



Instituto Politécnico Nacional  
Escuela Superior de Cómputo



Aplicaciones para Comunicaciones en Red

## **Práctica 1: “Servicio de transferencia de archivos”**

Alumnos:

Malagón Baeza Alan Adrian  
Martínez Chávez Jorge Alexis

Profesor:

Moreno Cervantes Axel Ernesto

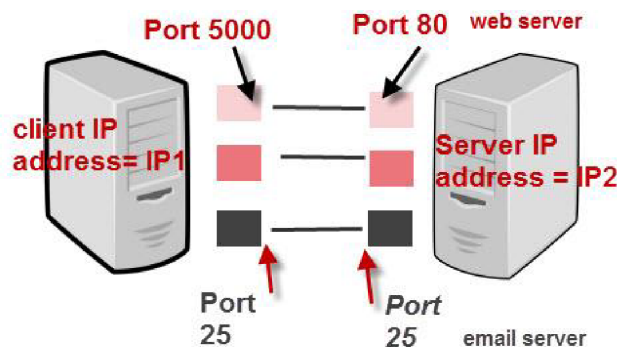
Grupo: 6CM1

# 1. Introducción

Un socket TCP/IP se utiliza para las comunicaciones entre dos computadoras. El socket incluye la dirección del protocolo de Internet (IP), así como el host o puerto de red el cual identifica la aplicación o el servicio que se ejecuta en la computadora.

Todas las aplicaciones que intervienen en la transmisión utilizan el socket para enviar y recibir información. Una dirección IP por sí sola no es suficiente para ejecutar aplicaciones de red, ya que una computadora puede ejecutar múltiples aplicaciones o servicios.

El siguiente diagrama muestra una conexión de computadora a computadora e identifica las direcciones IP y los puertos.



IP Address + Port number = Socket

## TCP/IP Ports And Sockets

El protocolo TCP/IP admite dos tipos de puerto: puerto TCP y puerto UDP.

- TCP: Es para aplicaciones orientadas a la conexión. Ha incorporado la verificación de errores y retransmite los paquetes faltantes.
- UDP: Es para aplicaciones sin conexión. No tiene verificación de errores incorporada y no retransmitirá los paquetes faltantes.

Las aplicaciones están diseñadas para usar el protocolo de capa de transporte UDP o TCP según el tipo de conexión que requieran. Por ejemplo, un servidor web normalmente usa el puerto TCP 80.

## 1.1 Servidor/Cliente

Normalmente, un servidor se ejecuta en una computadora específica y tiene un socket que está vinculado a un número de puerto específico. El servidor solo espera, escuchando el socket para que un cliente realice una solicitud de conexión.

Por otro lado, el cliente conoce el nombre de host de la máquina en la que se ejecuta el servidor y el número de puerto en el que escucha el servidor. Para realizar una solicitud de conexión, el cliente intenta reunirse con el servidor en la máquina y el puerto del servidor. El cliente también necesita identificarse ante el servidor para vincularse a un número de puerto local que utilizará durante esta conexión. Normalmente lo asigna el sistema.



Si todo va bien, el servidor acepta la conexión. Tras la aceptación, el servidor obtiene un nuevo socket vinculado al mismo puerto local y también tiene su punto final remoto establecido en la dirección y el puerto del cliente. Necesita un nuevo socket para que pueda continuar escuchando el socket original para las solicitudes de conexión mientras atiende las necesidades del cliente conectado.



En el lado del cliente, si se acepta la conexión, se crea correctamente un socket y el cliente puede usar el socket para comunicarse con el servidor. El cliente y el servidor ahora pueden comunicarse escribiendo o leyendo desde sus sockets.

## 2. Desarrollo

Esta práctica se desarrolló en Java, ya que dispone de toda una API para trabajar con sockets 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.

### 2.1 Socket Servidor

Los sockets servidor o *ServerSocket* permiten que aplicaciones Java puedan establecer una conexión en un equipo en un puerto determinado y de esa manera ser capaces de recibir conexiones de clientes para comunicarse con dicha aplicación.

Para establecer un socket servidor sólo es necesario indicar el puerto en el que la aplicación quedará “escuchando” las conexiones de los clientes, en nuestro caso el `HOST_PORT = 1201`.

Una vez establecida la conexión, la clase *ServerSocket* dispone del método *accept()* que bloquea la ejecución de la aplicación hasta que se recibe la conexión de un cliente. En ese momento se devuelve una referencia al socket de dicho cliente y es posible establecer los flujos de comunicación con el mismo para comenzar a dar servicio.

