Instituto Politécnico Nacional

Escuela Superior de Cómputo

DESARROLLO DE SISTEMAS DISTRIBUIDOS

Práctica 3

***Profesor**: Chadwick Carreto Arellano*

***Grupo**: 4CM11*

***Alumno**: González Hipólito Miguel Ángel*

Fecha de entrega: 11/Marzo/2025

Contents

1 Antecedentes

El uso de la tecnología Multicast resulta especialmente eficiente, ya que permite la transmisión de datos desde un servidor hacia múltiples clientes de forma simultánea. A diferencia de una conexión punto a punto (unicast), el protocolo multicast optimiza el ancho de banda al enviar un solo paquete que es replicado por la red únicamente cuando se necesita, reduciendo así la carga en el servidor.

En esta práctica, se implementó un sistema de chat multicast que permite la comunicación en tiempo real entre múltiples clientes conectados a un mismo grupo multicast.

Esta solución demuestra la versatilidad de las aplicaciones distribuidas en entornos colaborativos, donde la eficiencia en la transmisión de datos y la gestión concurrente de múltiples clientes son factores clave para el éxito del sistema.

2 Planteamiento del Problema

La comunicación en red es un aspecto fundamental en el desarrollo de aplicaciones distribuidas, especialmente en entornos donde se requiere el intercambio de mensajes y archivos entre múltiples usuarios.

El problema que se aborda en esta práctica es la implementación de un **chat grupal multicast** que permita:

- Enviar y recibir mensajes en tiempo real.
- Compartir archivos entre los usuarios.
- Visualizar la lista de archivos disponibles en el servidor.
- Descargar archivos fragmentados de forma segura, garantizando que cada fragmento sea recibido correctamente.

Este proyecto tiene como objetivo principal desarrollar un sistema de comunicación que combine la interacción en tiempo real con la transferencia segura de archivos mediante el uso de **sockets de datagrama (UDP)** y el protocolo **multicast**.

3 Propuesta de Solución

Para resolver el problema planteado, se propone el desarrollo de una aplicación que implemente un chat basado en multicast utilizando la clase MulticastSocket en Java.

El sistema estará compuesto por dos componentes principales:

Socket multicast: para compartir mensajes entre los múltiples clientes que se conecten.

Socket udp: Permite enviar compartir archivos de manera rápido.

Confirmación ack: Permite realizar una retransmión de los paquetes UDP que no pudieran llegar del cliente al servidor y viceversa.

Cliente: Permite al usuario enviar y recibir mensajes, visualizar la lista de archivos disponibles, subir archivos al servidor y descargar archivos fragmentados de forma segura.

Servidor: Se encarga de gestionar las solicitudes de los clientes, almacenar los archivos y enviar las confirmaciones de recepción (ACK) para garantizar la entrega exitosa de los fragmentos.

Se utilizarán objetos serializados para enviar y recibir datos, lo que facilitará la gestión de mensajes, archivos y confirmaciones dentro del sistema.

4 Materiales y Métodos empleados

4.1 Materiales Utilizados

- **Lenguaje de programación:** Java 11
- **Entorno de desarrollo:** VS Code
- **Sistema operativo:** Windows
- **Protocolos de red:** UDP y Multicast
- **Herramientas adicionales:** Biblioteca javax.swing para la interfaz gráfica

4.2 Métodos Empleados

Se aplicaron los siguientes principios y técnicas en la implementación de los hilos:

I. **Manejo de Sockets:** Se utilizaron DatagramSocket y MulticastSocket para establecer la comunicación en red mediante el protocolo UDP.

II. **Protocolo Multicast:** Se empleó la dirección 230.0.0.0 para la difusión de mensajes en el grupo multicast, permitiendo que varios clientes reciban la misma información de forma simultánea.

III. **Fragmentación y Control de Flujo:** Para garantizar la transferencia segura de archivos grandes, se implementó un sistema de fragmentación, donde cada fragmento se envía junto con un índice, y el servidor espera una confirmación (ACK) para cada segmento antes de continuar.

