Instituto Politécnico Nacional

Escuela Superior de Cómputo

Aplicaciones para Comunicaciones en Red

Práctica 3:
Multicast P2P

Alumnos:

Malagón Baeza Alan Adrian

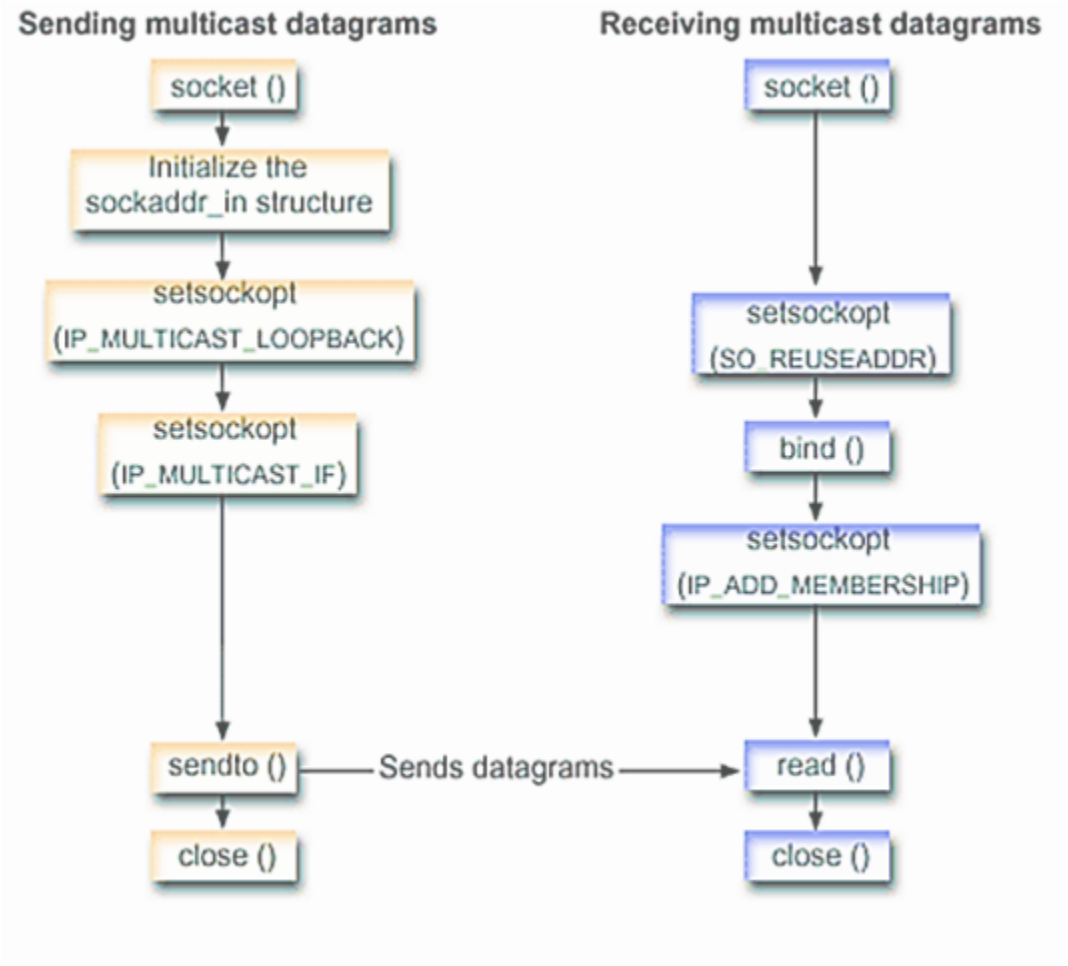
Martínez Chávez Jorge Alexis

Profesor:
Moreno Cervantes Axel Ernesto

Grupo: 6CM1

1. Introducción

La multidifusión es un proceso de comunicación que tiene lugar en un entorno de red. Básicamente, una multidifusión es un mensaje que se origina con un solo usuario y es recibido por múltiples puntos finales en la red. En cierto sentido, una multidifusión es algo así como enviar un solo correo electrónico a varias direcciones de correo electrónico. Sin embargo, la diferencia clave es que una multidifusión no depende de ningún tipo de dirección de correo electrónico o software, y la transmisión se limita a los usuarios que están conectados a la red única.



Recepción de datagramas de multidifusión.

Unirse a un grupo de multidifusión.

La transmisión es más fácil de implementar que la multidifusión. No requiere procesos para darle al kernel algunas reglas sobre qué hacer con los paquetes de transmisión.

Sin embargo, con la multidifusión, es necesario informar al núcleo en qué grupos de multidifusión se está interesado. Es decir, se tiene que pedirle al núcleo que se "úna" a esos grupos de multidifusión. Según el hardware subyacente, los datagramas de multidifusión se filtran por el hardware o por la capa IP. Solo se aceptan aquellos con un grupo de destino previamente registrado a través de una unión.

Se debe tener en consideración que es posible unirse al mismo grupo en más de una interfaz de red en particular. Si no se especifica una interfaz concreta, el kernel la elegirá en función de sus tablas de enrutamiento cuando se envíen los datagramas. También es posible que más de un proceso se una al mismo grupo de multidifusión en la misma interfaz. Todos recibirán los datagramas enviados a ese grupo a través de esa interfaz.

Finalmente, se considera que para que un proceso reciba datagramas de multidifusión, tiene que pedirle al núcleo que lo una al grupo y vincule el puerto al que se envían estos datagramas. La capa UDP utiliza tanto la dirección como el puerto de destino para demultiplexar los paquetes y decidir a qué socket los envía.

Abandonar a un grupo de multidifusión.

Cuando un proceso ya no está interesado en un grupo de multidifusión, le informa al kernel que desea abandonar ese grupo. Es importante comprender que esto no significa que el kernel ya no aceptará datagramas de multidifusión destinados a ese grupo de multidifusión. Todavía lo hará si hay más procesos que emitieron una petición de "unirse a multidifusión" para ese grupo y todavía están interesados. En ese caso , el anfitrión sigue siendo miembro del grupo, hasta que todos los procesos decidan abandonar el grupo.

Si abandona el grupo, pero permanece vinculado al puerto en el que estaba recibiendo el tráfico de multidifusión, y hay más procesos que se unieron al grupo, seguirá recibiendo las transmisiones de multidifusión.

La idea es que unirse a un grupo de multidifusión solo le dice a la capa de enlace de datos e IP que acepte datagramas de multidifusión destinados a ese grupo. No es una membresía por proceso, sino una membresía por host.

Direccionamiento de multidifusión.

Existe la posibilidad de asignar un direccionamiento estático en el que, por ejemplo, se puede configurar una conexión a un servidor multicast para que este ofrezca el servicio correspondiente. Por otra parte, las direcciones multicast pueden también tener una asignación dinámica, ya que los grupos de multidifusión subyacentes no tienen porqué existir de forma permanente. Esto significa que es muy fácil crear grupos privados y también eliminarlos. Independientemente de si la asignación de direcciones ha sido realizada de forma estática o dinámica, el rango de direcciones de las redes IP es de 224.0.0.0 a 239.255.255.255 (o bien FF00::/8), también conocido como espacio de direcciones clase D, que está reservado a tal fin.

1.2 Concurrencia en Java

La concurrencia es la capacidad de hacer más de una cosa al mismo tiempo. A menudo a los desarrolladores de software se nos presenta el problema de, más allá de conseguir que una aplicación funcione correctamente, que lo haga de manera más rápida para satisfacer los requisitos del cliente.

Para esto, conviene diferenciar los conceptos de concurrencia y paralelismo. Concurrencia se da cuando dos o más tareas se desarrollan en el mismo intervalo de tiempo, pero que no necesariamente están progresando en el mismo instante. Es un concepto más general que el paralelismo, el cual consiste en llevar a cabo multitareas en el mismo instante literalmente.

