Instituto Politécnico Nacional

Escuela Superior de Computo

Practica 3

SERVICIO DE CHAT

MATERIA

Aplicaciones para comunicaciones de red

PROFESOR

Moreno Cervantes Axel Ernesto

GRUPO

6CM2

ALUMNOS

- Guevara Badillo Areli Alejandra
- Ramírez Martínez Alejandro

FECHA

sábado, 12 de abril de 2025

INTRODUCCIÓN

El desarrollo de aplicaciones de mensajería representa un ejercicio fundamental en la comprensión de arquitecturas cliente-servidor, programación concurrente, interfaces gráficas y manejo de archivos. En esta práctica se implementó una aplicación de chat en Java que permite a los usuarios interactuar dentro de diferentes salas virtuales, enviar mensajes públicos y privados, así como compartir archivos entre ellos.

OBJETIVOS

- Implementar una aplicación de chat funcional con múltiples salas de conversación.
- Permitir el envío y recepción de mensajes públicos y privados.
- Incorporar la funcionalidad para enviar archivos entre usuarios o al grupo.
- Gestionar las conexiones y desconexiones de usuarios de forma dinámica.

DESARROLLO

La arquitectura de la aplicación de chat se basa en el envío y recepción de objetos serializados a través de una red usando el protocolo UDP multicast. Esto permite que múltiples clientes se suscriban a un mismo grupo y reciban mensajes de forma simultánea.

La aplicación está dividida en varios módulos, cada uno con responsabilidades claras para garantizar el funcionamiento del chat.

Cada archivo de la práctica cumple una función específica:

- **Mensaje.java:** Encapsula la información (texto y archivos) para su transmisión.
- **Main.java:** Inicia la aplicación y captura el nombre del usuario.
- **SalaSelector.java:** Permite seleccionar o crear salas y gestiona la persistencia de la información.
- **Interfaz.java:** Proporciona la interfaz principal para la interacción, iniciando la conexión con el servidor de chat.
- **Cliente.java:** Es el núcleo de la comunicación, estableciendo la conexión multicast, gestionando los hilos de escucha y envío, y facilitando la transmisión de mensajes y archivos.

Las partes más importantes giran en torno a la configuración y el manejo de la conexión multicast en Cliente.java, la cual se ejecuta a través de la creación del socket, el uso de joinGroup(), la recepción y el envío de mensajes a través de datagramas UDP. Todo el diseño se enfoca en garantizar una comunicación en tiempo real, eficiente y escalable para múltiples participantes en la misma sala.

ARCHIVO PRINCIPAL (MAIN.JAVA)

Main.java es el punto de entrada de la aplicación. Su función principal es solicitar al usuario el ingreso de un nombre, validarlo y, si es correcto, abrir la ventana del selector de salas. Aquí se define la primera interfaz de usuario.

Captura de Usuario y Validación:

Se utiliza un JTextField para que el usuario introduzca su nombre y un JButton cuya acción verifica que el campo no esté vacío. Si se ingresa un nombre, se procede a ocultar la ventana principal y a instanciar SalaSelector.

```
botonConectar.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if (!campoUsuario.getText().equals("")) {
            setVisible(false);
            new SalaSelector(campoUsuario.getText().trim());
        } else {
            JOptionPane.showMessageDialog(Main.this, "Nombre de usuario vacío", "Error",
JOptionPane.ERROR_MESSAGE);
```

```
        }
    }
});
```

Flujo de la Aplicación:

Este archivo establece el punto de partida para la navegación dentro de la aplicación, permitiendo que el usuario se identifique y posteriormente seleccione o cree una sala de chat.

SALASELECTOR.JAVA

El archivo SalaSelector.java gestiona la selección o creación de salas de chat. Proporciona una interfaz en la que el usuario puede escoger una sala existente (cargada desde un archivo de texto salas.txt) o escribir una nueva sala. Al confirmar, esta ventana se cierra y se abre la interfaz principal del chat.

Carga y Persistencia de Salas:

Se emplea el método cargarSalas() para leer el archivo salas.txt y obtener la lista de salas previamente guardadas. Si el usuario ingresa una sala nueva, se guarda en este archivo mediante el método guardarSala().

```
private List<String> cargarSalas() {
    List<String> salas = new ArrayList<>();
    File archivo = new File("salas.txt");
    if (archivo.exists()) {
        try (BufferedReader br = new BufferedReader(new FileReader(archivo))) {
            String linea;
            while ((linea = br.readLine()) != null) {
                if (!linea.trim().isEmpty()) salas.add(linea.trim());
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    return salas;
}
```

Transición a la Interfaz de Chat:

Una vez confirmada la sala (seleccionada o escrita), se procede a cerrar la ventana actual y a abrir la ventana de chat (Interfaz.java), pasando los parámetros de conexión como la dirección multicast y el puerto.