IV. **Interfaz Gráfica (GUI):** Se diseñó una interfaz amigable utilizando JFrame, JPanel, y JTextArea para facilitar la interacción del usuario con el sistema.

V. **Gestión de Errores:** Se emplearon bloques try-catch para manejar posibles fallos en la conexión o errores durante la transferencia de datos.

5 Desarrollo

5.1 Implementación del cliente

El cliente se desarrolló utilizando la clase `MulticastSocket`, la cual permite enviar y recibir mensajes dentro del grupo multicast. Además, se implementaron las siguientes funciones:

- **Enviar mensajes:** Envía un mensaje formateado que incluye el nombre del usuario.

```
private void sendMessage(String message) {
    try {
        byte[] buffer = message.getBytes();
        DatagramPacket packet = new DatagramPacket(buffer, buffer.length, group, MULTICAST_PORT);
        multicastSocket.send(packet);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

- **Enviar emojis:** Permite seleccionar un emoji en formato ASCII para integrarlo en el chat.

```
private void sendEmoji() {
    // Lista de "emojis" tipo ASCII
    String[] asciiEmojis = {
        ":-)",   // Sonrisa
        ":-D",   // Risa
        ":-(",   // Tristeza
        ":-O",   // Sorprendido
        ";-)",   // Guiño
        ":-|",   // Neutral
        ":-((",  // Llorando
        ":P",    // Saca la lengua
        ":-/",   // Dudoso
        "<3"     // Corazón
    };

    String selectedEmoji = (String) JOptionPane.showInputDialog(
        parentComponent:null,
        message:"Selecciona un emoji:",
        title:"Seleccionar Emoji",
        JOptionPane.PLAIN_MESSAGE,
        icon:null,
        asciiEmojis,
        asciiEmojis[0] // Valor por defecto
    );

    if (selectedEmoji != null && !selectedEmoji.isEmpty()) {
        sendMessage("<" + username + "><Emoji> " + selectedEmoji);
    }
}
```

- **Listar archivos:** Envía una solicitud al servidor para recibir la lista de archivos disponibles.

```
private void listFiles() {
    try {
        Objeto request = new Objeto(accion:"listar", mensaje:null, archivo:null, nombreArchivo:null);
        sendObjectToServer(request);

        Objeto response = receiveObjectFromServer();
        JOptionPane.showMessageDialog(parentComponent:null, response.getMensaje(), title:"Archivos del Servidor",
    } catch (IOException | ClassNotFoundException e) {
        e.printStackTrace();
    }
}
```

- **Subir archivos:** Divide el archivo en fragmentos de 60 KB y envía cada uno con su respectivo índice.

```java
private void uploadFile() {
    JFileChooser fileChooser = new JFileChooser();
    int returnValue = fileChooser.showOpenDialog(parent:null);

    if (returnValue == JFileChooser.APPROVE_OPTION) {
        File file = fileChooser.getSelectedFile();

        System.out.print(file.getName());

        try (FileInputStream fis = new FileInputStream(file)) {
            byte[] buffer = new byte[60000];
            int bytesRead, index = 0;

            while ((bytesRead = fis.read(buffer)) != -1) {
                Objeto fragment = new Objeto(accion:"subir", file.getName(), Arrays.copyOf(buffer, bytesRead), file.getName());
                fragment.setIndiceFragmento(index);
                fragment.setUltimoFragmento(bytesRead < buffer.length);

                sendObjectToServer(fragment);

                Objeto ack = receiveObjectFromServer();
                if (!ack.getMensaje().equals("ACK:" + index)) {
                    throw new IOException("ACK no recibido para el fragmento: " + index);
                }

                index++;
            }
            JOptionPane.showMessageDialog(parentComponent:null, message:"Archivo subido exitosamente.", title:"Subida", JOptionPane.INFORMATION_MESSAGE);
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
```

- **Descargar archivos:** Solicita al servidor un archivo específico y recibe cada fragmento con confirmación (`ACK`) para garantizar la integridad de la transferencia.