Como se observa en el siguiente código, el *ServerSocket* es creado en un bloque try-catch para el manejo de excepciones y la recepción de clientes se encuentra dentro de un ciclo for ever para que el servidor sea capaz de atender la petición de varios clientes.

```
private static final int HOST_PORT = 1201;

private static String folder = "remoto";

public static void main(String[] args) {
    // Establecemos una conexión
    try {
        // Crea un socket de servidor, vinculado al puerto especificado.
        ServerSocket s = new ServerSocket(HOST_PORT);

        // Habilita/deshabilita la opción de socket SO_REUSEADDR.
        s.setReuseAddress(true);

        System.out.println("Servidor iniciado esperando archivos..");
    }
}
```

```

        for ( ; ; ) {
            // Escucha la conexión que se realizará con este socket y la
acepta.
            Socket cl = s.accept();

            // Imprimimos la dirección y el número de puerto remoto
            // al que está conectado este socket
            System.out.println("\n\nCliente conectado desde " +
cl.getInetAddress() + ":" + cl.getPort());

            // Crea un flujo de entrada que usa el flujo de entrada
subyacente especificado.
            // #getInputStream : Devuelve un flujo de entrada para este
socket.
            DataInputStream dis = new DataInputStream(cl.getInputStream());

            // Bandera
            // #readInt : Lee cuatro bytes de entrada y devuelve un valor
int.
            int flag = dis.readInt();

            // Opciones
            switch (flag) {
                // 0 : Recibir archivos/carpeta
                case 0 -> descargarArchivo(dis);
                // 1 : Ver carpeta/archivos
                case 1 -> {
                    folder = dis.readUTF();
                    listarArchivos(dis, cl);
                }
                // 2 : Crear carpeta
                case 2 -> crearCarpeta(dis);
                // 3 : Eliminar archivos/carpeta
                case 3 -> eliminarArchivo(dis, cl);
                // 4 : Enviar archivos/carpeta
                case 4 -> {
                    String nombre = dis.readUTF();
                    String directory = dis.readUTF();
                    File f = new File(nombre);
                    enviarArchivo(dis, cl, f, directory);
                }
                // 5 : Renombrar archivos/carpeta
                case 5 -> renombrarArchivo(dis, cl);
            } // switch
            cl.close(); // Cierra este socket.
        } // for
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

## 2.1.1 Recibir archivos del cliente

Para la recepción de archivos se utilizó un método `descargarArchivo` el cual toma como parámetro un *`DataInputStream`*. El *`DataInputStream`* permite crear un flujo de entrada para leer el nombre del archivo, el directorio destino, el tamaño del archivo, además de una bandera que nos permite identificar si se trata de un archivo o directorio. Cuando la bandera está encendida nos indica que se trata de un archivo, cuando está apagada entonces se trata de un directorio.

```
// Descargar archivos y carpetas desde la carpeta local
private static void descargarArchivo(DataInputStream dis) throws
IOException {
    String nombre = dis.readUTF(); // Lectura nombre del archivo
    String directory = dis.readUTF(); // Obtenemos el directorio de destino
    long tam = dis.readLong(); // Lectura tamaño del archivo
    int flag = dis.readInt(); // Lectura flag opcion
```

En el caso de que sea un archivo, comienza su descarga y se almacena en el directorio destino (indicado por el cliente), si el directorio no existe se crea. Posterior a ello, se hace una instancia de la clase *`DataOutputStream`* con ella se establecerá 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, 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.

```
if (flag == 1) { // Es un archivo
    System.out.println("Comienza descarga del archivo " + nombre + " de " +
tam + " bytes\n");

    // Creamos directorio
    File f = new File("");
    String absoluta = f.getAbsolutePath();
    String ruta_archivos = absoluta + "\\ " + directory + "\\ ";

    File f2 = new File(ruta_archivos);
    f2.mkdirs();
    f2.setWritable(true);

    DataOutputStream dos = new DataOutputStream(new
FileOutputStream(ruta_archivos + nombre)); // Flujo de salida
    long recibidos = 0;
    int l = 0, porcentaje = 0, buffer = 1500;
    while (recibidos < tam) {
        byte[] b = new byte[buffer];
```

```

        l = dis.read(b);
        dos.write(b, 0, l); // dos.write(b)
        dos.flush();
        recibidos = recibidos + l;
        porcentaje = (int) ((recibidos * 100) / tam);
        System.out.println("Recibido el " + porcentaje + " % del archivo.
Bytes leídos: " + recibidos + "/" + tam);
    } // while
    System.out.println("Archivo " + nombre + " recibido..");
    dos.close();
} // if

```

Por otro lado, si se trata de un directorio, el servidor recibe a través del flujo de entrada establecido con el Cliente, nombre del directorio, su tamaño en bytes y directorio destino. Luego el servidor se encarga de crear el directorio correspondiente si no existe. Finalmente se cierran los flujos de entrada, salida y cliente.

```

        else { // Es una carpeta
            System.out.println("Comienza descarga del directorio " + nombre + "
de " + tam + " bytes");
            File file = new File(directory + "\\" + nombre);
            if (!file.exists()) {
                file.mkdir();
            }
        }
        dis.close();
    }
}

```

## 2.1.2 Enviar archivos al cliente

```

// Subir archivos y carpetas a la carpeta local
private static void enviarArchivo(DataInputStream dis, Socket cl, File f,
String directory) {
    try {
        String nombre = f.getName();
        String path = f.getAbsolutePath();
        long tam = f.length();

        if (f.isFile()) {
            System.out.println("\nPreparandose para enviar archivo " + path
+ " de " + tam + " bytes\n");