INTERFAZ.JAVA

Interfaz.java define la ventana principal del chat. Aquí se integran los componentes gráficos para mostrar mensajes en formato HTML, enviar textos, gestionar archivos y finalizar la conexión. Este archivo es responsable de instanciar el cliente que realizará la comunicación multicast.

Componentes Gráficos y Organización:

La interfaz cuenta con un JEditorPane para mostrar mensajes, un JTextArea para introducir nuevos mensajes, botones para enviar mensajes, adjuntar archivos y salir de la sala. La organización se realiza mediante paneles (JPanel) que facilitan la disposición de estos elementos.

Inicialización del Cliente Mediante un Listener:

Se utiliza un WindowListener (clase interna CorreCliente) para iniciar la conexión al abrir la ventana. Aquí se instancia la clase Cliente, que establecerá la comunicación multicast, y se envía un saludo notificando la conexión del usuario.

```
private class CorreCliente extends WindowAdapter {
    public void windowOpened(WindowEvent we) {
        miCliente = new Cliente(nombre, host, puerto, editor, usuarioConectado, sala);
        miCliente.saludo(nombre);
    }

    public void windowClosing(WindowEvent e) {
        if (miCliente != null) {
            miCliente.despedida(nombre);
            miCliente.cerrarConexion();
        }
        System.exit(0);
    }
}
```

CLIENTE.JAVA

El archivo Cliente.java es el núcleo de la aplicación en lo que se refiere a la conexión y comunicación en red. Se encarga de establecer la conexión multicast, enviar y recibir mensajes (y archivos) mediante UDP, y gestionar la lógica de la aplicación en cuanto a la interacción entre clientes.

Establecimiento de la Conexión Multicast

La conexión se establece en el constructor de la clase Cliente. Se crea un MulticastSocket, se obtiene la dirección del grupo multicast (por ejemplo, 230.1.1.1) y se une el cliente al grupo mediante el método joinGroup().

```
try {
    // Creación del socket multicast
    cliente = new MulticastSocket(puerto);
    // Obtención de la dirección IP del grupo multicast
    grupo = InetAddress.getByName(host);
    // El cliente se une al grupo multicast para recibir los mensajes
    cliente.joinGroup(grupo);
} catch (Exception e) {
    e.printStackTrace();
}
```

Importancia:

Este bloque es crucial, ya que permite la comunicación en grupo. Al unirse al grupo multicast, el cliente puede enviar y recibir mensajes de todos los participantes de la sala.

Hilo de Escucha Multicast

Una vez establecida la conexión, se lanza un hilo (implementado en la clase interna EscuchaMensajes) que se encarga de recibir continuamente los mensajes enviados al grupo. Este hilo utiliza un DatagramPacket para captar los datagramas y luego deserializarlos para obtener el objeto Mensaje.

```
private class EscuchaMensajes implements Runnable {
    public void run() {
        System.out.println("Escuchando Mensajes");
```

```
        try {
            DatagramPacket recibido = new DatagramPacket(new byte[6500], 6500);
            while (true) {
                // Recibe un paquete del grupo multicast
                cliente.receive(recibido);
                ObjectInputStream ois = new ObjectInputStream(new
ByteArrayInputStream(recibido.getData()));
                Mensaje msj = (Mensaje) ois.readObject();

                // Filtrar mensajes de salas distintas
                if (!sala.equals(msj.getSala())) {
                    continue;
                }

                switch (msj.getTipo()) {
                    case 0:
                        // Saludo al conectarse
                        if (!msj.getUsuarioOrigen().equals(nombre)) {
                            mostrar(msj.getMensaje());
                            usuarioConectado.addItem(msj.getUsuarioOrigen());
                            Mensaje respuesta = new Mensaje("", nombre,
msj.getUsuarioOrigen(), 3, sala);
                            enviar(respuesta);
                        }
                        break;
                    case 1:
                        // Mensaje público
                        mostrar(msj.getMensaje());
                        break;
                    case 4:
                        // Mensaje privado
                        if (msj.getUsuarioDestino().equals(nombre)) {
                            mostrar("[Privado]: " + msj.getMensaje());
                        }
                        break;
                    case 5:
                        // Despedida
                        mostrar(msj.getMensaje());
                        break;
                }
                ois.close();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
```

```
    }
}
```

Importancia:

El hilo de escucha es fundamental para mantener la comunicación en tiempo real. Permite que cada cliente reciba y procese los mensajes enviados a la sala, asegurando que la aplicación funcione de manera interactiva y sin bloqueos.