Existen dos conceptos básicos asociados a la concurrencia:

- Proceso: es un programa en ejecución. Tiene su propio espacio de memoria, enlaces a recursos, I/O. Los procesos están aislados entre sí.
- Hilo: es un camino de ejecución dentro de un proceso. Cada proceso tiene al menos un hilo, llamado hilo principal. Los hilos comparten los recursos del proceso, incluida la memoria, por lo que pueden comunicarse entre sí. Cada hilo tiene su propia callstack.

Una vez que hemos decidido optimizar nuestro programa mediante concurrencia, y aunque no es estrictamente obligatorio conocer el nivel de paralelismo máximo

que ofrece la máquina para la cual desarrollamos, es muy importante para hacer previsiones de optimización o para contrastar los resultados en pruebas.

Cada placa base de una computadora dispone de uno o varios sockets donde insertar procesadores. El número de “cores” o procesadores físicos puede ser uno o varios. Si nuestro equipo tiene un socket y n cores, significa que se podrían ejecutar literalmente n procesos realmente en paralelo.
Ahora bien, en Java el código que ejecuta un thread se define en clases que implementan la interfaz Runnable. Para la creación de hilos se cuenta con la clase Thread, la cual permite crear un hilo de ejecución en un programa. La clase Thread cuenta con los siguientes métodos:

- run(): actividad del thread
- start(): activo run() y vuelve al llamante
- join(): espera por la terminación (timeout opcional)
- interrupt(): sale de un wait, sleep o join
- isInterrupted()
- yield()
- stop(), suspend(), resume() (deprecated)

Métodos estáticos

- sleep(milisegundos)
- currentTread()

Métodos de la clase Object que controlan la suspensión de threads

- wait(), wait(milisegundos), notify(), notifyAll()

2. Desarrollo

Esta práctica se desarrolló en Java, ya que dispone de toda una API para trabajar con sockets, hilos y todo lo necesario para desarrollar aplicaciones cliente-servidor.

Para trabajar con sockets en Java disponemos de la clase Socket y ServerSocket para realizar conexiones desde un cliente o para establecer la conexión de un servidor, respectivamente. También se usó JavaFX para realizar la interfaz gráfica.

2.1 Servidor

El proyecto Java “Practica3RedesServer” cuenta con 7 clases:

1. Constants
2. DownloadServer
3. FoundFile
4. MD5Checksum
5. MulticastServer
6. SearchServer
7. StartServers

Se procederá a dar una breve explicación de estas clases y las partes importantes del código para el funcionamiento de la aplicación.

2.1.1 Constants

Contiene las constantes utilizadas en el servidor como mensajes, direcciones, puertos y rutas.

2.1.2 DownloadServer

Este servidor únicamente se encargará de recibir peticiones de descarga y enviar los archivos, o fragmentos de archivos solicitados al cliente. El método sendFile()

El envío o descarga del archivo se almacena en el directorio de la aplicación cliente. El proceso de descarga es el siguiente, se hace una instancia de la clase DataOutputStream con ella se establece el flujo de salida para escribir los datos del archivo, los cuales se almacenan en un arreglo binario de tamaño 1500 bytes. Al usuario se le muestra el porcentaje y la cantidad de bytes leídos del archivo mediante una barra de progreso, cuando la cantidad de bytes recibidos es igual al tamaño se termina de escribir pues el archivo se descargó completamente. Finalmente se cierran los flujos de entrada, salida y el cliente. Lo cual se puede notar en la siguiente imagen.

```
package com.ipn.practica3redes;

import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;

/**Stream de server que se usará para enviar los archivos solicitados*/
public class DownloadServer extends Thread{
   private ServerSocket s;
   private Socket cl;

   public void run(){
       startServer();
       sendFile();
   }
```

```
   public void startServer(){
       try {
           s = new ServerSocket(Constants.DOWNLOAD_PORT);
           s.setReuseAddress(true);
       }catch (Exception e){
           e.printStackTrace();
       }
   }

   public void sendFile(){
       while(true){
           try {
               cl = s.accept();

System.out.println(String.format(Constants.CLIENT_CONNECTION,cl.getInetAdd
ress(), cl.getPort()));

               /**Leyendo entrada*/
                                                DataInputStream   dis   =   new
DataInputStream(cl.getInputStream());
               String name = (String) dis.readUTF();
               File f = new File(name);
               String fileName = f.getName();
               long tam = f.length();
               String path = f.getAbsolutePath();
               System.out.println(Constants.SENDING_FILE);

System.out.println(String.format(Constants.FILE_SIZE,fileName, tam));
                                DataOutputStream  dos  =  new  DataOutputStream
(cl.getOutputStream());

                           DataInputStream  disFile  =  new  DataInputStream(new
FileInputStream(path));

               /**Enviando archivo*/
               dos.writeUTF(fileName);
               dos.flush();
               dos.writeLong(tam);
               byte[] b = new byte[Constants.FILE_BUFFER_SIZE];
               long sent = 0;
               int n;
               while(sent  <tam){
                   n = disFile.read(b);
                   dos.write(b, 0, n);
                   dos.flush();
                   sent += n;
               }
               disFile.close();
               dos.close();
               cl.close();
               System.out.println(Constants.FILE_SENT_SUCCESSFULLY);
               dis.close();
               cl.close();
           }catch (IOException e){
               e.printStackTrace();
           }
```

```
        }
    }
}
```

2.1.3 FoundFile

Contiene los métodos getFileName(), setFileName(), getPath, setPath(), getMd5() y setMd5() para establecer el nombre y dirección así como el MD5 perteneciente a un archivo

2.1.4 MD5Checksum

Permite obtener el MD5 de un archivo, a continuación se explica el procedimiento. Primero se crea un flujo de entrada con el archivo seleccionado, de este se leen cierta cantidad de bytes y se guardan en un arreglo de tipo byte, se crea un MessageDigest que será el encargado de crear y actualizar el digesto correspondiente de acuerdo con la cantidad de bytes leídos. Luego, cuando se termina la lectura del archivo se cierra el flujo de entrada y se regresa el digesto correspondiente. Después, se convierte el digesto obtenido en formato hexadecimal a cadena de texto y se obtiene el resultado. Para ello se utilizan los métodos createChecksum() y getMD5Checksum()

2.1.5 MulticastServer

Este servidor básicamente se encargará de crear un hilo, el cual anunciará cada 5 segundos el puerto de servicio del servidor de flujo. En la siguiente imagen se observa el código implementado para esta clase, se puede notar que también se dispone del método send() de tipo booleano que permite enviar el anuncio. Si el resultado obtenido del método send() es true el hilo permanecerá dormido 5 segundos, en caso contrario no lo hará.

```
package com.ipn.practica3redes;

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.InetAddress;
import java.net.MulticastSocket;

public class MulticastServer extends Thread{

   // Dirección multicast
   private InetAddress group = null;

   public void run(){
```

```
        try {
            group = InetAddress.getByName(Constants.MULTICAST_ADDRESS);
            while (true){
                send(   "--------------ANUNCIO--------------" + "\n" +
                                                                "MULTICAST  ADDRESS:
"+Constants.MULTICAST_ADDRESS+"\n" +
                        "MULTICAST PORT: "+Constants.MULTICAST_PORT+"\n" +
                        "DOWNLOAD PORT: "+Constants.DOWNLOAD_PORT+"\n" +
                        "SEARCH PORT: "+Constants.SEARCH_PORT+"\n"+
                        "-----------------------------------");
                try {
                    Thread.sleep(5000);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
        }catch (IOException e){
            System.exit(2);
        }
    }

    public Boolean send(String message){
        try {
            // Creamos el socket multicast
                                    MulticastSocket   sendSocket   =   new
MulticastSocket(Constants.MULTICAST_PORT);
            // Unimos el socket al grupo
            sendSocket.joinGroup(group);
            byte[] buffer = message.getBytes();
                        DatagramPacket  packet  =  new  DatagramPacket(buffer,
buffer.length, group, Constants.MULTICAST_PORT);
            // Enviamos el mensaje multicast
            sendSocket.send(packet);
            // Esperamos 5 segundos antes de enviar otro anuncio
            sendSocket.close();
            return true;
        }catch (IOException e){
            e.printStackTrace();
            return false;
        }
    }

    public static void main(String[] args) {
        try {
            MulticastServer multicastServer = new MulticastServer();
            multicastServer.start();
        }catch (Exception e){
            e.printStackTrace();
        }
    }
}
```