```java
private void downloadFile() {
        String fileName = JOptionPane.showInputDialog("Ingrese el nombre del archivo a descargar:");
        if (fileName == null || fileName.trim().isEmpty()) {
            return;
        }

        try {
            Objeto request = new Objeto("descargar", "", null, fileName);
            sendObjectToServer(request);

            // Ruta de la carpeta del usuario
            String userDirectoryPath = "C:\\Users\\Migue\\Documents\\GitHub\\Aplicaciones_para_comunicaciones_en_red\\Practica3_chat\\client\\" + username;
            File userDirectory = new File(userDirectoryPath);

            // Crear la carpeta del usuario si no existe
            if (!userDirectory.exists()) {
                boolean dirCreated = userDirectory.mkdirs();
                if (!dirCreated) {
                    JOptionPane.showMessageDialog(null, "No se pudo crear la carpeta para el usuario.", "Error", JOptionPane.ERROR_MESSAGE);
                    return;
                }
            }

            // Ruta completa para el archivo
            File file = new File(userDirectory, fileName);

            try (FileOutputStream fos = new FileOutputStream(file)) {
                Objeto fragment;
                int fragmentIndex = 0;
                boolean lastFragment = false;
                while (!lastFragment) {
                    fragment = receiveObjectFromServer();
                    fos.write(fragment.getArchivo());

                    // Enviar ACK después de recibir el fragmento
                    Objeto ack = new Objeto("ACK", "ACK:" + fragmentIndex, null, null);
                    sendObjectToServer(ack);

                    lastFragment = fragment.isUltimoFragmento();
                    fragmentIndex++;
                }

                JOptionPane.showMessageDialog(null, "Archivo descargado con éxito.", "Descarga", JOptionPane.INFORMATION_MESSAGE);
            }
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
```

5.2 Implementación del server

El servidor se diseñó para gestionar múltiples clientes simultáneamente. Sus principales funcionalidades son:

- **Recepción de solicitudes:** Procesa las acciones recibidas del cliente (listar, subir o descargar archivos).

```
Run main | Debug main | Run | Debug
public static void main(String[] args) {
    try {
        int pto = 8081;
        DatagramSocket s = new DatagramSocket(pto);
        InetAddress multicastGroup = InetAddress.getByName(MULTICAST_ADDRESS);
        System.out.println(x:"Servidor iniciado. Esperando clientes...");

        while (true) {
            DatagramPacket p = new DatagramPacket(new byte[65535], length:65535);
            s.receive(p);

            ObjectInputStream ois = new ObjectInputStream(new ByteArrayInputStream(p.getData()));
            Objeto recibido = (Objeto) ois.readObject();

            InetAddress clienteIP = p.getAddress();
            int clientePort = p.getPort();

            switch (recibido.getAccion()) {
                case "listar" -> {···
                case "subir" -> {···
                case "descargar" -> {···
                default -> System.out.println("Acción desconocida: " + recibido.getAccion());
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

5.3 Función listar

- Se crea un objeto File apuntando a la carpeta del servidor que almacena los archivos.
- Con carpetaServidor.list() se obtiene un arreglo de nombres de archivos.
- Si hay archivos, se genera una cadena con sus nombres separados por salto de línea (\n).
- Si no hay archivos, se envía el mensaje "No hay archivos disponibles en el servidor.".
- Finalmente, se crea un objeto Objeto con la acción "listar" y se envía como respuesta al cliente.

```java
case "listar" -> {
    File carpetaServidor = new File(pathname:"C:\\Users\\Migue\\Documents\\GitHub\\Desarrollo_aplicaciones_distribuidas\\P
    String[] archivosServidor = carpetaServidor.list();

    String listaArchivos = (archivosServidor != null && archivosServidor.length > 0)
            ? String.join(delimiter:"\n", archivosServidor)
            : "No hay archivos disponibles en el servidor.";

    Objeto respuesta = new Objeto(accion:"listar", listaArchivos, archivo:null, nombreArchivo:null);
    enviarObjeto(s, respuesta, clienteIP, clientePort);
}
```

