

# **Programación de Servicios y Procesos**

## **Chat Multicliente**

Aplicación de mensajería grupal con Sockets y JavaFX

---

### **Autor:**

Santi Martínez

### **Tecnologías:**

Java 21 — JavaFX — Sockets — Maven

21 de enero de 2026

# ÍNDICE

<b>1. Introducción</b>	<b>3</b>
1.1. Contexto del Proyecto . . . . .	3
1.2. Objetivos del Proyecto . . . . .	3
1.3. Tecnologías Utilizadas . . . . .	4
1.4. Estructura de la Documentación . . . . .	4
<b>2. Arquitectura del Sistema</b>	<b>5</b>
2.1. Visión General . . . . .	5
2.2. Estructura de Paquetes . . . . .	5
2.3. Diagrama de Componentes . . . . .	5
2.4. Principios de Diseño Aplicados . . . . .	6
2.4.1. Separación de Responsabilidades . . . . .	6
2.4.2. Concurrencia Controlada . . . . .	6
2.4.3. Modularidad . . . . .	6
<b>3. Implementación del Servidor</b>	<b>7</b>
3.1. Clase Servidor . . . . .	7
3.2. Características del Servidor . . . . .	7
3.2.1. Puerto de Escucha . . . . .	7
3.2.2. Gestión de Clientes . . . . .	8
3.2.3. Pool de Hilos . . . . .	8
3.2.4. Método Broadcast . . . . .	8
3.3. Clase HandlerCliente . . . . .	8
3.4. Ciclo de Vida de una Conexión . . . . .	9
<b>4. Implementación del Cliente</b>	<b>10</b>
4.1. Clase App . . . . .	10
4.2. Características del Cliente . . . . .	11
4.2.1. Inicio Automático del Servidor . . . . .	11
4.2.2. Diálogo de Identificación . . . . .	11
4.2.3. Carga de la Interfaz . . . . .	11
4.3. Clase MainController . . . . .	11
4.4. Gestión de Mensajes . . . . .	12
4.4.1. Envío de Mensajes . . . . .	12
4.4.2. Recepción de Mensajes . . . . .	12
<b>5. Protocolo de Comunicación</b>	<b>14</b>
5.1. Formato de Mensajes . . . . .	14
5.2. Flujo de Comunicación . . . . .	14
5.3. Diagrama de Secuencia . . . . .	14
<b>6. Interfaz de Usuario</b>	<b>15</b>
6.1. Componentes de la Interfaz . . . . .	16
6.2. Archivo FXML . . . . .	16
6.3. Estilos CSS . . . . .	17
6.4. Métodos de Visualización . . . . .	17

6.4.1. Agregar Mensaje Propio . . . . .	17
6.4.2. Agregar Mensaje Recibido . . . . .	18
<b>7. Compilación y Ejecución</b>	<b>19</b>
7.1. Estructura del POM . . . . .	19
7.2. Comandos de Ejecución . . . . .	19
7.2.1. Compilar el Proyecto . . . . .	19
7.2.2. Ejecutar la Aplicación . . . . .	19
7.2.3. Ejecutar Múltiples Instancias . . . . .	19
7.3. Clase Launcher . . . . .	19
<b>8. Problemas Encontrados y Soluciones</b>	<b>20</b>
8.1. Problema 1: Sincronización de Hilos . . . . .	20
8.2. Problema 2: Actualización de la Interfaz desde Hilos . . . . .	20
8.3. Problema 3: Cierre de Conexiones . . . . .	20
8.4. Problema 4: Mensajes Duplicados . . . . .	20
8.5. Problema 5: Configuración del Puerto . . . . .	21
8.6. Problema 6: Scroll Automático . . . . .	21
<b>9. Conclusión</b>	<b>22</b>
9.1. Objetivos Alcanzados . . . . .	22
9.2. Aprendizajes Técnicos . . . . .	22
9.3. Posibles Mejoras Futuras . . . . .	22
9.4. Reflexión Personal . . . . .	23
<b>10. Bibliografía</b>	<b>24</b>

# 1. Introducción

Esta documentación desarrolla un **sistema de chat multicliente** implementado en Java, utilizando sockets TCP/IP para la comunicación en red y JavaFX para la interfaz gráfica de usuario.

El proyecto tiene como objetivo demostrar la implementación práctica de conceptos fundamentales de programación concurrente, comunicación en red y desarrollo de interfaces gráficas, creando una aplicación funcional que permite la comunicación simultánea entre múltiples usuarios conectados a través de una red local o internet.