Envío de Mensajes Multicast

Para enviar mensajes, se serializa un objeto Mensaje y se encapsula en un DatagramPacket, el cual se envía a través del socket multicast. Esto se realiza en la clase interna EnviaMensajes.

```
private class EnviaMensajes implements Runnable {
    private Mensaje msj;

    public EnviaMensajes(Mensaje msj) {
        this.msj = msj;
    }

    public void run() {
        try {
            // Serialización del objeto Mensaje
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            ObjectOutputStream oos = new ObjectOutputStream(baos);
            oos.writeObject(msj);
            oos.flush();
            byte[] msjBytes = baos.toByteArray();

            // Construcción del DatagramPacket y envío mediante multicast
            DatagramPacket p = new DatagramPacket(msjBytes, msjBytes.length, grupo,
puerto);
            cliente.send(p);

            oos.close();
            baos.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Importancia:

Este mecanismo permite que los mensajes se distribuyan a todos los clientes conectados a la sala de manera eficiente y simultánea. La serialización y envío mediante MulticastSocket es la base de la comunicación en este sistema.

Envío y Recepción de Archivos

Para el manejo de archivos, se fragmenta el archivo en bloques (por ejemplo, bloques de 4000 bytes) y se envía cada fragmento como un objeto Mensaje. En el receptor, los fragmentos se acumulan y se reconstruye el archivo.

```
while (enviado < tamanio) {
    Mensaje datos = new Mensaje(
        file.getName(), nombre, destinatario, 2, file.length(), "", ++i, sala
    );
    ByteArrayOutputStream baos = new ByteArrayOutputStream(6400);
    ObjectOutputStream oos = new ObjectOutputStream(new BufferedOutputStream(baos));
    oos.flush();

    byte[] b = new byte[4000];
    n = dis.read(b);
    byte[] b2 = new byte[n];
    System.arraycopy(b, 0, b2, 0, n);
    datos.setDatos(b2);
    datos.setBytesEnviados(n);

    oos.writeObject(datos);
    oos.flush();

    byte[] d = baos.toByteArray();
    DatagramPacket paqueteEnvio = new DatagramPacket(d, d.length, grupo, puerto);
    cliente.send(paqueteEnvio);

    Thread.sleep(500);
    enviado += n;
    oos.close();
    baos.close();
}
```

Importancia:

La fragmentación y la posterior reconstrucción en el receptor permiten el manejo de archivos de tamaño considerable, superando las limitaciones de tamaño de un solo datagrama y aprovechando la comunicación multicast para compartir archivos entre usuarios.

MENSAJE.JAVA

Este archivo define la clase Mensaje, la cual sirve como contenedor de información para el intercambio de datos entre clientes. Se utiliza para encapsular tanto mensajes de texto (públicos y privados) como fragmentos de archivos. Al implementar la interfaz Serializable, los objetos de esta clase pueden transformarse en secuencias de bytes, permitiendo su envío a través de la red mediante UDP.

Constructores Diferenciados:

Hay dos constructores:

- Uno para mensajes simples, que recibe el contenido, usuario de origen, destino, tipo (por ejemplo, saludo, mensaje público o privado) y la sala a la que pertenece.
- Otro para el envío de archivos, que además incluye el nombre del archivo, el tamaño, la ruta, el número de paquete (para fragmentar el archivo), etc.