5.4 Función subir

```
case "subir" -> {
    String nombreArchivo = recibido.getNombreArchivo();
    File archivo = new File("C:\\Users\\Migue\\Documents\\GitHub\\Desarrollo_aplicaciones_distribuidas\\Practica3_chat\\s

    try (FileOutputStream fos = new FileOutputStream(archivo, append:true)) {
        fos.write(recibido.getArchivo());
    }

    // Enviar ACK al cliente
    Objeto ack = new Objeto(accion:"ACK", "ACK:" + recibido.getIndiceFragmento(), archivo:null, nombreArchivo:null);
    enviarObjeto(s, ack, clienteIP, clientePort);

    if (recibido.isUltimoFragmento()) {
        System.out.println("Archivo " + nombreArchivo + " recibido completamente.");

        // Notificar a los clientes en el grupo multicast
        String mensaje = "<Archivo> El archivo '" + nombreArchivo + "' ha sido subido al servidor.";
        enviarMulticast(mensaje, multicastGroup);
    }
}
```

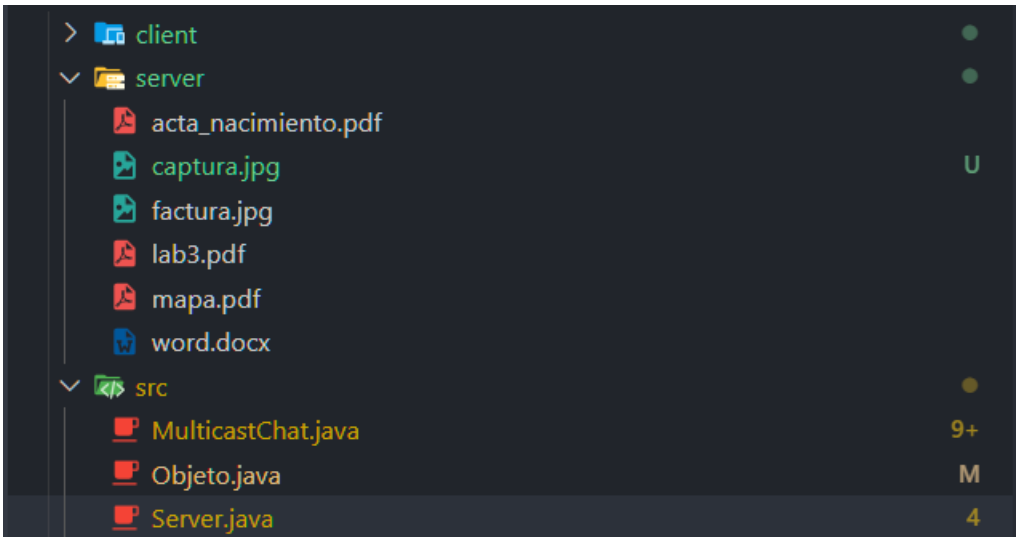
- Se obtiene el nombre del archivo desde recibido.getNombreArchivo().
- El archivo se guarda en la carpeta del servidor usando FileOutputStream.
- El modo de apertura (true) indica que se abre en modo append, es decir, si el archivo ya existe se agregará contenido al final (ideal para recibir fragmentos).
- Después de escribir el fragmento, se envía un mensaje de confirmación (ACK) con el índice del fragmento recibido.
- Si el fragmento es el último (isUltimoFragmento()), se muestra un mensaje de éxito en consola y se envía un mensaje multicast notificando a todos los clientes.