## 1.1. Contexto del Proyecto

En el ámbito de la programación de servicios y procesos, la comunicación cliente-servidor mediante sockets representa uno de los pilares fundamentales para el desarrollo de aplicaciones distribuidas. Este proyecto surge como respuesta a la necesidad de implementar un sistema que permita:

- La comunicación en tiempo real entre múltiples clientes
- La gestión eficiente de conexiones concurrentes mediante hilos
- Una interfaz de usuario intuitiva y responsive
- El manejo robusto de errores y desconexiones inesperadas

## 1.2. Objetivos del Proyecto

Los objetivos específicos que se persiguen con este desarrollo son:

1. **Implementar un servidor multihilo** capaz de gestionar conexiones simultáneas de múltiples clientes
2. **Desarrollar un cliente con interfaz gráfica** utilizando JavaFX que permita una experiencia de usuario fluida
3. **Gestionar la comunicación** mediante sockets TCP/IP, asegurando la transmisión fiable de mensajes
4. **Aplicar conceptos de programación concurrente** para garantizar la sincronización correcta entre hilos
5. **Implementar un sistema de difusión** (broadcast) que permita enviar mensajes a todos los clientes conectados
6. **Manejar eventos de conexión y desconexión** de forma transparente para los usuarios

### **1.3. Tecnologías Utilizadas**

El proyecto se desarrolla sobre las siguientes tecnologías y herramientas:

- **Java 21:** Versión LTS del lenguaje, aprovechando las últimas características y mejoras de rendimiento
- **JavaFX 21.0.6:** Framework para desarrollo de interfaces gráficas modernas y responsivas
- **Maven:** Herramienta de gestión de dependencias y construcción del proyecto
- **Sockets TCP/IP:** API de Java para comunicación en red
- **Threads y Concurrencia:** ExecutorService, ConcurrentHashMap para gestión de hilos

### **1.4. Estructura de la Documentación**

Este documento se organiza en las siguientes secciones:

- **Arquitectura del Sistema:** Descripción general de la estructura y componentes principales
- **Implementación del Servidor:** Análisis detallado del componente servidor y gestión de clientes
- **Implementación del Cliente:** Desarrollo del cliente con interfaz gráfica
- **Protocolo de Comunicación:** Protocolo de mensajes y flujo de información
- **Interfaz de Usuario:** Diseño y funcionalidades de la GUI con JavaFX
- **Problemas y Soluciones:** Dificultades encontradas durante el desarrollo
- **Conclusiones:** Reflexiones finales y aprendizajes del proyecto

## 2. Arquitectura del Sistema

La aplicación de chat multicliente sigue una arquitectura cliente-servidor clásica, donde un servidor central gestiona las conexiones y redistribuye los mensajes entre todos los clientes conectados.

### 2.1. Visión General

El sistema se compone de los siguientes elementos principales:

- **Servidor Central:** Acepta conexiones entrantes, crea un hilo para cada cliente y gestiona la difusión de mensajes
- **Clientes JavaFX:** Aplicaciones con interfaz gráfica que se conectan al servidor
- **Protocolo de Comunicación:** Basado en líneas de texto sobre sockets TCP/IP
- **Sistema de Broadcast:** Mecanismo para enviar mensajes a todos los clientes simultáneamente

### 2.2. Estructura de Paquetes

El proyecto está organizado en paquetes siguiendo principios de separación de responsabilidades:

- `com.chat.chatmulticraithreads.servicios`: Contiene las clases relacionadas con el servidor y la gestión de clientes (`Servidor`, `HandlerCliente`)
- `com.chat.chatmulticraithreads.cliente`: Incluye las clases del cliente JavaFX (`App`, `MainController`, `Launcher`)
- `resources`: Archivos de recursos como FXML y CSS para la interfaz gráfica
- `pom.xml`: Archivo de configuración de Maven con dependencias y plugins

### 2.3. Diagrama de Componentes

El siguiente diagrama ilustra la relación entre los componentes principales:

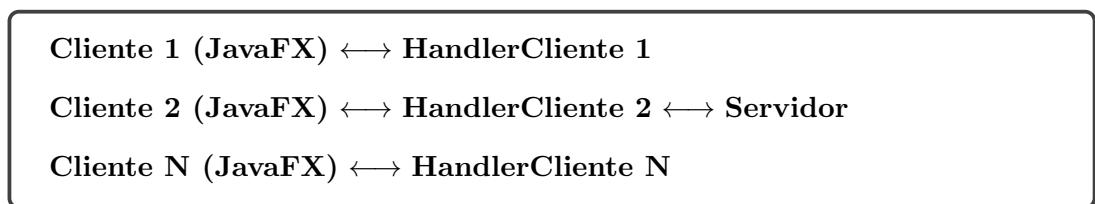


Figura 1: Arquitectura Cliente-Servidor del Chat

## 2.4. Principios de Diseño Aplicados

### 2.4.1. Separación de Responsabilidades

Cada clase tiene una responsabilidad única y bien definida:

- **Servidor**: Aceptar conexiones y coordinar broadcast
- **HandlerCliente**: Gestionar comunicación con un cliente específico
- **App**: Inicializar interfaz gráfica y conectar al servidor
- **MainController**: Controlar eventos de UI y mostrar mensajes

### 2.4.2. Concurrencia Controlada

El manejo de múltiples clientes simultáneos se realiza mediante:

- **ExecutorService** con pool de hilos cacheados
- **ConcurrentHashMap.newKeySet()** para almacenar clientes de forma thread-safe
- Sincronización automática en operaciones de broadcast

### 2.4.3. Modularidad

La separación entre lógica de red y presentación permite:

- Probar el servidor independientemente de la interfaz
- Reemplazar la GUI sin modificar la lógica de red
- Extender funcionalidades sin afectar componentes existentes

## 3. Implementación del Servidor

El servidor es el componente central que coordina toda la comunicación entre clientes. Su responsabilidad principal es aceptar conexiones entrantes y gestionar el envío de mensajes entre todos los participantes.

### 3.1. Clase Servidor

La clase `Servidor` implementa el socket servidor que escucha en un puerto específico y crea un nuevo `HandlerCliente` para cada conexión entrante.

```
1 public class Servidor {
2     private static final int PUERTO = 8080;
3     public static final Set<HandlerCliente> clientes =
4         ConcurrentHashMap.newKeySet();
5
6     public static void main(String[] args) {
7         ExecutorService pool = Executors.newCachedThreadPool();
8         System.out.println("Servidor iniciado en puerto " +
9             PUERTO);
10
11     try (ServerSocket serverSocket = new ServerSocket(
12         PUERTO)) {
13         while (true) {
14             Socket socket = serverSocket.accept();
15             System.out.println("Cliente conectado: " +
16                 socket.getInetAddress());
17             pool.execute(new HandlerCliente(socket));
18         }
19     } catch (IOException e) {
20         System.err.println("Error: " + e.getMessage());
21     }
22
23     public static void broadcast(String mensaje) {
24         clientes.forEach(c -> c.enviar(mensaje));
25     }
}
```

Listing 1: Implementación de la clase Servidor

### 3.2. Características del Servidor

#### 3.2.1. Puerto de Escucha

El servidor escucha en el puerto **8080**. Esta es una constante que puede modificarse según las necesidades de despliegue.

### 3.2.2. Gestión de Clientes

Se utiliza un `ConcurrentHashMap.newKeySet()` para almacenar los manejadores de clientes activos. Esta estructura de datos es thread-safe y permite operaciones concurrentes sin necesidad de sincronización explícita.

### 3.2.3. Pool de Hilos

El servidor utiliza un `ExecutorService` con `newCachedThreadPool()`, que crea hilos según sea necesario y reutiliza hilos inactivos, optimizando el uso de recursos.

### 3.2.4. Método Broadcast

El método `broadcast()` es estático y puede ser invocado desde cualquier `HandlerCliente` para enviar un mensaje a todos los clientes conectados simultáneamente.

## 3.3. Clase HandlerCliente

Cada cliente conectado es gestionado por una instancia de `HandlerCliente`, que se ejecuta en su propio hilo.

```
1 public class HandlerCliente implements Runnable {
2     private final Socket socket;
3     private PrintWriter salida;
4     private String nombre;
5
6     public HandlerCliente(Socket socket) {
7         this.socket = socket;
8     }
9
10    @Override
11    public void run() {
12        try {
13            BufferedReader entrada = new BufferedReader(
14                new InputStreamReader(socket.getInputStream()));
15                ;
16            salida = new PrintWriter(socket.getOutputStream(),
17                true);
18
19            nombre = entrada.readLine();
20            if (nombre == null || nombre.isEmpty())
21                nombre = "Usuario";
22
23            Servidor.clientes.add(this);
24            Servidor.broadcast(nombre + " se ha unido");
25
26            String mensaje;
27            while ((mensaje = entrada.readLine()) != null) {
28                if (mensaje.equalsIgnoreCase("salir"))
29                    break;
30                Servidor.broadcast(mensaje);
31            }
32        } catch (IOException e) {
33            e.printStackTrace();
34        }
35    }
36}
```

```

30     } catch (IOException e) {
31         // Cliente desconectado
32     } finally {
33         Servidor.clientes.remove(this);
34         if (nombre != null) {
35             Servidor.broadcast(nombre + " se ha ido");
36         }
37         try {
38             socket.close();
39         } catch (IOException ignored) {
40             }
41     }
42 }
43
44 public void enviar(String mensaje) {
45     if (salida != null)
46         salida.println(mensaje);
47 }
48 }
```

Listing 2: Implementación de HandlerCliente

### 3.4. Ciclo de Vida de una Conexión

1. **Aceptación:** El servidor acepta la conexión y crea un socket
2. **Creación:** Se instancia un HandlerCliente con el socket
3. **Registro:** El cliente se añade al conjunto de clientes activos
4. **Anuncio:** Se notifica a todos que el cliente se ha unido
5. **Escucha:** Se entra en un bucle para recibir mensajes
6. **Broadcast:** Cada mensaje recibido se reenvía a todos
7. **Desconexión:** Al cerrar, se elimina del conjunto y se notifica

## 4. Implementación del Cliente

El cliente es una aplicación JavaFX que proporciona una interfaz gráfica intuitiva para conectarse al servidor y participar en el chat.

