.

- Secondary Client Thread - ClientReceiver Thread:

This thread is in charge of receiving messages sent from the corresponding ServerThread and showing them into the ChatUIs’ message area.
This double-threaded approach was developed due to a single thread not being able to handle listening messages from the user and from the server efficiently.

4.2. Supplementary Classes

A couple of supplementary classes were developed in order to provide an easier development. It is to be reminded that the project was developed using Java.

4.2.1. NameSocket

Class used to create a triplet of {user, joinTime, socket}. Its main purpose is to handle connected users easier in the server’s part.

Code

```
public record NameSocket(String username,
LocalDateTime joinTimeDate, Socket socket) {}
```

4.2.2. TimeFormatter

Class used to effectively show in a simple format times and dates:

Code

```
public class timeFormatter {

	// Atributes
	private DateTimeFormatter formatter;

	// Constructor
	public timeFormatter(String pattern) {
		if(pattern.equals("hourDate"))
			formatter = DateTimeFormatter.ofPattern("HH:mm:ss -
MM/dd/yyyy");
		else if(pattern.equals("hour"))
			formatter =
DateTimeFormatter.ofPattern("HH:mm:ss");
	}

	// Main Function.
	public String format(LocalDateTime time) {
		return time.format(formatter);
	}

}
```

4.3. Server Logic

4.3.1. ServerAttender Class

As we have already seen, this class' purpose is to receive users' connection requests and assign them a new ServerThread. Here are its main methods:

Main Method - run

Here, we can find the main method, run. Its purpose is to enter into a loop accepting new requests, and assigning new clients to ServerThreads, as we already discussed.
When entering this method, the first thing that id does is write an introductory message on the chat area (using chatui.appendMessage() method). This is done in mutual exclusion using the "screen" semaphore and its acquire (access resource) and release (free resource) methods.
Afterwards, the thread enters into a client accepting loop, in which:

- Accepts a client and assigns it a new socket. (Socket socket = server.accept();).
- Assigns it a new ServerThread, adds it to a list of the active server threads and launches it (thread = new ServerThread(...); threads.add(thread); thread.start();).
  We obviously access the thread list using a Semaphore (mutex).
- The loops' flow is controlled using the boolean variable "running".
  The whole loop is surrounded with a try-catch statement so that, whenever the loop wants to be stopped, the server will be closed, which will cause an Exception since it is likely that the thread will be blocked there.
- Finally, the connection is closed using the "communicateClosing()" method, which will be explained later.

```
@Override
      public void run() {

            // Get screen lock.
            try { screen.acquire(); } catch (InterruptedException e) {
e.printStackTrace(); }
            this.chatui.appendMessage("This text area will serve as a log of
the messages sent to the chat, either by the server or by the users chatting
in it.", "server");
            screen.release();

            try {
           while (running) { // Main client accepting loop.
               try {
                   Socket socket = server.accept();

                     // We create the thread that will be in charge of this
client.
                       // We create the thread that will be in
charge of this client.
                       ServerThread thread = null;
                       try {
                             thread = new ServerThread(mutex, screen,
socket, clients, chatui);
                       } catch (IOException e) {
e.printStackTrace(); }


                  try { mutex.acquire(); } catch
(InterruptedException e) { e.printStackTrace(); }
                     threads.add(thread);
```

```
                    mutex.release();

                    thread.start();
                } catch (SocketException e) {
                    chatui.appendMessage("Socket Closed", "server");
                    if (!running) break; // stop was called.
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally { communicateClosing(); }

    }
```

---

communicateClosing Method

This methods purpose is to completely shut down the server (used to implement the “Close Chat” button).

It closed every active ServerThread active, cleans the threads list, and changes to GUI to go back to the main menu.

```
public void communicateClosing() {
            for(ServerThread thread : this.threads) {
                try { thread.close(); } catch (Exception ignored)
{ }
            }

            threads.clear();

            // Switch back to menu using a javafx thread.
        javafx.application.Platform.runLater(() -> {
            this.originalStage.setScene(originalScene);
            this.originalStage.setTitle("ChatsApp Menu");
            this.originalStage.show();
        });
    }
```

---

closeServer Method

This method is used so that the main loop and be interrupted, and the server can start closing the server, since this interruption will lead to the execution of the communicateClosing method.

```
public void closeServer() {

            chatui.appendMessage("closing server", "server");

            // First, we will put the "running" variable to false,
so that the server will
            // get out of the main loop.
            this.running = false;

            // After that, we will close the server socket, just in
case it is blocked there.
            // This will let us end the chat even is the
serverAttender thread is blocked
            // waiting for a new client to connect.
            try { this.server.close(); } catch (IOException e) {
e.printStackTrace(); }
      }
```