5.5 Función descargar

- El servidor verifica si el archivo solicitado existe. Si no, muestra un mensaje y no envía nada.
- El archivo se lee usando un búfer de 60,000 bytes por fragmento.
- Cada fragmento se envía en un bucle hasta que se reciba el ACK correspondiente. Si no se recibe el ACK, el fragmento se retransmite.
- El ciclo se repite hasta que se hayan enviado todos los fragmentos.
- El índice (indice) controla el orden de los fragmentos, y el atributo setUltimoFragmento(true) indica si es el último bloque del archivo.

```java
case "descargar" -> {
                        String nombreArchivo = recibido.getNombreArchivo();
                        File archivo = new File("C:\\Users\\Migue\\Documents\\GitHub\\Desarrollo_aplicaciones_distribuidas\\Practica3_chat\\
server\\" + nombreArchivo);

                        if (!archivo.exists()) {
                            System.out.println("Archivo solicitado no existe: " + nombreArchivo);
                            continue;
                        }

                        try (FileInputStream fis = new FileInputStream(archivo)) {
                            byte[] buffer = new byte[60000];
                            int bytesLeidos, indice = 0;

                            while ((bytesLeidos = fis.read(buffer)) != -1) {
                                boolean ackRecibido = false;

                                while (!ackRecibido) {
                                    Objeto fragmento = new Objeto(
                                            "descargar",
                                            "Enviando fragmento " + indice,
                                            Arrays.copyOf(buffer, bytesLeidos),
                                            nombreArchivo
                                    );
                                    fragmento.setIndiceFragmento(indice);
                                    fragmento.setUltimoFragmento(bytesLeidos < buffer.length);

                                    enviarObjeto(s, fragmento, clienteIP, clientePort);

                                    try {
                                        DatagramPacket ackPacket = new DatagramPacket(new byte[65535], 65535);
                                        s.receive(ackPacket);

                                        ObjectInputStream ackOis = new ObjectInputStream(new ByteArrayInputStream(ackPacket.getData()));
                                        Objeto ack = (Objeto) ackOis.readObject();
                                        ackRecibido = ack.getMensaje().equals("ACK:" + indice);
                                    } catch (SocketTimeoutException e) {
                                        System.out.println("Timeout esperando ACK para fragmento " + indice + ". Retransmitiendo...");
                                    }
                                }
                                indice++;
                            }
                        }
                        System.out.println("Archivo enviado: " + nombreArchivo);
                    }
```

- **Gestión de archivos:** Almacena los archivos recibidos en una carpeta específica compartida y verifica su integridad.



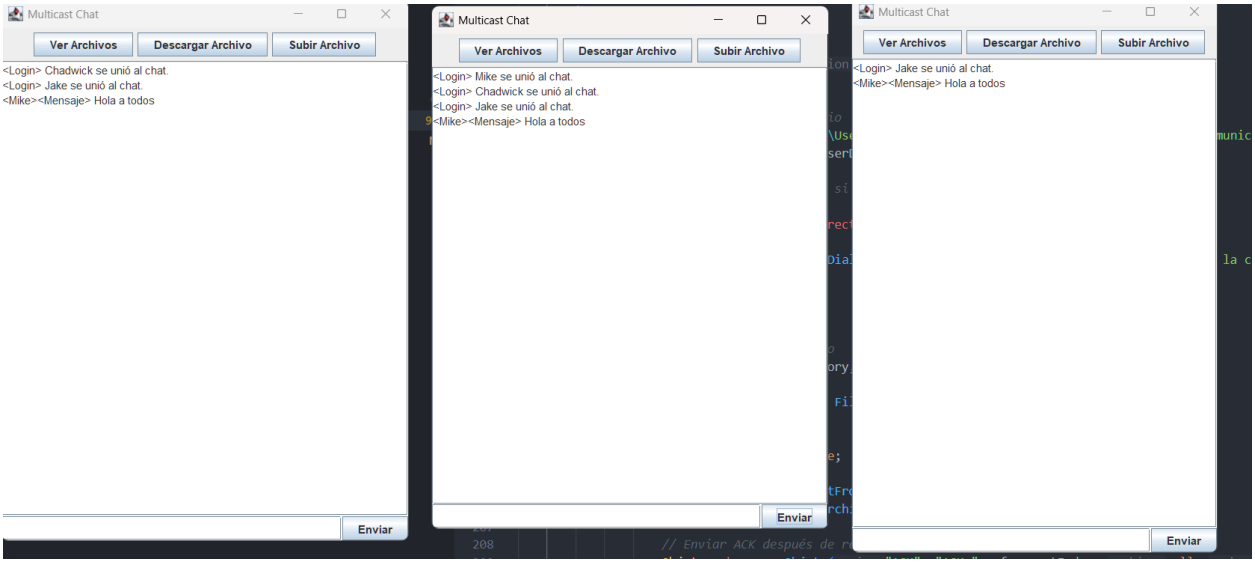
- **Envío de confirmaciones (ACK)** para garantizar que los fragmentos lleguen correctamente.