### 4.1. Clase App

La clase `App` es el punto de entrada de la aplicación JavaFX y gestiona el ciclo de vida de la interfaz gráfica.

```
1  public class App extends Application {
2      private static boolean servidorIniciado = false;
3
4      @Override
5      public void start(Stage stage) throws IOException {
6          if (!servidorIniciado) {
7              new Thread(() ->
8                  com.chat.chatmulticlientes.servicios.Servidor
9                      .main(null)
10                 ).start();
11             servidorIniciado = true;
12             try {
13                 Thread.sleep(300);
14             } catch (InterruptedException e) {
15             }
16
17             TextInputDialog dialog = new TextInputDialog();
18             dialog.setTitle("Chat");
19             dialog.setContentText("Tu nombre:");
20
21             String nombre = dialog.showAndWait().orElse("").trim();
22             if (nombre.isEmpty()) {
23                 return;
24             }
25
26             FXMLLoader loader = new FXMLLoader(
27                 App.class.getResource("index.fxml"));
28             Scene scene = new Scene(loader.load(), 360, 640);
29             MainController controller = loader.getController();
30             controller.setNombreCliente(nombre);
31             controller.conectarAlServidor();
32
33             stage.setTitle(nombre);
34             stage.setScene(scene);
35             stage.setResizable(false);
36             stage.setOnCloseRequest(e -> controller.desconectar());
37             stage.show();
38         }
39     }
```

---

Listing 3: Clase principal App

## 4.2. Características del Cliente

### 4.2.1. Inicio Automático del Servidor

Para facilitar las pruebas, la primera instancia de la aplicación inicia automáticamente el servidor en un hilo separado. Esto permite ejecutar múltiples clientes desde la misma máquina sin configuración adicional.

### 4.2.2. Diálogo de Identificación

Al iniciar, se solicita al usuario que introduzca su nombre mediante un `TextInputDialog`. Este nombre se envía al servidor como primer mensaje y se utiliza para identificar los mensajes del usuario.

### 4.2.3. Carga de la Interfaz

La interfaz se carga desde un archivo FXML (`index.fxml`), lo que separa la estructura de la UI del código lógico y facilita el diseño visual.

## 4.3. Clase MainController

El controlador gestiona todos los eventos de la interfaz y la comunicación con el servidor.

```
1  public class MainController {
2      @FXML
3      private VBox contenedor_chat_cuerpo;
4      @FXML
5      private TextField input_mensaje;
6      @FXML
7      private ScrollPane chat;
8      @FXML
9      private Label label_titulo_chat;
10
11     private Socket socket;
12     private PrintWriter salida;
13     private String nombreCliente;
14
15     public void setNombreCliente(String nombre) {
16         this.nombreCliente = nombre;
17         label_titulo_chat.setText(nombre);
18     }
19
20     public void conectarAlServidor() {
21         try {
22             socket = new Socket("localhost", 8080);
23             salida = new PrintWriter(socket.getOutputStream(),
true);
```

```

24     BufferedReader entrada = new BufferedReader(
25         new InputStreamReader(socket.getInputStream()))
26         ;
27     salida.println(nombreCliente);
28
29     new Thread(() -> {
30         try {
31             String msg;
32             while ((msg = entrada.readLine()) != null)
33             {
34                 String m = msg;
35                 Platform.runLater(() -> agregarMensaje(
36                     m));
37             }
38         } catch (IOException e) {
39             }
40         }).start();
41     } catch (IOException e) {
42         agregarMensaje("Error: " + e.getMessage());
43     }
44 }

```

Listing 4: MainController - Variables y métodos principales

## 4.4. Gestión de Mensajes

### 4.4.1. Envío de Mensajes

Cuando el usuario presiona Enter o hace clic en el botón de enviar, se ejecuta el método `enviarMensaje()`:

```

1 @FXML
2 private void enviarMensaje() {
3     String msg = input_mensaje.getText().trim();
4     if (!msg.isEmpty() && salida != null) {
5         salida.println("[" + nombreCliente.toUpperCase() + "]":
6             " + msg);
7         agregarMensajePropio(msg);
8         input_mensaje.clear();
9     }

```

Listing 5: Método de envío de mensajes

### 4.4.2. Recepción de Mensajes

Un hilo separado escucha continuamente los mensajes entrantes del servidor. Para actualizar la interfaz de forma segura, se utiliza `Platform.runLater()`:

```

1 new Thread(() -> {
2     try {

```

```
3     String msg;
4     while ((msg = entrada.readLine()) != null) {
5         String m = msg;
6         Platform.runLater(() -> agregarMensaje(m));
7     }
8 } catch (IOException e) {
9 }
10 }).start();
```

Listing 6: Recepción asíncrona de mensajes

## 5. Protocolo de Comunicación

El protocolo implementado es simple pero efectivo, basado en el intercambio de líneas de texto sobre sockets TCP/IP.