4.3.2. ServerThread Class

Main Method - Run

This method first receives the username from its respective user. With this, the thread can finally add the user to the users list (using mutual exclusion, with the mutex Semaphore). Then, the user receives a welcome message (which is also written in the server's log). Finally, we have the main loop, where the thread receives messages from the user and treats them with the "handleMessage()" function (whichs' logic won't be covered here, since it is more clear reading the project's code directly).
Once the main loop ends (by either a Bye message ("closeBye()" function or by an interruption (same method used as in the ServerAttender thread, "close()" function), the connection is closed.

```
@Override
   public void run() {
        try {
            // First, get username.
            this.username = in.readLine();

            // Critical section: add client to client list.
            try { mutex.acquire(); } catch (InterruptedException e)
{ e.printStackTrace(); }
            this.actualClient = new NameSocket(this.username,
LocalDateTime.now(), socket);
            clients.add(actualClient);
            mutex.release();
```

```
            // Send Welcome message.
            String msg =
this.hourFormatter.format(LocalDateTime.now()) + " | Welcome " +
this.username;
            try { screen.acquire(); } catch (InterruptedException
e) { e.printStackTrace(); }
            this.chatui.appendMessage(msg, "server");
            screen.release();
            // Broadcast welcome of the new client to all active
clients.
            // The letter "S" is just to signal styling to the
client.
            broadcast("S" + msg);
            String line;
            while (running && (line = in.readLine()) != null) {
                 // Here, we check if the user inserted Bye.
                 // In that case, we end the execution of the
server thread (handleMessage returns true).
                if(handleMessage(line))
                 this.closeBye();
            }
        } catch (IOException e) {
            // Client disconnected unexpectedly or socket closed.
            // We close the connection in that case.
        } finally {
            close(); // Treat unexpected closing.
        }
    }
```

***The remaining class methods are better understood directly looking at the complete class' code.**

4.4. Client Logic

4.4.1. ClientSender Class

Main Method - Run

In this method, we initially write a series of messages in the chat area so that the user may understand how the message's styling works.

After this procedure, the ClientSender thread creates the ClientReceiver thread and launches it, giving it a reference to himself, so that the ClientReceiver can stop the ClientSender when the chats' activity stops.

Finally, the main loop is really simple; the thread just listens to messages from the user (chatui.getMessage()) and sends them to the ServerThread (this.sendMsg(message)).

When the loop ends, the connection is closed (method “closeConnection()”, which unblocks the thread (in case it is blocked) and changes the GUI back to the main menu.

```
@Override
      public void run() {
            try {
                  this.chatui.appendIntroUser();

                  // We pass a reference to ourselves so that the
client receiver thread
                  // can communicate us that the connections needs
to be closed.
                  ClientReceiver receiver = new
ClientReceiver(socket, this.chatui, this);
                  receiver.start();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            // Main loop.
            while (running) {
                try {
                  String message = chatui.getMessage();
                     if (message == null || !running) break;
                     this.sendMsg(message);
                     if (message.equals("Bye")) {
                         closeConnection();
                         return;
                     }
                } catch (InterruptedException e) {
                     break; // allow exit on interruption.
                } catch (IOException e) {
                     e.printStackTrace();
                     break;
                }
            }

            this.closeConnection();
      }
```

***The remaining class methods are better understood directly looking at the complete class’ code.**

4.4.2. ClientReceiver Class

Main Method - Run

This thread is even simpler. It just receives messages from the server (message = in.readLine();), interprets them with styling purposes (treatMessage() method) and, if the

message is a closing message, ends the connection (closing the clientSender as well (this.sender.closeConnection)), otherwise it continues the loop.

```
@Override
	public void run() {
		// Now, this thread only has to receive messages from
the server and
		// act based on the message received.
		while (true) {
			String message;
			try {
				message = in.readLine();
			} catch (IOException e) {
				break; // exit if socket closed
			}
			if (message == null || message.equals("")) {
				continue;
			}
			message = treatMessage(message);
			if (message.equals("Bye") ||
message.equals("Server is Being Closed - Bye")) {

				try { Thread.sleep(2000); } catch
(InterruptedException e) { e.printStackTrace(); }

				this.sender.closeConnection();
				return;
			}
		}
```

***The remaining class methods are better understood directly looking at the complete class' code.**

4.5. GUI

The GUI part of the code is based on the "Menu.java" and "ChatUI.java" classes.
Since our main interests when developing this app is the TCP communication infrastructure, we won't get deep into this GUI JavaFX coding.
Here, we will only show how our threads are launched from the main menu, and how a handler is established so that the connection is ended when closing is requested.

Starting Server
We can see that the menu invokes a new thread to start the server.

First it creates it (giving the actual stage and scene, so that the server can return the GUI to the menu), creates a handler, which will close the server if requested (chatui.setOnExitCallBack…) and launches it.

```
new Thread(() -> {
              try {
                 ServerAttender server = new ServerAttender(port,
chatui, stage, this.originalScene);

                 chatui.setOnExitCallback(() ->
server.closeServer());

                 server.start();
              } catch (IOException | InterruptedException e) {
                  e.printStackTrace();
              }
          }).start();
```

Starting Client

As we can see, this code is pretty similar to the launching of the server, but adapted to the client (basically, instead of ServerAttender, ClientSender).

```
new Thread(() -> {
              try {
                 ClientSender client = new ClientSender(ip, port,
chatui, stage, this.originalScene);

                 chatui.setOnExitCallback(() ->
client.closeConnection());

                 client.start();
              } catch (IOException e) { e.printStackTrace();
              } catch (InterruptedException e) {
e.printStackTrace();
              } catch (ExecutionException e) { e.printStackTrace();
}

          }).start();
```