2.1.6 SearchServer

Se implementa el servidor de búsqueda de archivos, el cual recibe como parámetro de entrada el nombre y dirección del archivo a ser buscado, y devuelve como salida el nombre y path del archivo encontrado, así como el MD5 del archivo encontrado. Si no se encuentra el archivo se retorna la cadena de texto “unknow”. El método encargado de realizar esta acción es searchFile() el cual devuelve un objeto de tipo FoundFile.

```
package com.ipn.practica3redes;

import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;

public class SearchServer extends Thread{

   File directory = new File(Constants.FILES_PATH);
   private ServerSocket s;
   private Socket cl;

   public void run(){
       startServer();
       searchFile();
   }

   public void startServer() {
       try{
           s = new ServerSocket(Constants.SEARCH_PORT);
           s.setReuseAddress(true);
       }catch (Exception e){
           e.printStackTrace();
       }
   }

   public void searchFile() {
       while (true){
           try {
               cl=s.accept();

System.out.println(String.format(Constants.CLIENT_CONNECTION,cl.getInetAdd
ress(),cl.getPort()));

               // Leemos la entrada
                                          DataInputStream   dis   =   new
DataInputStream(cl.getInputStream());
               String name = dis.readUTF();
               File f = new File(name);
               String fileName = f.getName();

               int servers = dis.readInt();

               //Enviamos la busqueda
               for (int i = 0; i < servers; i++) {
                   FoundFile result = new FoundFile();
                   result.setFileName("unknown");
```

```
                    File dir = directory;
                    File[] files = dir.listFiles();
                    for(File file : files)
                        if(file.isFile())
                            if(file.getName().equals(fileName)){
                                result.setFileName(file.getName());
                                result.setPath(file.getAbsolutePath());

result.setMd5(getMD5(file.getAbsolutePath()));
                            }

                                                    ObjectOutputStream oos = new
ObjectOutputStream(cl.getOutputStream());
                    oos.writeObject(result);
                    oos.flush();
                    oos.close();
                }

                dis.close();
                cl.close();
            } catch (IOException e){
                e.printStackTrace();
            }
        }
    }

    public String getMD5(String path){
        try{
            MD5Checksum md5 = new MD5Checksum();
            return md5.getMD5Checksum(path);
        }catch (Exception e){
            e.printStackTrace();
        }
        return null;
    }
}
```

2.1.7 StartServers

Permite iniciar los servidores: DownloadServer, SearchServer y MulticastServer.

2.2 Cliente

El proyecto Java “Practica3RedesServer” cuenta con 11 clases:

1. ClientApplication
2. ClientController
3. Constants
4. DataFromServer

5. DownloadClient
6. Files
7. FileServer
8. FoundFile
9. MulticastClient
10. MulticastServersWatcher
11. SearchClient

Se procederá a dar una breve explicación de estas clases y las partes importantes del código para el funcionamiento de la aplicación.

2.2.1 ClientApplication

Este código es una aplicación cliente en JavaFX. La clase `ClientApplication` extiende de la clase `Application` y se encarga de cargar la interfaz gráfica de usuario (GUI) de la aplicación desde un archivo FXML llamado `app.fxml`. En el método `start`, se crea un objeto `FXMLLoader` y se le pasa como parámetro la ruta del archivo FXML. Luego, se carga la escena con el método `load` del `FXMLLoader` y se establece como escena principal en la ventana `Stage`. Por último, se establece el título de la ventana con la constante `Constants.WINDOW_TITLE` y se muestra la ventana con el método `show`. En el método `main`, se inicia la aplicación JavaFX mediante el método `launch()`.

```
public class ClientApplication extends Application {
    @Override
    public void start(Stage stage) throws IOException {
                                        FXMLLoader     fxmlLoader     =     new
FXMLLoader(ClientApplication.class.getResource("app.fxml"));
        Scene scene = new Scene(fxmlLoader.load());
        stage.setTitle(Constants.WINDOW_TITLE);
        stage.setScene(scene);
        stage.show();
    }

    public static void main(String[] args) {
        launch();
    }
}
```

2.2.2 ClientController

En esta parte tenemos la interfaz gráfica del usuario, básicamente consta de un botón para buscar el archivo solicitado y una vez encontrado nos lo muestra en la lista, si damos clic al elemento el la lista muestra su información y se habilita un

botón para descargar el archivo y una barra de progreso de la descarga se hace visible. A continuación se muestra el código de esta clase.

```
public class ClientController implements Initializable {

    Files db = new Files();

    DownloadClient dClient = new DownloadClient(this);
    MulticastClient mClient = new MulticastClient(db);
    MulticastServersWatcher msw = new MulticastServersWatcher(db);
    SearchClient sClient = new SearchClient(db,this);

    @FXML
    private Button downloadBtn;

    @FXML
    private ProgressBar downloadPB;

    @FXML
    private ListView<String> filesLV;

    @FXML
    private Label md5Lbl;

    @FXML
    private Label nameLbl;

    @FXML
    private TextField nameTxt;

    @FXML
    private Label ruteLbl;

    @FXML
    private Label serverLbl;

    @FXML
    void onDownloadButtonClick(ActionEvent event) {
        if(selectedLV != -1){
            downloadPB.setVisible(true);
            FileServer fs = db.getFiles().get(selectedLV);
            dClient.serverConnection(fs.getFileInServer());
            dClient.receiveFile(fs.getFoundFile());
        }
    }

    @FXML
    void onSearchButtonClick(ActionEvent event) {
        if(nameTxt.getText().isEmpty()){
            Alert alert = new Alert(Alert.AlertType.WARNING);
            alert.setHeaderText(null);
            alert.setContentText(Constants.EMPTY_FILE_NAME);
            alert.show();
            return;
        }
        List<DataFromServer> servers = db.getServers();
```

```
        db.clearFiles();
        if(!servers.isEmpty()){
            sClient.serverConnection();
            sClient.receiveFile(nameTxt.getText());
        } else{
            Alert alert = new Alert(Alert.AlertType.WARNING);
            alert.setHeaderText(null);
            alert.setContentText(Constants.EMPTY_SERVERS_MESSAGE);
            alert.show();
        }
    }

    public void search(List<FileServer> files){
        filesLV.getItems().clear();
        List<String> filesNames = new ArrayList<>();
        for (FileServer fs: files
             ) {
            filesNames.add(fs.getFoundFile().getFileName());
        }
        filesLV.getItems().addAll(filesNames);

        if(filesNames.isEmpty()){
            Alert alert = new Alert(Alert.AlertType.INFORMATION);
            alert.setHeaderText(null);
            alert.setContentText(Constants.FILE_NOT_FOUND);
            alert.show();
        }else{
            Alert alert = new Alert(Alert.AlertType.INFORMATION);
            alert.setHeaderText(null);
            alert.setContentText(Constants.FOUND_FILE);
            alert.show();
        }
    }

    private int selectedLV = -1;
    @Override
    public void initialize(URL url, ResourceBundle resourceBundle) {
        clientsStart();

        filesLV.setOnMouseClicked(mouseEvent -> {
            selectedLV = filesLV.getSelectionModel().getSelectedIndex();
            if(selectedLV < 0) return;
            FileServer fs = db.getFiles().get(selectedLV);
            nameLbl.setText(fs.getFoundFile().getFileName());
            serverLbl.setText(fs.getFileInServer());
            ruteLbl.setText(fs.getFoundFile().getPath());
            md5Lbl.setText(fs.getFoundFile().getMd5());
            downloadBtn.setVisible(true);
            downloadPB.setVisible(true);
        });
    }
```

En el método clientsStart empezamos la ejecución de los clientes.

```
    public void clientsStart(){
        mClient.start();
```

```
        msw.start();
        sClient.start();
        dClient.start();
        downloadBtn.setVisible(false);
        downloadPB.setVisible(false);
    }

    public void setProgressBar(int percentage) {
        downloadPB.setProgress(percentage);
    }
}
```

2.2.3 Constants

Contiene las constantes utilizadas en el cliente como mensajes, direcciones, puertos y rutas.