### 5.1. Formato de Mensajes

Los mensajes siguen una estructura específica:

- **Identificación:** Primera línea enviada por el cliente contiene el nombre de usuario
- **Mensajes de chat:** Formato [NOMBRE] : contenido del mensaje
- **Notificaciones:** Usuario se ha unido o Usuario se ha ido
- **Comando de salida:** salir para cerrar la conexión

### 5.2. Flujo de Comunicación

1. Cliente se conecta al servidor en el puerto 8080
2. Cliente envía su nombre como primera línea
3. Servidor añade el cliente al conjunto de participantes
4. Servidor notifica a todos: Usuario se ha unido
5. Cliente puede enviar mensajes que se redistribuyen a todos
6. Al cerrar, servidor notifica: Usuario se ha ido

### 5.3. Diagrama de Secuencia

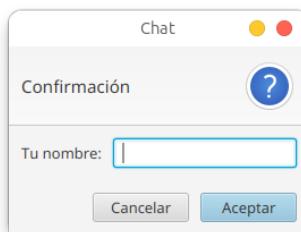
```
Cliente → Servidor: Conexión
Cliente → Servidor: "Juan"
Servidor → Todos: "Juan se ha unido"
Cliente → Servidor: "[JUAN]: Hola a todos"
Servidor → Todos: "[JUAN]: Hola a todos"
Cliente → Servidor: "salir"
Servidor → Todos: "Juan se ha ido"
```

Figura 2: Secuencia de comunicación típica

## 6. Interfaz de Usuario

La interfaz gráfica está diseñada para ser simple e intuitiva, proporcionando una experiencia de chat similar a aplicaciones de mensajería modernas.

En primera instancia, se solicita al usuario su nombre de usuario:



Al introducir un nombre, se muestra la ventana principal del chat, en la que aparece el mensaje de bienvenida.