```
public Mensaje(String mensaje, String usuarioOrigen, String usuarioDestino, int tipo,
String sala) {
    this.mensaje = mensaje;
    this.usuarioOrigen = usuarioOrigen;
```

```
        this.usuarioDestino = usuarioDestino;
        this.tipo = tipo;
        this.sala = sala;
    }
```

Serialización:

La capacidad de convertir un objeto Mensaje en un arreglo de bytes es fundamental para el envío a través de UDP, ya que permite transmitir información compleja (incluyendo archivos fragmentados) sin depender de un protocolo más pesado.

PRUEBAS Y RESULTADOS

Para verificar el correcto funcionamiento de la aplicación de chat, se realizaron pruebas en un entorno de red local simulando múltiples instancias del cliente. Los casos de prueba incluyeron:

Conexión simultánea de múltiples usuarios:

Se inició la aplicación en varias instancias en el mismo equipo, permitiendo que varios usuarios (seis usuarios) ingresaran a la misma sala mediante la selección de salas existentes o la creación de nuevas.

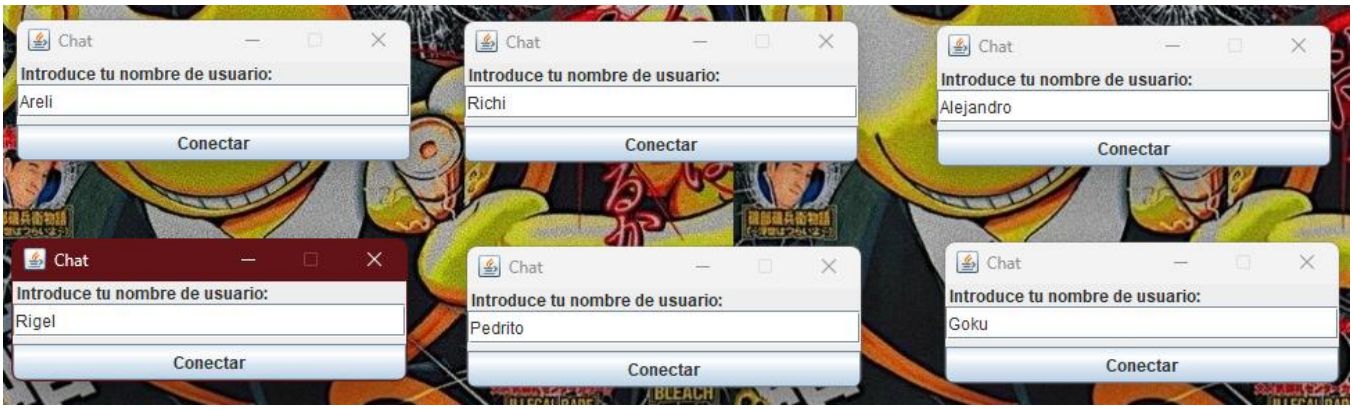