2.2.4 DataFromServer

Esta clase se encarga de establecer la información que contendrá cada servidor, en este caso es la dirección, su temporizador y el puerto.

```
public class DataFromServer implements Serializable {
    private String address;
    private int temp;
    private int port;

    public DataFromServer(String address, int temp, int port) {
        this.address = address;
        this.temp = temp;
        this.port = port;
    }

    public DataFromServer(){}

    public String getAddress() {
        return address;
    }
    public int getTemp() {
        return temp;
    }

    public void setTemp(int temp) {
        this.temp = temp;
    }

    public int getPort() {
        return port;
```

```
    }

    public void setPort(int port) {
        this.port = port;
    }
}
```

2.2.5 DownloadClient

En este cliente lo que hacemos es básicamente un cliente de flujo que se encargará de conectarse al servidor de flujo y recibirá los archivos que el servidor le mandó, en otras palabras se encargará de descargarlos.

```
public class DownloadClient extends Thread{
    private ClientController app;
    private Socket cl;

    public DownloadClient(ClientController app){
        this.app = app;
    }

    /**
     * Metodo que conecta al servidor local
     */
    public void serverConnection(String address){
        try {
            cl = new Socket(address, Constants.DOWNLOAD_PORT);
            System.out.println(Constants.CLIENT_CONNECTED);
        }catch (Exception e){
            e.printStackTrace();
        }
    }

    /**
     * Metodo que descarga el archivo especificado
     */
    public void receiveFile(FoundFile file){
        if(cl != null){
            try {
                DataOutputStream dos = new
DataOutputStream(cl.getOutputStream());
                //*Mandando el archivo que queremos descargar
                dos.writeUTF(file.getPath());
                dos.flush();
                /**Esperando una respuesta*/
                DataInputStream dis = new
DataInputStream(cl.getInputStream());
                /**Recibiendo el archivo*/
```

```
                String name = dis.readUTF();
                long tam = dis.readLong();
                DataOutputStream dosFile = new DataOutputStream(new
FileOutputStream(String.format(Constants.FILES_PATH,name)));
                byte []b = new byte[Constants.FILE_BUFFER_SIZE];
                long received = 0;
                int percentage;
                int n;
                while(received < tam){
                    n = dis.read(b);
                    dosFile.write(b, 0, n);
                    dosFile.flush();
                    received += n;
                    percentage = (int)((received*100) / tam);
                    app.setProgressBar(percentage);
                }
                dos.close();
                dosFile.close();
                dis.close();
                cl.close();

                Alert alert = new Alert(Alert.AlertType.INFORMATION);
                alert.setHeaderText(null);
                alert.setContentText(Constants.DOWNLOADED_FILE);
                alert.showAndWait();

                app.setProgressBar(0);

            }catch (IOException e){
                e.printStackTrace();
            }
        }
    }

    public void run(){}
}
```

2.2.6 Files

Esta clase nos va a servir como una base de datos dentro de nuestro programa, pues contendrá el archivo que está en el servidor, el archivo que ha sido encontrado en el servidor y la lista de servidores que tenemos disponibles, además de contar con un método que nos sirve para agregar servidores.

```
public class Files {

    private List<DataFromServer> servers = new ArrayList<>();

    private List<FileServer> files = new ArrayList<>();

    public List<DataFromServer> getServers() {
        return servers;
    }
```

```
    public List<FileServer> getFiles(){return files;}
    public void setServers(List<DataFromServer> servers) {
        this.servers = servers;
    }
    public void addServer(DataFromServer dfs){
        servers.add(dfs);
    }
    public void setFiles(List<FileServer> files){this.files = files;}
    public void addFile(FileServer fs){files.add(fs);}
    public void clearFiles(){files.clear();}
```

2.2.7 FileServer

La clase `FileServer` es una clase simple que se utiliza para guardar la información de un archivo y el servidor de origen. Tiene dos propiedades privadas: `fileInServer`, que es un `String` que representa el nombre del archivo en el servidor, y `foundFile`, que es una instancia de la clase `FoundFile`, que contiene información sobre la ubicación del archivo y otros detalles. La clase tiene dos constructores: uno que acepta dos parámetros, `fileInServer` y `foundFile`, para inicializar las propiedades, y otro sin parámetros. La clase también tiene cuatro métodos públicos `get` y `set` para acceder y modificar las propiedades `fileInServer` y `foundFile`. Los métodos `get` devuelven el valor actual de las propiedades y los métodos `set` establecen un nuevo valor para las propiedades. En resumen, la clase `FileServer` es una clase simple que se utiliza para almacenar información relacionada con un archivo y su ubicación en un servidor.

```
public class FileServer {
    private String fileInServer = new String();
    private FoundFile foundFile = new FoundFile();

    public FileServer(String fileInServer, FoundFile foundFile) {
        this.fileInServer = fileInServer;
        this.foundFile = foundFile;
    }

    public FileServer() {
    }

    public String getFileInServer() {
        return fileInServer;
    }

    public void setFileInServer(String fileInServer) {
        this.fileInServer = fileInServer;
    }

    public FoundFile getFoundFile() {
        return foundFile;
    }
```

```
    public void setFoundFile(FoundFile foundFile) {
        this.foundFile = foundFile;
    }
}
```

2.2.8 FoundFile

Contiene los métodos getFileName(), setFileName(), getPath, setPath(), getMd5() y setMd5() para establecer el nombre y dirección así como el MD5 perteneciente a un archivo.

2.2.9 MulticastClient

Primeramente hacemos la conexión al servidor multicast obtenemos el grupo multicast, esto lo hacemos directamente en el constructor como se ve a continuación.

```
public class MulticastClient extends Thread{
    private List<DataFromServer> servers = new ArrayList<>();
    private InetAddress group = null;
    private final Files db;

    public MulticastClient(Files db){
        this.db = db;
        try {
            group = InetAddress.getByName(Constants.MULTICAST_ADDRESS);
        }catch (UnknownHostException e){
            e.printStackTrace();
            System.exit(1);
        }
    }
```

En el método run nos unimos al grupo multicast y nos mantenemos a la escucha de todos los datagramas que lleguen, a partir de estos vamos obteniendo la información que será IP del anunciante y los datos donde estará el puerto del servidor de flujo del servidor anunciante. Cada vez que se reinicia el temporizador vamos agregando ese servidor a nuestra lista de servidores.

```
    public void run(){
        try {
```

```
            MulticastSocket socket = new
MulticastSocket(Constants.MULTICAST_PORT);
            socket.joinGroup(group);
            String msg = "";
            for(;;){
                byte []buffer = new byte[Constants.BUFFER_SIZE];
                DatagramPacket recv = new DatagramPacket(buffer,
buffer.length);
                socket.receive(recv);
                byte []data = recv.getData();
                msg = new String(data);
                msg = msg.trim();

                System.out.println(msg);

                /**Verifica si el servida ya esta guardado*/
                DataFromServer curServer = new
DataFromServer(recv.getAddress().toString().substring(1), 6,
recv.getPort());
                /**Agrega server*/
                servers = db.getServers();
                int pos = isInList(servers, curServer);
                if(pos == -1){
                    db.addServer(curServer);

                }else{
                    servers.get(pos).setTemp(6);
                    db.setServers(servers);
                }
            }
        }catch (IOException e){
            e.printStackTrace();
            System.exit(2);
        }
    }

    public int isInList(List<DataFromServer>list, DataFromServer dfs){
        int i;
        for(i = 0; i < list.size(); i++){
            if(list.get(i).getAddress().equals(dfs.getAddress())){
                return i;
            }
        }
        return -1;
    }

}
```