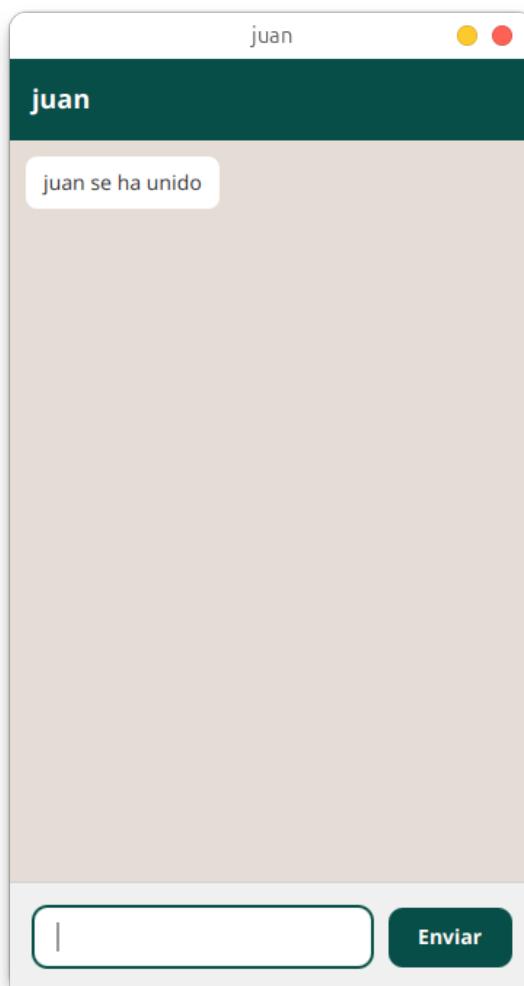


Figura 3: Interfaz de inicio del cliente

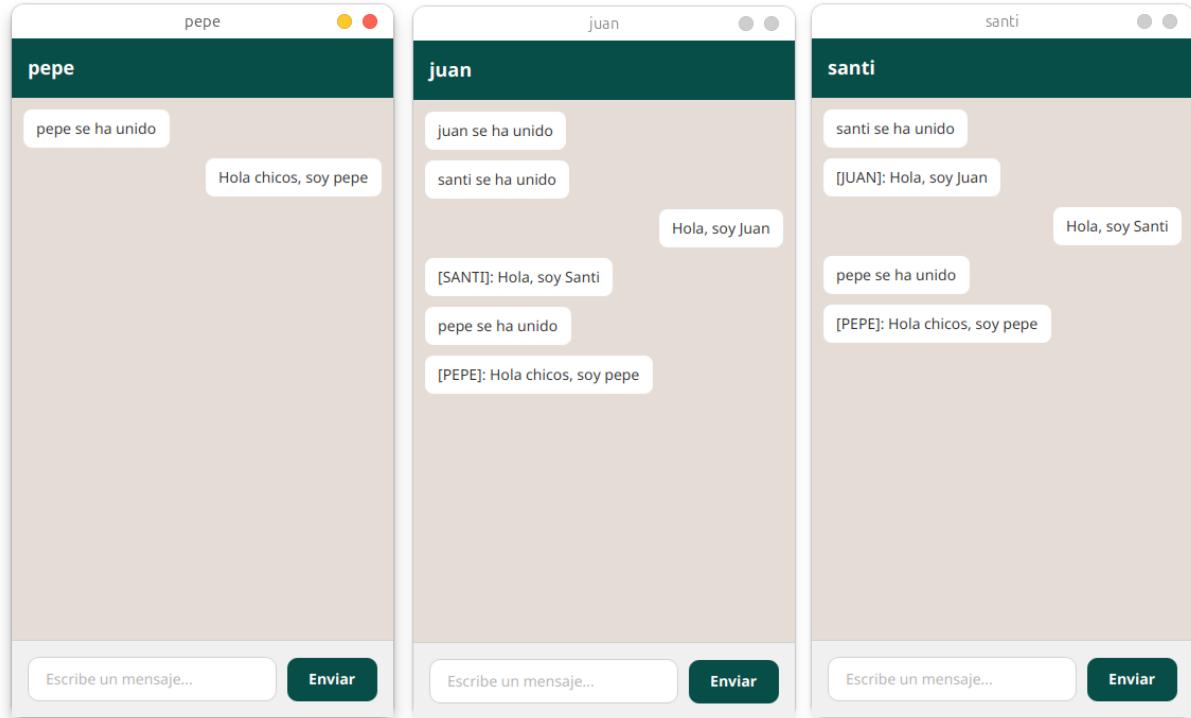


Figura 4: Tres usuarios simultáneos



Figura 5: Botón rojo para cerrar

## 6.1. Componentes de la Interfaz

La interfaz se compone de los siguientes elementos:

- **Etiqueta de título:** Muestra el nombre del usuario
- **Área de mensajes:** ScrollPane con VBox que contiene todos los mensajes
- **Campo de entrada:** TextField para escribir mensajes
- **Diferenciación visual:** Mensajes propios a la derecha, recibidos a la izquierda

## 6.2. Archivo FXML

La estructura de la interfaz se define en el archivo `index.fxml`, que describe la jerarquía de componentes JavaFX.

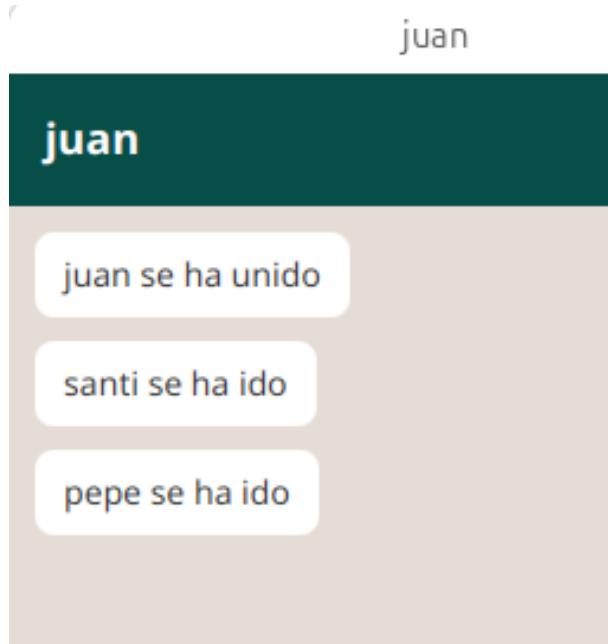


Figura 6: Usuarios saliendo del chat

### 6.3. Estilos CSS

El archivo `styles.css` define los estilos visuales de la aplicación:

- Estilos para mensajes enviados: alineados a la derecha
- Estilos para mensajes recibidos: alineados a la izquierda
- Colores y tipografía coherentes
- Diseño responsive que se adapta al contenido