Ilustración 1. Ingreso de los usuarios.

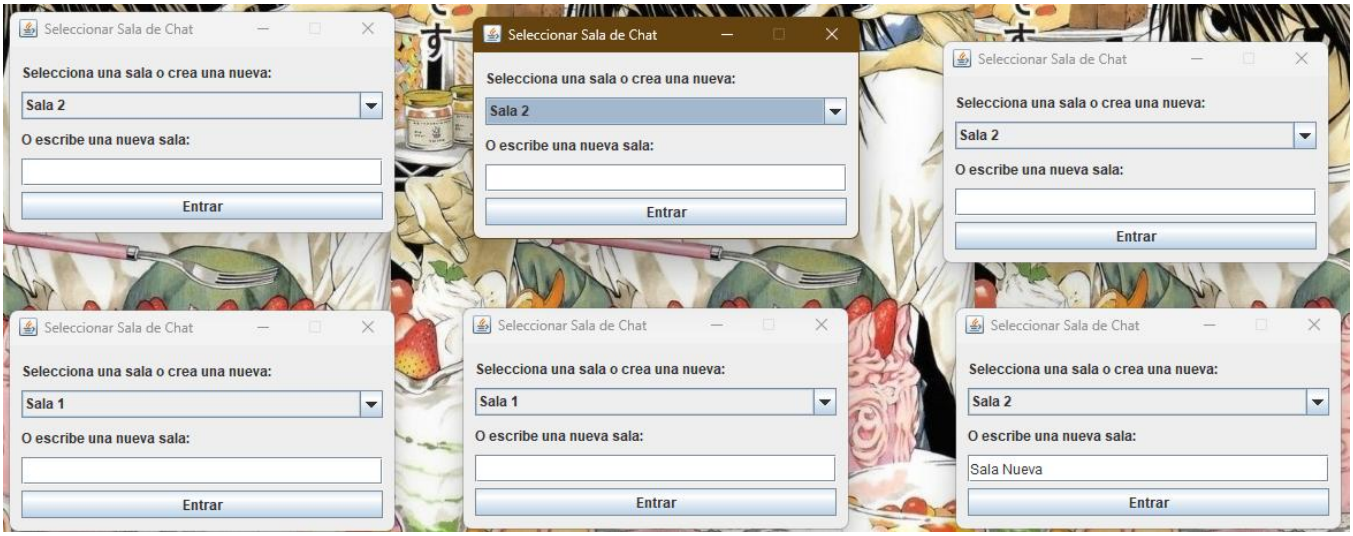

Ilustración 2. Vista de selección de salas.

Envío y recepción de mensajes públicos y privados:

Cada cliente pudo enviar mensajes que fueron recibidos en tiempo real por todos los participantes de la misma sala. Además, se probaron mensajes privados que se mostraron únicamente al destinatario correspondiente.

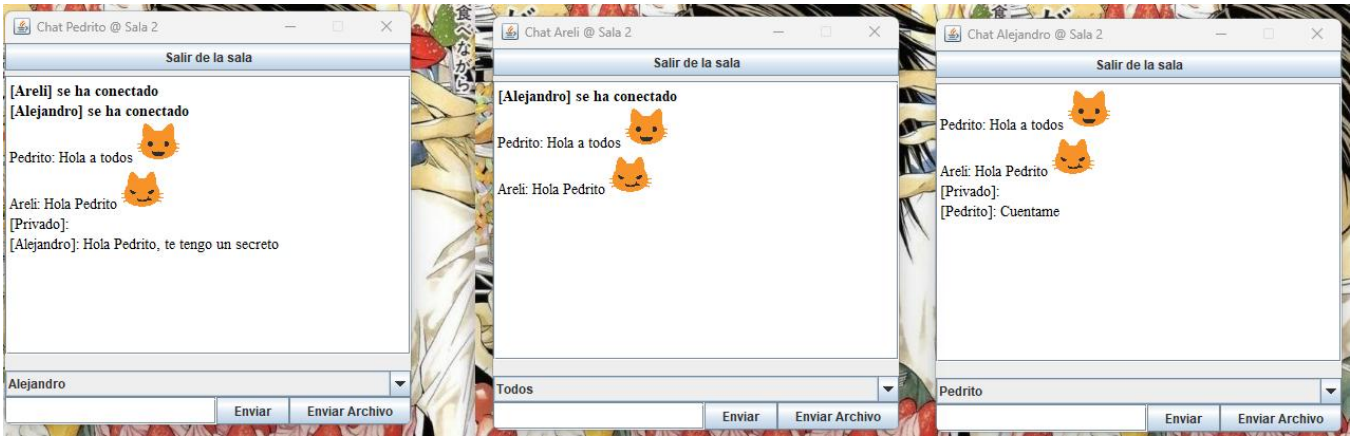

Ilustración 3. Conversación con uso de emojis y mensajes privados.

Transferencia de archivos:

Se seleccionaron archivos de diferentes tamaños para comprobar la correcta fragmentación, envío y reconstrucción de estos. Se verificó que los archivos fueran almacenados en la estructura de carpetas adecuada en el sistema local (por usuario y sala).

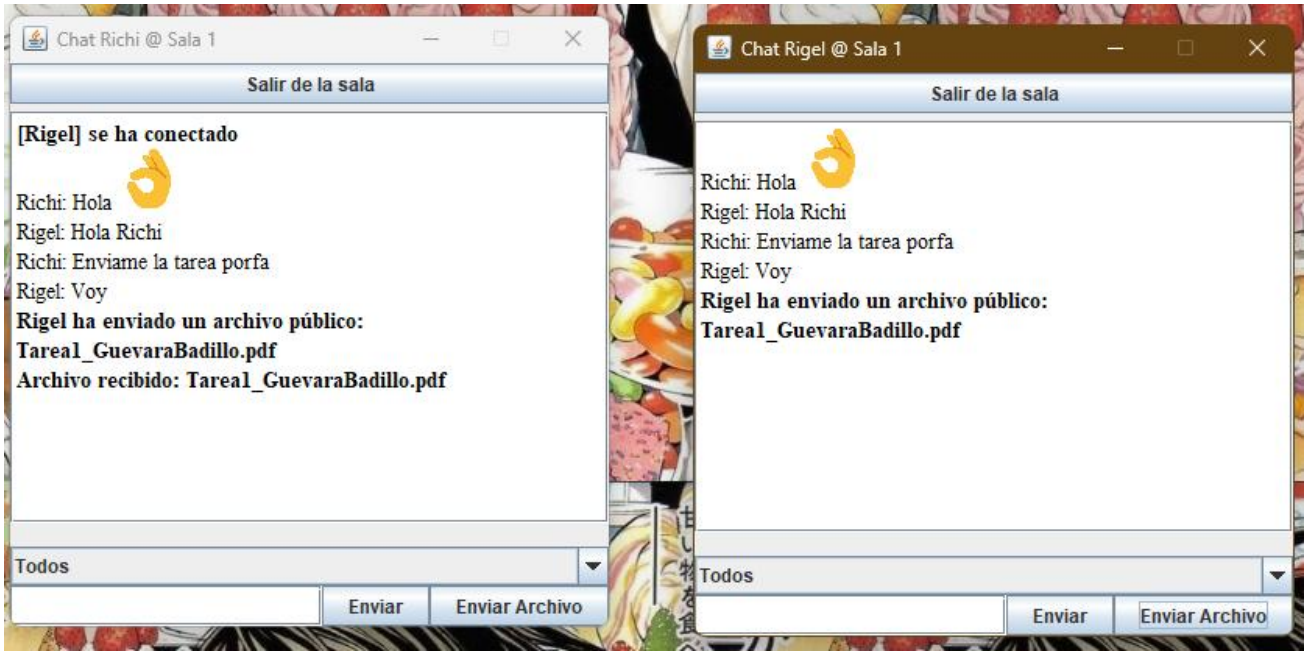

Ilustración 4. Envió de archivos.

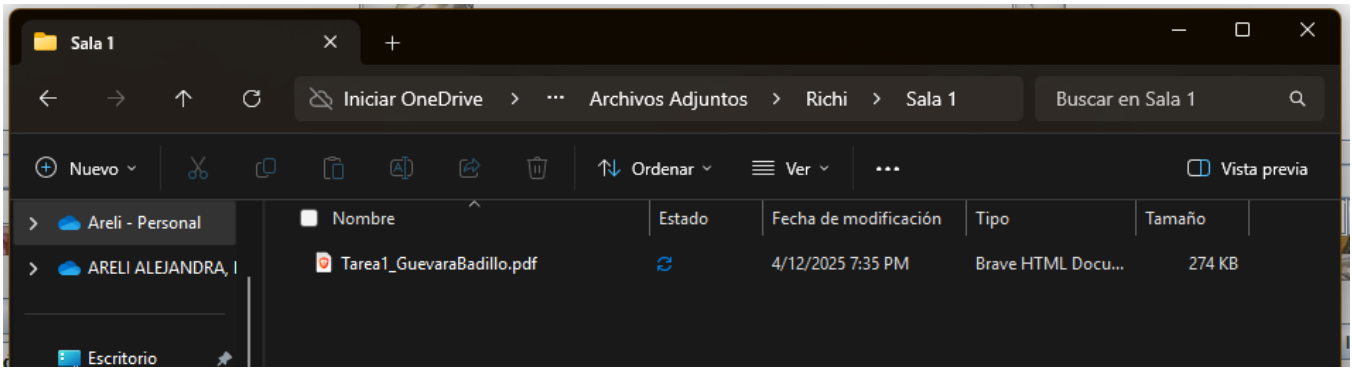

Ilustración 5. Carpeta de usuario donde se guardan los archivos recibidos.

Actualización de la lista de usuarios:

Al ingresar y salir usuarios de la sala, se observaron las notificaciones de saludo y despedida enviadas mediante la conexión multicast, y la lista de usuarios se actualizó dinámicamente en la interfaz de chat.

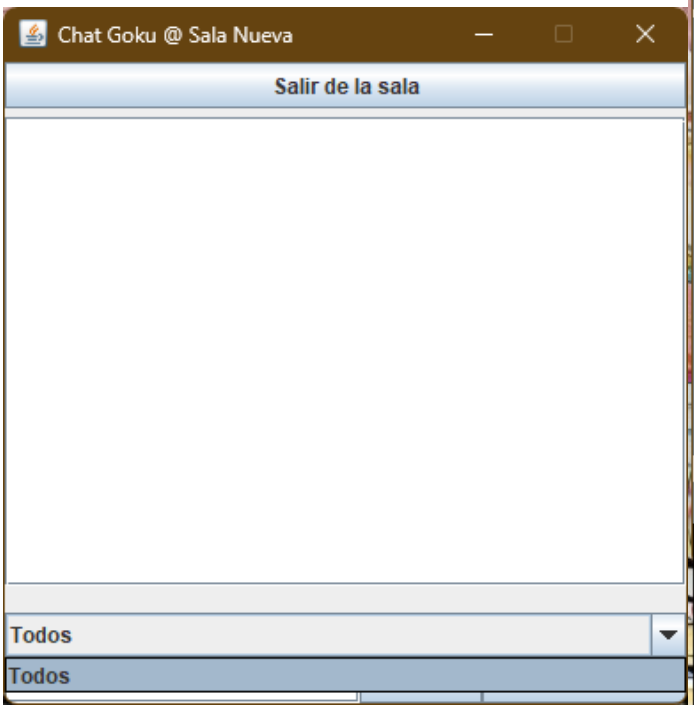

Ilustración 6. Chat vacio.

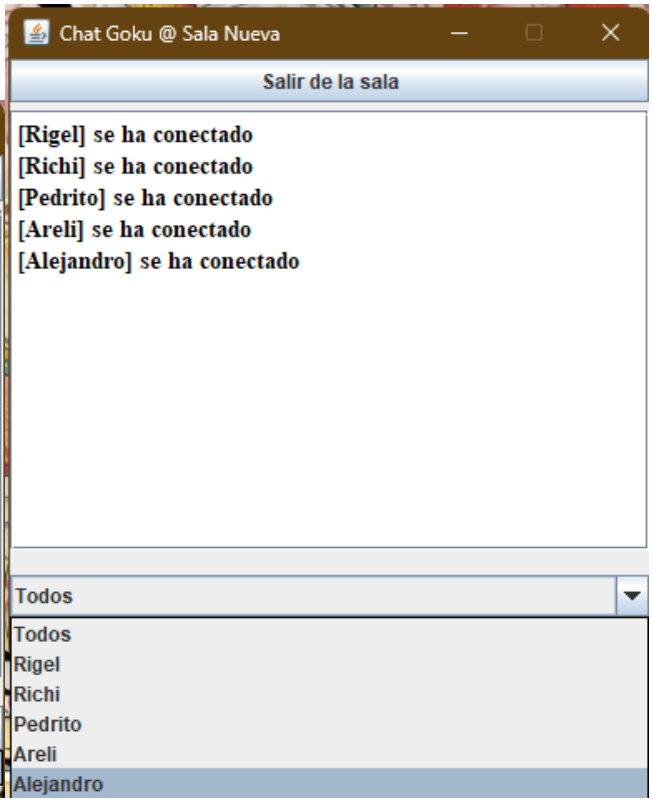

Ilustración 7. Chat con más usuarios.

Los resultados de las pruebas demostraron que:

- La conexión multicast es estable y permite la distribución efectiva de mensajes a múltiples clientes en tiempo real.
- La aplicación soporta tanto el envío de mensajes de texto (públicos y privados) como la transferencia de archivos sin pérdidas de información.
- La persistencia de datos (como la lista de salas) y la actualización dinámica de la interfaz de usuario contribuyen a una experiencia de chat funcional y confiable.
- Las pruebas prácticas indicaron que la fragmentación y reconstrucción de archivos son viables, incluso con archivos de mayor tamaño, siempre que se respeten los límites de tamaño del datagrama y se establezcan pausas controladas en el envío de paquetes.

Conclusiones

Esta práctica de crear un servicio de chat fue una experiencia muy valiosa que permitió aplicar y entender de manera práctica muchos conceptos importantes del desarrollo de aplicaciones en red. Uno de los puntos centrales fue aprender a trabajar con la comunicación multicast usando UDP; ver cómo se configura y se usa el **MulticastSocket** para enviar mensajes a varios clientes a la vez fue muy útil. Además, el uso de hilos para la escucha continua y la actualización de la interfaz demostró la importancia de manejar la concurrencia y de mantener una comunicación en tiempo real.

Durante el desarrollo, surgieron varios desafíos, como el manejo de la fragmentación y reconstrucción de archivos. Dividir un archivo en fragmentos pequeños y luego juntarlos correctamente en el otro extremo nos obligó a prestar atención al control de flujo y sincronización en la red. Otro reto fue mantener la lista de usuarios actualizada, especialmente cuando alguien se desconectaba, lo cual nos permitió profundizar en la integración entre la lógica de red y la interfaz de usuario.