2.2.10 MulticastServersWatcher

En esta clase vamos a estar actualizando la lista de servidores cada segundo, si un servidor no se reporta, esto es cuando el temporizador llega a 0, se elimina de

la lista de servidores, en caso de un datagrama sea anunciado la dirección de una entrada ya existente en la lista, el contador será reiniciado.

```
public class MulticastServersWatcher extends Thread{
   private final Files db;

   public MulticastServersWatcher(Files db){
       this.db = db;
   }

   /**
    * Comprobamos el temporizador inicializado en 6 segundos esperando un
datagrama
    * si no se captura un datagrama borramos el servidor de la lista
    */
   public void run(){
       int i = 0;
       for(;;){
           try {
               List<DataFromServer> servers = db.getServers();
               if(servers.size() != 0){

                   for (i = 0; i < servers.size(); i++){
                       /**Eliminar servidor si no se reporta*/
                       if(servers.get(i).getTemp() == 0)
                           servers.remove(i);
                       else

servers.get(i).setTemp(servers.get(i).getTemp()-1);
                   }
               }
               Thread.sleep(1000);
           }catch (InterruptedException e){
               Thread.currentThread().interrupt();
           }
       }
   }
}
```

2.2.11 SearchClient

En este cliente lo que hacemos es básicamente un cliente de flujo que se encargará de conectarse al servidor de flujo y mandará el nombre del archivo a buscar para recibir los archivos que el servidor encuentre, en otras palabras se encargará de devolver la búsqueda con los datos de los archivos y de que servidor provienen.

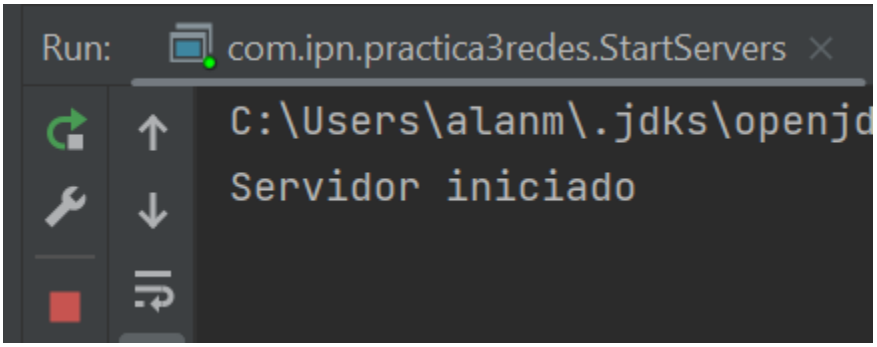
```
public class SearchClient extends Thread{

   private ClientController app;
   private Socket cl;
   private final Files db;
```

```
    public SearchClient(Files db, ClientController helloController) {
        this.app = helloController;
        this.db = db;
    }

    public void serverConnection(){
        try {
                        cl  =  new  Socket(db.getServers().get(0).getAddress(),
Constants.SEARCH_PORT);
            System.out.println(Constants.CLIENT_CONNECTED);
        }catch (Exception e){
            e.printStackTrace();
        }
    }

    /**
     * Método que busca el archivo especificado
     * @param fileName
     */
    public void receiveFile(String fileName){
        if(cl != null){
            try {
                // Mandamos el archivo a buscar
                                                DataOutputStream   dos   =   new
DataOutputStream(cl.getOutputStream());
                dos.writeUTF(fileName);
                dos.flush();

                int i;
                try {
                    List<DataFromServer> servers = db.getServers();
                    dos.writeInt(servers.size());
                    dos.flush();

                    List<FileServer> files = new ArrayList<>();
                    if(servers.size() != 0){
                        for(i = 0; i < servers.size(); i++){
                            // Esperamos por la respuesta
                                                    ObjectInputStream  ois  =  new
ObjectInputStream(cl.getInputStream());
                            // Recibimos la busqueda
                                                FoundFile  response  =  (FoundFile)
ois.readObject();
                            if(!response.getFileName().equals("unknown")){
                                                        FileServer  fs  =  new
FileServer(servers.get(i).getAddress(),response);
                                files.add(fs);
                                db.addFile(fs);
                            }
                            ois.close();
                        }
                    }
                    app.search(files);
                }catch (Exception e){
                    e.printStackTrace();
                }
                dos.close();
```

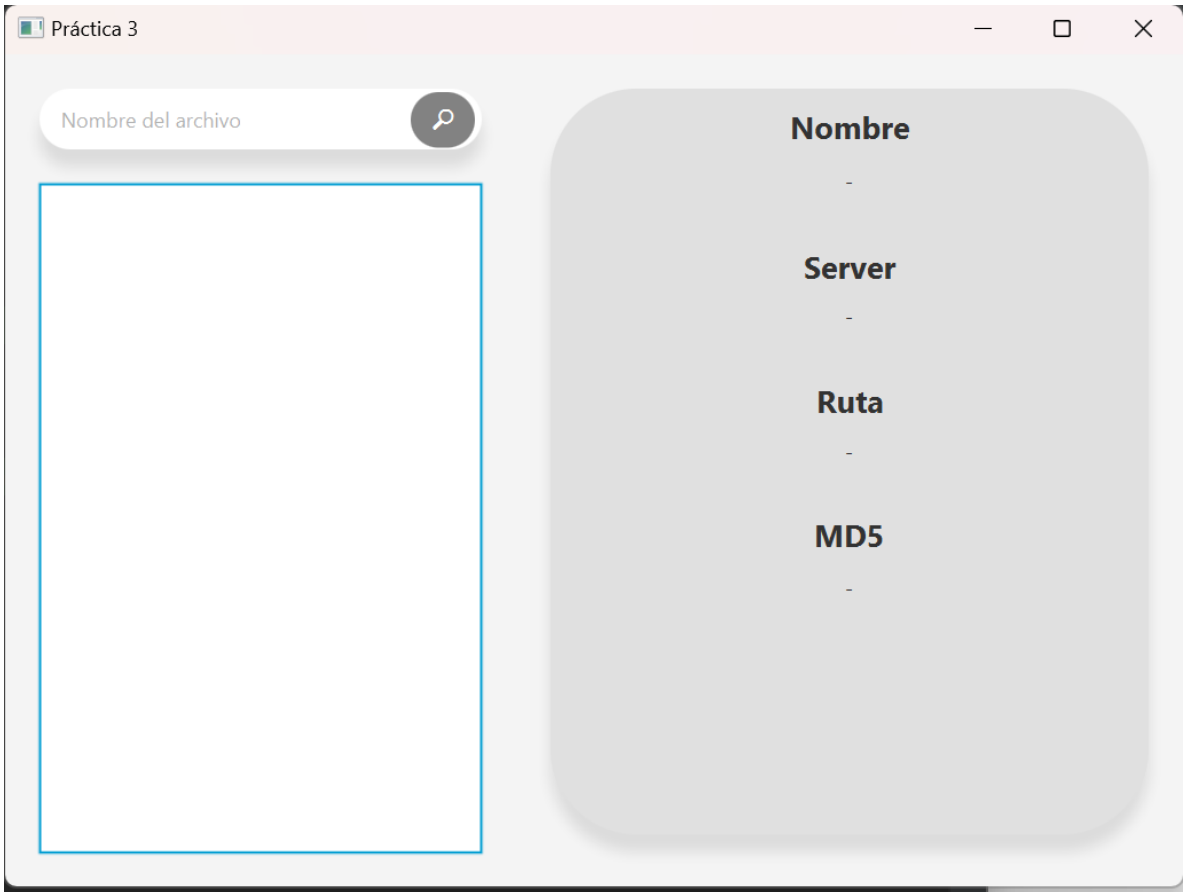
```
                cl.close();

            }catch (IOException e){
                e.printStackTrace();
            }
        }
    }

    public void run(){}
}
```