### 6.4. Métodos de Visualización

#### 6.4.1. Agregar Mensaje Propio

```

1 private void agregarMensajePropio(String texto) {
2     Label label = new Label(texto);
3     label.setWrapText(true);
4     label.getStyleClass().add("mensaje");
5     label.getStyleClass().add("mensaje_enviado");
6
7     HBox contenedor = new HBox(label);
8     contenedor.setAlignment(Pos.CENTER_RIGHT);
9     contenedor_chat_cuerpo.getChildren().add(contenedor);
10    chat.setVvalue(1.0); // Scroll al final
11 }
```

Listing 7: Visualización de mensajes enviados

#### 6.4.2. Agregar Mensaje Recibido

```
1 private void agregarMensaje(String texto) {  
2     if (texto.startsWith("[ " + nombreCliente.toUpperCase() + "  
3         ]: ")) {  
4         return; // Ignorar mensajes propios  
5     }  
6     Label label = new Label(texto);  
7     label.setWrapText(true);  
8     label.getStyleClass().add("mensaje");  
9     label.getStyleClass().add("mensaje_recibido");  
10    HBox contenedor = new HBox(label);  
11    contenedor.setAlignment(Pos.CENTER_LEFT);  
12    contenedor_chat_cuerpo.getChildren().add(contenedor);  
13    chat.setVvalue(1.0);  
14 }
```

Listing 8: Visualización de mensajes recibidos

## 7. Compilación y Ejecución

El proyecto utiliza Maven como herramienta de gestión de dependencias y construcción.

### 7.1. Estructura del POM

El archivo `pom.xml` define las dependencias y plugins necesarios:

- **JavaFX Controls:** Para componentes de interfaz gráfica
- **JavaFX FXML:** Para cargar archivos FXML
- **Maven Compiler Plugin:** Configurado para Java 21
- **JavaFX Maven Plugin:** Para ejecutar la aplicación

### 7.2. Comandos de Ejecución

#### 7.2.1. Compilar el Proyecto

```
mvn clean compile
```

#### 7.2.2. Ejecutar la Aplicación

```
mvn clean javafx:run
```

#### 7.2.3. Ejecutar Múltiples Instancias

Para probar el chat, se pueden ejecutar múltiples instancias del comando anterior en terminales diferentes. Cada instancia actuará como un cliente separado.

## 7.3. Clase Launcher

La clase `Launcher` proporciona un punto de entrada alternativo que facilita la ejecución directa:

```
1 public class Launcher {  
2     public static void main(String[] args) {  
3         Application.launch(App.class, args);  
4     }  
5 }
```

Listing 9: Clase Launcher

## 8. Problemas Encontrados y Soluciones

Durante el desarrollo del proyecto se enfrentaron diversos desafíos técnicos que requirieron soluciones específicas.

### 8.1. Problema 1: Sincronización de Hilos

**Descripción:** Inicialmente se utilizaba un `ArrayList` para almacenar los clientes, lo que causaba `ConcurrentModificationException` cuando múltiples hilos intentaban modificar la lista simultáneamente.

**Solución:** Se reemplazó el `ArrayList` por `ConcurrentHashMap.newKeySet()`, que es thread-safe y permite operaciones concurrentes sin sincronización explícita.

### 8.2. Problema 2: Actualización de la Interfaz desde Hilos

**Descripción:** Al intentar actualizar la interfaz JavaFX directamente desde el hilo de recepción de mensajes, se producían excepciones porque JavaFX requiere que todas las actualizaciones de UI se hagan desde el hilo de aplicación JavaFX.

**Solución:** Se utilizó `Platform.runLater()` para encolar las actualizaciones de interfaz en el hilo correcto:

```
Platform.runLater(() → agregarMensaje(mensaje));
```

### 8.3. Problema 3: Cierre de Conexiones

**Descripción:** Al cerrar una ventana de cliente, el servidor no siempre detectaba la desconexión inmediatamente, dejando referencias a clientes inactivos.

**Solución:** Se implementó un bloque `finally` en `HandlerCliente` que garantiza la limpieza de recursos:

```
1  finally {
2      Servidor.clientes.remove(this);
3      if (nombre != null) {
4          Servidor.broadcast(nombre + " se ha ido");
5      }
6      try {
7          socket.close();
8      } catch (IOException ignored) {
9      }
10 }
```

### 8.4. Problema 4: Mensajes Duplicados

**Descripción:** Los clientes veían sus propios mensajes duplicados: uno al enviar y otro cuando el servidor los redistribuía.

**Solución:** Se implementó un filtro en `agregarMensaje()` que ignora los mensajes que comienzan con el nombre del cliente actual:

```
1 if (texto.startsWith("[" + nombreCliente.toUpperCase() + "]": ""))
2 {
3     return;
```

## 8.5. Problema 5: Configuración del Puerto

**Descripción:** En algunos sistemas, el puerto 8080 ya estaba en uso por otras aplicaciones (como servidores web de desarrollo).