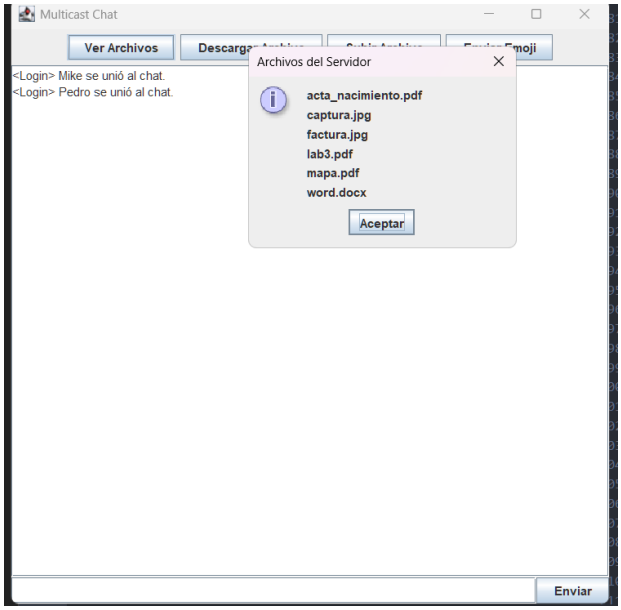
5.6 Implementacion de la clase

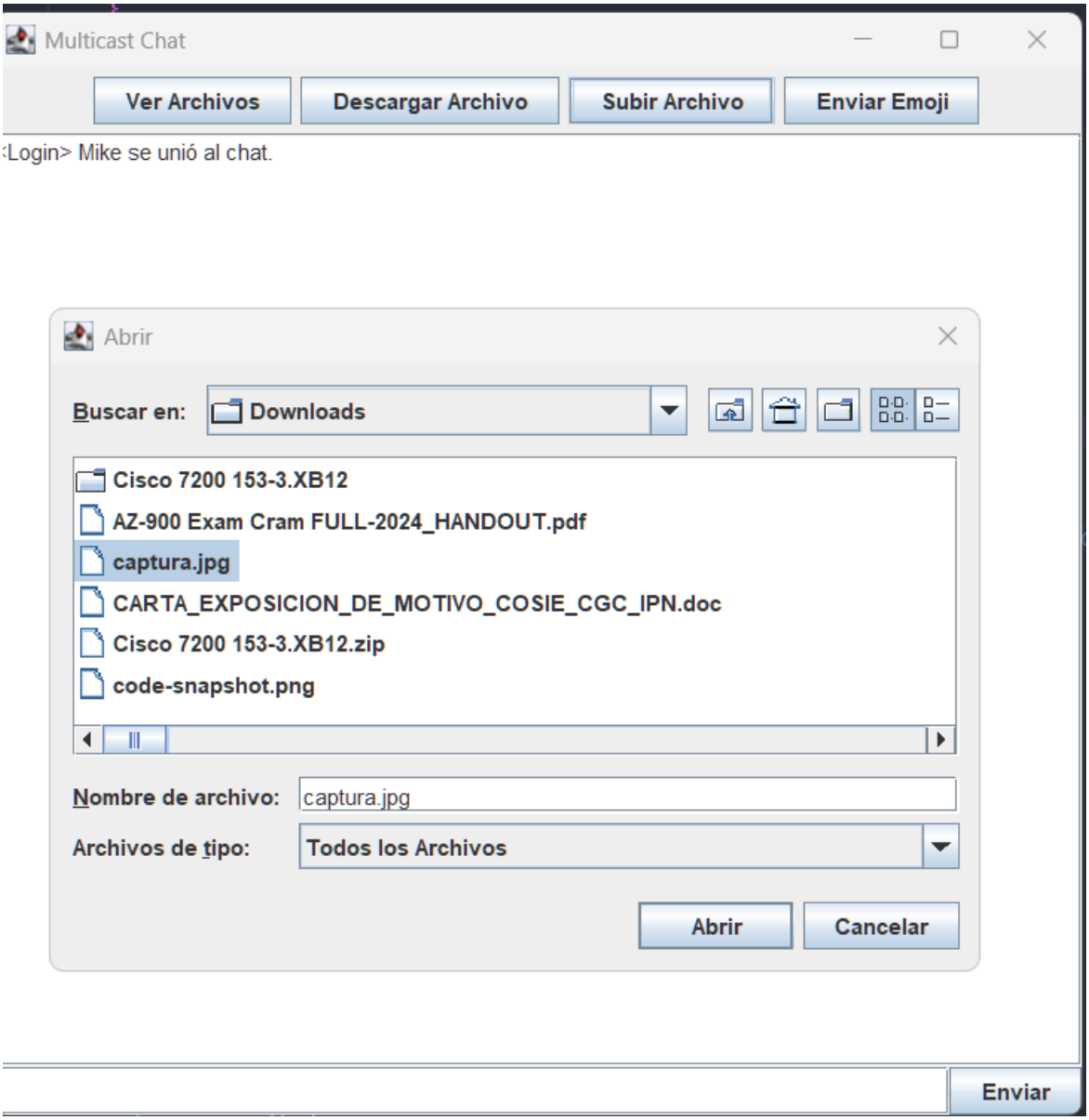
La clase Objeto es una clase diseñada para facilitar la comunicación entre el cliente y el servidor en tu aplicación de chat multicast. Esta clase implementa la interfaz Serializable, lo que permite que sus instancias puedan ser enviadas a través de la red utilizando flujos de datos.