3. Pruebas

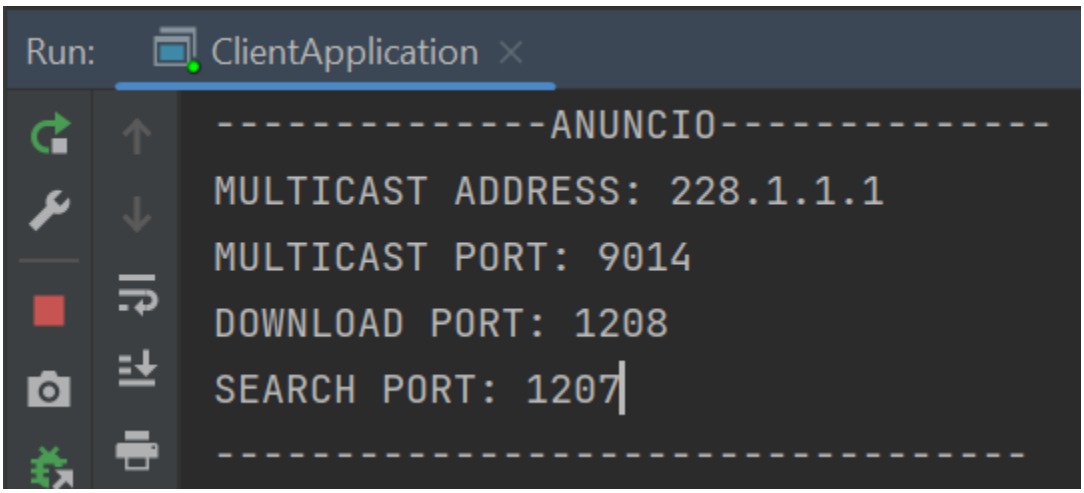
Para comenzar con el programa tenemos que correr la clase startServers de parte del servidor, este nos mostrará el siguiente mensaje.



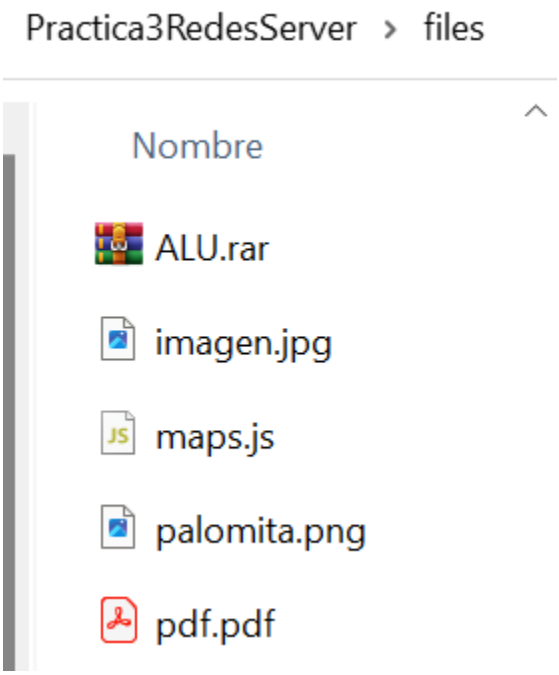
Una vez corrido nuestro servidor podemos correr el cliente con la clase ClientApplication del cliente, al correrla tenemos la siguiente interfaz.



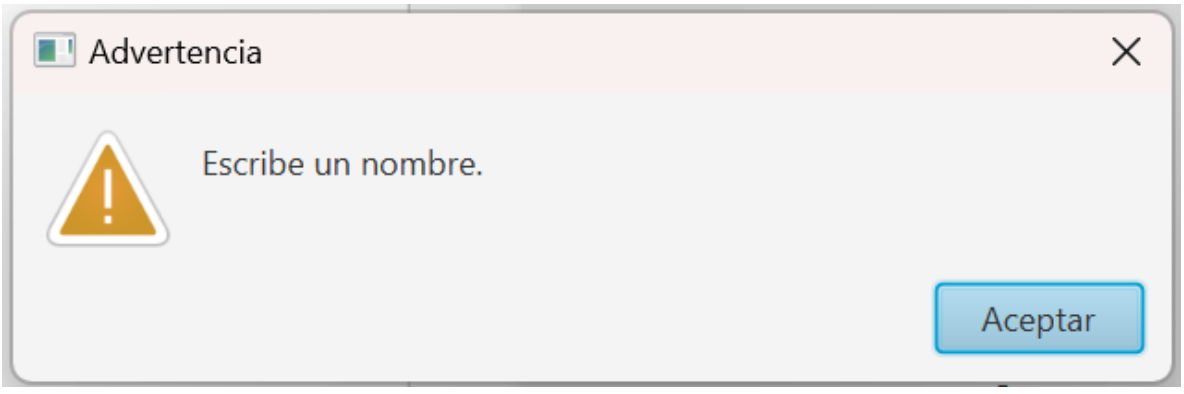
El servidor mandará al cliente cada 5 segundos un anuncio con la dirección multicast, y el el puerto multicast, download y search.



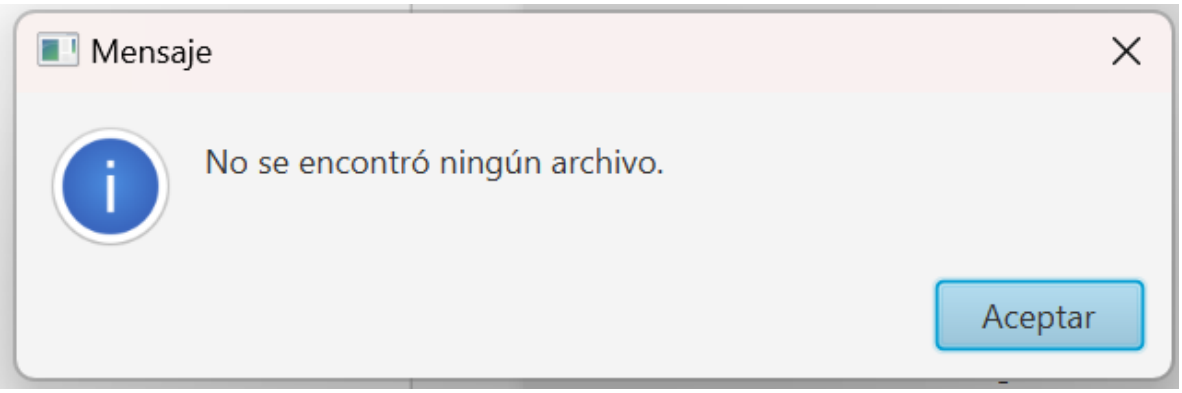
Para buscar archivos tenemos que buscar alguno que se encuentre en la carpeta establecida, la cual contiene los siguientes archivos.