**Solución:** Se documentó el puerto como constante configurable y se proporcionó información sobre cómo liberar puertos en el archivo `liberarPuertos.md`.

## 8.6. Problema 6: Scroll Automático

**Descripción:** Al recibir nuevos mensajes, el área de chat no se desplazaba automáticamente hacia abajo, obligando al usuario a hacer scroll manualmente.

**Solución:** Se añadió `chat.setVvalue(1.0)` después de agregar cada mensaje para posicionar el scroll al final automáticamente.

## 9. Conclusión

El desarrollo de este sistema de chat multicliente ha permitido poner en práctica conceptos fundamentales de programación concurrente, comunicación en red y desarrollo de interfaces gráficas.

### 9.1. Objetivos Alcanzados

Se han cumplido satisfactoriamente todos los objetivos planteados inicialmente:

- Se implementó un servidor robusto capaz de gestionar múltiples conexiones simultáneas mediante un pool de hilos
- Se desarrolló un cliente con interfaz gráfica intuitiva utilizando JavaFX
- Se estableció un protocolo de comunicación simple pero efectivo basado en sockets TCP/IP
- Se aplicaron correctamente conceptos de programación concurrente para garantizar la sincronización
- Se implementó un sistema de broadcast funcional que permite la comunicación grupal
- Se gestionaron adecuadamente los eventos de conexión y desconexión de usuarios

### 9.2. Aprendizajes Técnicos

Durante el desarrollo del proyecto se adquirieron conocimientos valiosos:

1. **Programación concurrente:** Comprensión profunda del uso de `ExecutorService`, gestión de hilos y estructuras de datos thread-safe como `ConcurrentHashMap`
2. **Comunicación en red:** Experiencia práctica con sockets TCP/IP, flujos de entrada/salida y manejo de conexiones de red
3. **JavaFX:** Desarrollo de interfaces gráficas modernas, separación de vista y lógica mediante FXML, y gestión correcta del hilo de aplicación
4. **Gestión de recursos:** Importancia del cierre correcto de conexiones y limpieza de recursos para evitar fugas de memoria
5. **Manejo de errores:** Implementación de estrategias robustas para gestionar desconexiones inesperadas y errores de red

### 9.3. Posibles Mejoras Futuras

Aunque el sistema cumple con los requisitos funcionales, existen oportunidades de mejora:

- **Persistencia de mensajes:** Almacenar el historial de chat en una base de datos

- **Salas múltiples:** Permitir crear y unirse a diferentes salas de chat
- **Mensajes privados:** Implementar comunicación directa entre dos usuarios
- **Autenticación:** Añadir sistema de registro e inicio de sesión
- **Cifrado:** Implementar comunicación segura mediante SSL/TLS
- **Indicadores de estado:** Mostrar qué usuarios están escribiendo o conectados
- **Compartir archivos:** Permitir el intercambio de imágenes y documentos
- **Emojis y formato:** Añadir soporte para emoticonos y formato de texto

#### **9.4. Reflexión Personal**

Este proyecto ha resultado muy enriquecedor desde el punto de vista educativo. La implementación de un sistema cliente-servidor funcional desde cero ha permitido comprender en profundidad cómo funcionan las aplicaciones de mensajería que utilizamos diariamente.

El mayor desafío fue gestionar correctamente la concurrencia y evitar condiciones de carrera entre hilos. La solución mediante estructuras thread-safe y el uso de `Platform.runLater()` para actualizar la interfaz fueron aprendizajes clave.

La arquitectura modular adoptada facilitó enormemente el proceso de desarrollo y depuración, permitiendo probar y validar cada componente de forma independiente antes de integrarlos.

En conclusión, el proyecto cumple con su propósito educativo y proporciona una base sólida para desarrollar aplicaciones más complejas de comunicación en red.

## 10. Bibliografía

- **Documentación oficial de Java:**

Oracle Java SE 21 Documentation

- **Tutorial de JavaFX:**

Getting Started with JavaFX

- **Documentación de Maven:**

Apache Maven Documentation

- **Java Networking Tutorial:**

Oracle Networking Tutorial

- **Java Concurrency in Practice:**

Brian Goetz et al. - Libro de referencia sobre concurrencia en Java

- **Socket Programming en Java:**

Baeldung - Guide to Java Sockets

- **ExecutorService y Thread Pools:**

Baeldung - ExecutorService Tutorial

- **JavaFX FXML:**

Introduction to FXML

- **ConcurrentHashMap en Java:**

Baeldung - ConcurrentHashMap Guide

- **Platform.runLater() en JavaFX:**

JavaFX Platform API Documentation

- **TCP/IP Protocol:**

RFC 793 - Transmission Control Protocol

- **Maven JavaFX Plugin:**

OpenJFX Maven Plugin GitHub