- La interfaz Serializable permite que la clase pueda ser convertida en una secuencia de bytes, facilitando su envío por medio de sockets en una red.
- No requiere que implementes métodos adicionales; simplemente marca la clase como serializable.
- Todas las propiedades de la clase también deben ser serializables para que funcione correctamente (en este caso, String, byte[], int y boolean son tipos primitivos o serializables por naturaleza).

6 Ejecución del programa







7 Conclusiones

La implementación de este sistema de chat multicast con transferencia de archivos permitió evidenciar la efectividad del protocolo multicast para el desarrollo de aplicaciones distribuidas. Entre las principales conclusiones obtenidas destacan las siguientes:

- **Eficiencia en la comunicación:** El uso de multicast demostró ser una solución eficaz para enviar mensajes simultáneamente a múltiples clientes, reduciendo significativamente el consumo de ancho de banda y la carga del servidor en comparación con conexiones unicast tradicionales.

- **Gestión de concurrencia:** La implementación del chat con múltiples clientes evidenció la necesidad de manejar de forma adecuada los procesos concurrentes para garantizar que los mensajes se recibieran correctamente sin bloqueos ni pérdidas de datos.
- **Transferencia segura de archivos:** La incorporación de un mecanismo de fragmentación de archivos con confirmación de recepción (`ACK`) aseguró la integridad de los datos durante el proceso de envío y descarga. Esto fue especialmente importante para archivos grandes, donde se logró minimizar el riesgo de pérdida de información.
- **Escalabilidad del sistema:** El diseño basado en multicast facilita la expansión del sistema, permitiendo que nuevos clientes se unan sin necesidad de reconfigurar el servidor o establecer múltiples conexiones directas.

En conclusión, esta práctica permitió comprender y aplicar conceptos fundamentales de la programación en red, destacando la importancia del manejo de sockets, la sincronización de procesos concurrentes y el uso eficiente de los recursos en sistemas distribuidos.