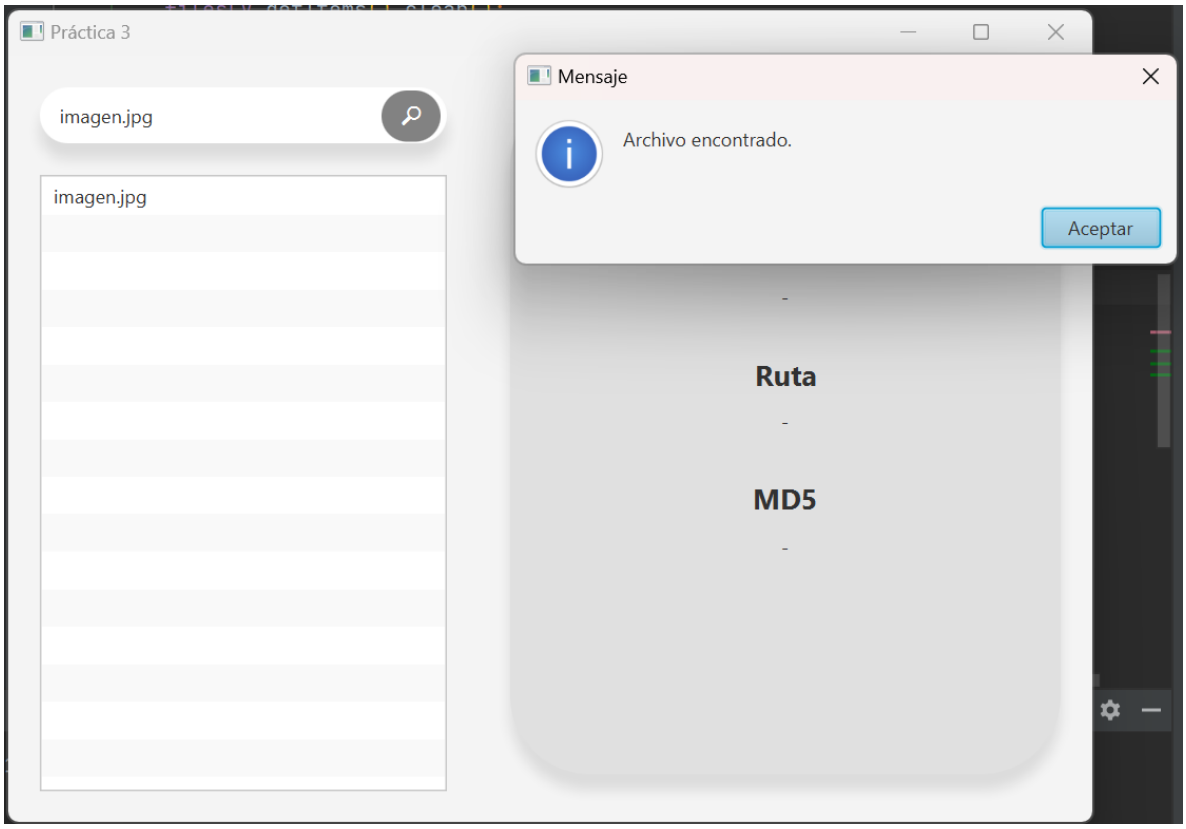
Si no escribimos un nombre y presionamos el botón de buscar mandará una advertencia.



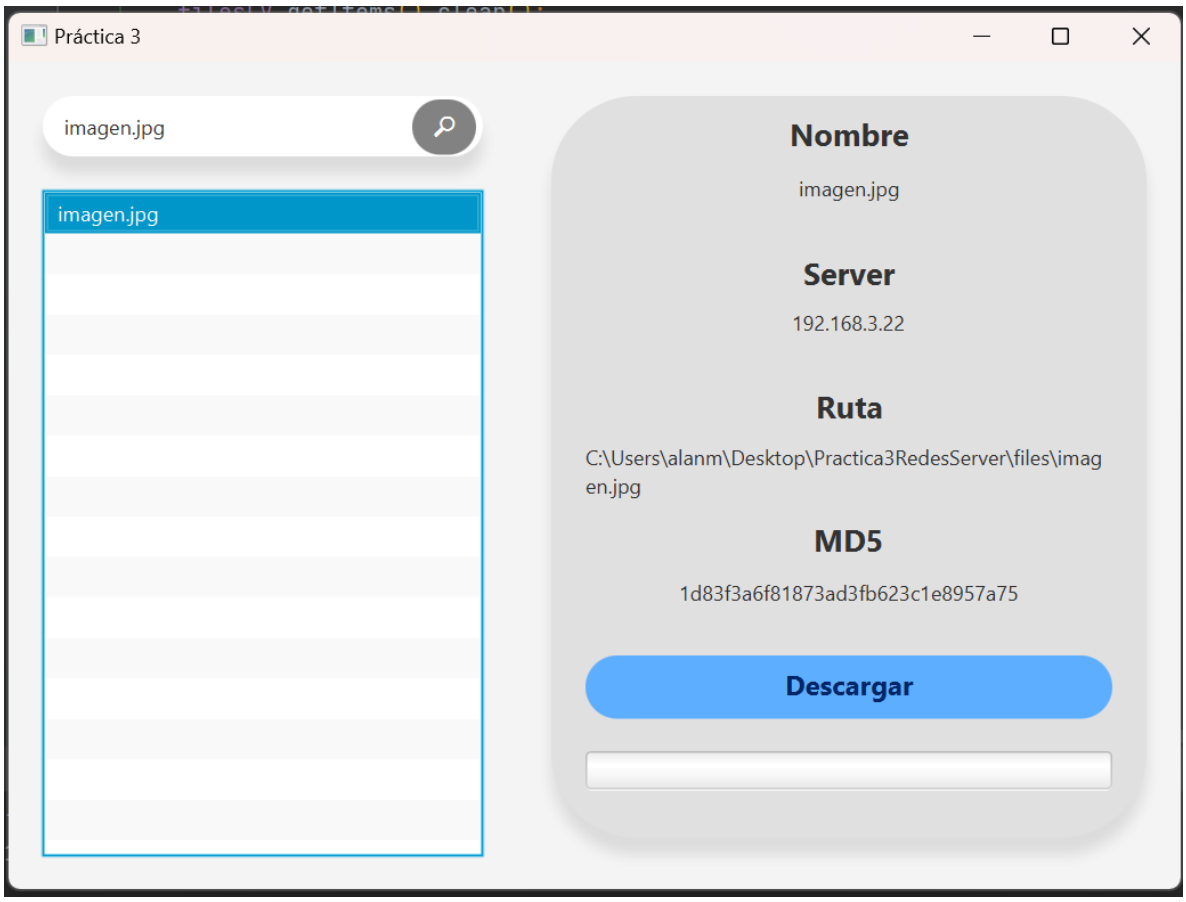
Si no encuentra el archivo mandará un mensaje.



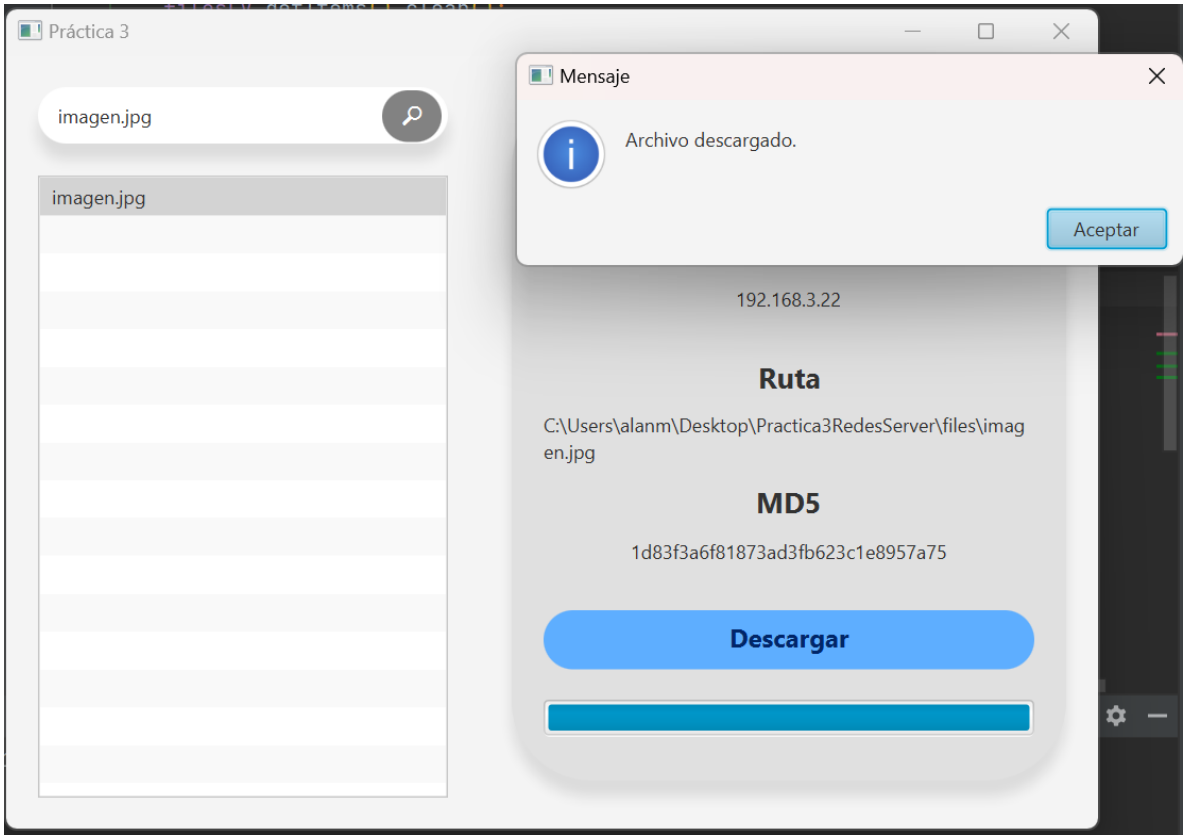
Si encuentra el archivo mandará un mensaje y agrega el archivo a la lista.

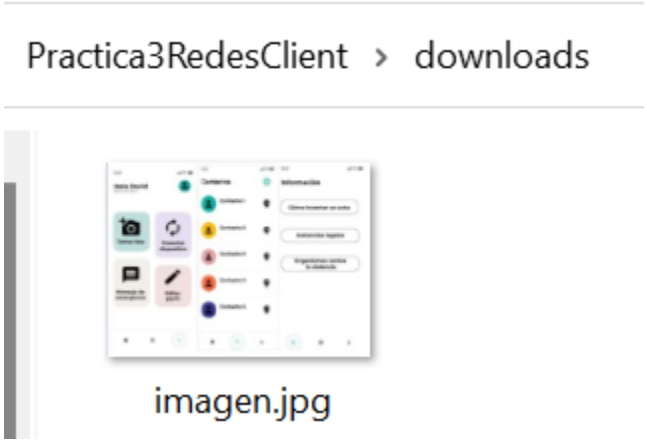


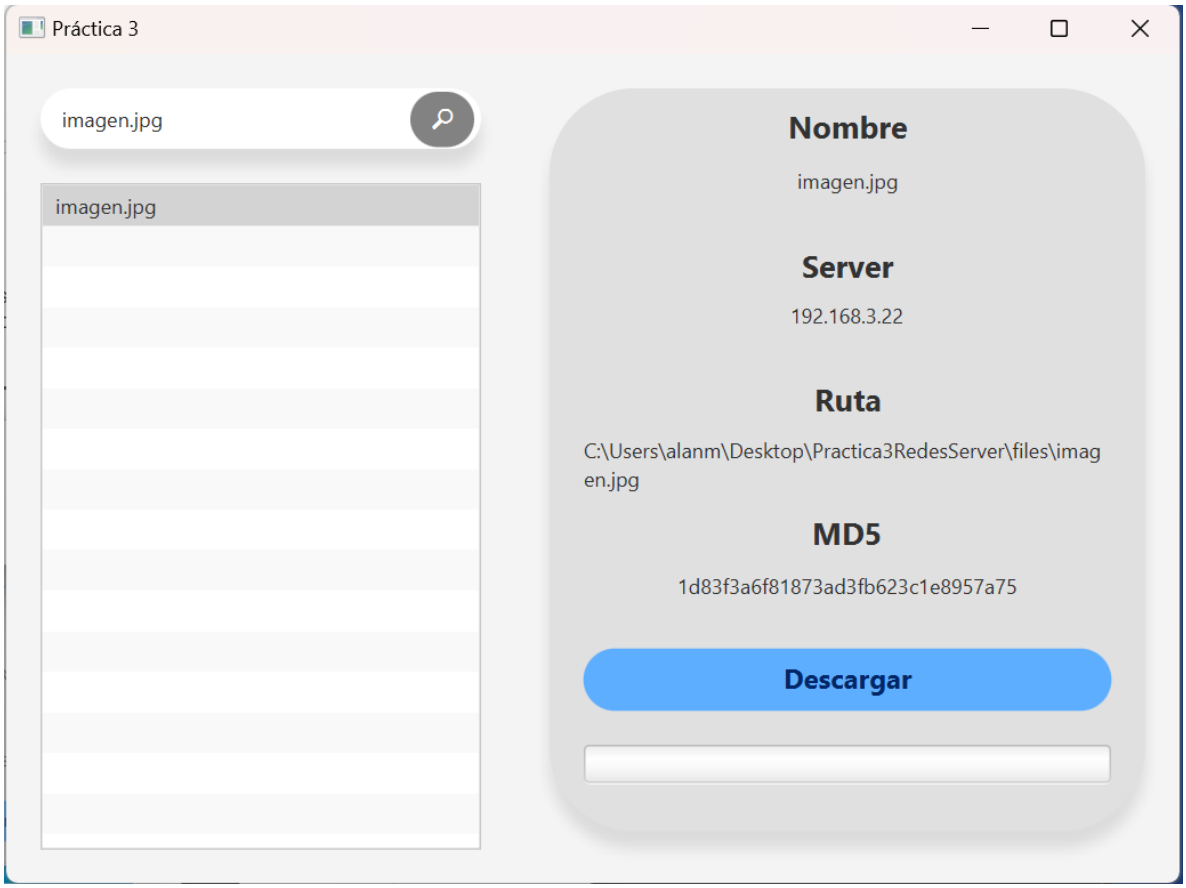
Al dar clic sobre el archivo encontrado de la lista mostrará su información también habilitará el botón para poder descargarlo y la barra de progreso de descarga.


Práctica 3
imagen.jpg
imagen.jpg
Nombre
imagen.jpg
Server
192.168.3.22
Ruta
C:\Users\alanm\Desktop\Practica3RedesServer\files\imagen.jpg
MD5
1d83f3a6f81873ad3fb623c1e8957a75
Descargar

Al dar clic en el botón de descargar mandará un mensaje, se guardará el archivo en la carpeta destinada para las descargas e irá cambiando el estado de la barra de progreso. Finalmente se restablecerá el estado de la barra de progreso. Se muestra en las siguientes tres imágenes.







4. Conclusiones

La práctica aplica conceptos importantes vistos durante el parcial. Primeramente, los conceptos más significativos fueron multicast e hilos que vimos nuevamente para búsqueda de los archivos. Como es un tema que vimos recientemente teníamos un poco de dificultad para elaborarlo. Esta práctica se encarga de crear un hilo y se detectan de disponibilidad de servidor mediante el hilo. últimamente implementamos DownloadServer, que utilizamos en la práctica 1 para envío de archivos. Como la implementación del servidor de flujo o DownloadServer fue igual que la anterior, no era necesario cambiar muchas cosas entonces fue más fácil implementarlo. Todavía no quedó del todo claro con la implementación del MD5Checksum, ya que solo utilizamos el código ya hecho por el profesor. El funcionamiento fue satisfactorio, y logramos repasar lo que aprendimos en el transcurso del parcial.

Bibliografía

- Tanenbaum, Andrew. “Redes de Computadoras”. Cuarta Edición, Pearson Prentice Hall, 2003
- Guía rápida de configuración para Multicast (Multidifusión). Cisco. (2022). Retrieved 10 May 2022, from https://www.cisco.com/c/es_mx/support/docs/ip/ipmulticast/9356-48.html.
- ¿Qué significa enlazar un socket de multidifusión (UDP)?. Foro Ayuda. (2022). Retrieved 10 May 2022, from https://foroayuda.es/que-significa-enlazar-un-socket-demultidifusion-udp/.
- Comportamiento de la opción de socket de multidifusión. Microsoft. (2022). Retrieved 10 May 2022, from https://docs.microsoft.com/eses/windows/win32/winsock/multicast-socket-option-behavior.
- Díaz, D. (2022). Introducción a la Concurrencia en Java (I). Blog.softtek.com. Retrieved 2 June 2022, from https://blog.softtek.com/es/java-concurrency.