            DataOutputStream dos = new
DataOutputStream(cl.getOutputStream()); // Flujo de salida
            dos.writeUTF(nombre); // Envio nombre del archivo
            dos.flush();
            if (directory.isEmpty())
                dos.writeUTF("local" + "\\Descargas"); // Sends the target
            directory
        } else
    }
}

```

```

        dos.writeUTF("local" + "\\Descargas\\" + directory);
        dos.flush();
        dos.writeLong(tam); // Envio tamaño del archivo
        dos.flush();
        dos.writeInt(1); // Envio flag tipo de archivo
        dos.flush();

        long enviados = 0;
        int l = 0, porcentaje = 0, buffer = 1500;
        dis = new DataInputStream(new FileInputStream(path)); //Flujo
de entrada
        while (enviados < tam) {
            byte[] b = new byte[buffer];
            l = dis.read(b);
            dos.write(b, 0, l); // dos.write(b);
            dos.flush();
            enviados = enviados + l;
            porcentaje = (int) ((enviados * 100) / tam);
            System.out.println("Enviado el " + porcentaje + " % del
archivo.Bytes enviados: " + enviados + "/" + tam);
        } // while

        System.out.println("Archivo " + f.getName() + " enviado...");

        dis.close();
        dos.close();
        cl.close();
    } //if
    else {
        DataOutputStream dos = new
DataOutputStream(cl.getOutputStream()); // Flujo de salida
        dos.writeUTF(nombre); // Envio nombre del archivo
        dos.flush();
        if (directory.isEmpty())
            dos.writeUTF("local"); // Envio directorio destino
        else
            dos.writeUTF("local" + "\\" + directory); // Envio
directorio destino
        dos.flush();
        dos.writeLong(0); // Envio tamaño directorio
        dos.flush();
        dos.writeInt(0); // Envio flag tipo de archivo
        dos.flush();

        File folder = new File(f.getAbsolutePath());
        File[] files = folder.listFiles();
        if (files != null) {
            dos.writeInt(files.length); // Envio numero de archivos en
el directorio
            dos.flush();

            for (File fi : files) {
                dos.writeUTF(fi.getAbsolutePath()); // Envio directorio
destino
                dos.flush();
                dos.writeUTF(directory); // Envio directorio padre
                dos.flush();
            }
        }
    }
}

```



```

        dos.writeUTF(folder.getName()); // Envio directorio
actual
        dos.flush();
    }
    } // if
    else {
        dos.writeInt(0); // Envio numero de archivos en el
directorio
        dos.flush();
    } // else
    dis.close();
    dos.close();
    cl.close();
    } // else
} catch (IOException e) {
    throw new RuntimeException(e);
}
}

```

## 2.1.3 Visualizar archivos y carpetas

Para poder observar la lista de archivos y directorios que se encuentran en el directorio del Servidor se hace uso del método `listarArchivos()` en donde después de establecer la conexión con el cliente, flujo de salida y de entrada, se procede a obtener el listado de archivos, si la carpeta está vacía se le informa al cliente, sino se procede a hacer el listado de archivos y directorios.

```

/* Visualizar carpetas (y su contenido), así como archivos almacenados
local y remotamente
*/
public static void listarArchivos(DataInputStream dis, Socket cl) throws
IOException {
    // Obtenemos el directorio
    File f = new File("");
    String absoluta = f.getAbsolutePath();
    String ruta_archivos = absoluta + "\\\" + folder;

    // Obtenemos los archivos en el directorio indicado
    File f2 = new File(ruta_archivos);
    File[] fileList = f2.listFiles();
    String name = "";
    List<Directory> directorios = new ArrayList<>();
    if (fileList != null) {
        Directory directory;
        for (int i = 0; i < fileList.length; i++) {
            directory = new Directory();
            name = fileList[i].getName();
            if (fileList[i].isDirectory()) {
                name = name + "\\\";
                directory.setImgSrc("folder.png");
            } else {
                directory.setImgSrc("file.png");
            }
        }
    }
}

```

```

        directory.setName(name);
        directory.setSize(fileList[i].length());
        directorios.add(directory);
    }
} else {
    System.out.println("Carpeta Vacía");
}
// Crea un nuevo flujo de salida de datos para escribir datos en el
flujo de salida subyacente especificado.
ObjectOutputStream oos = new ObjectOutputStream(cl.getOutputStream());
// Escribe un objeto en el flujo de salida subyacente
oos.writeObject(directorios);
// Cierra este flujo de salida/entrada y libera cualquier recurso del
sistema asociado con el flujo.
oos.flush();

oos.close();
dis.close();
}

```

## 2.1.4 Crear carpeta

```

// Crear carpetas local y remotamente
public static void crearCarpeta(DataInputStream dis) throws IOException {
    String nombre = dis.readUTF(); // Lectura nombre del directorio
    String directory = dis.readUTF(); // Lectura directorio destino

    File file = new File(directory + "\\\" + nombre);
    if (!file.exists()) {
        file.mkdir();
    }
}

```

## 2.1.5 Eliminar archivos y carpetas

Para eliminar archivos o directorios del servidor se utiliza el método `eliminarArchivo()`, después del establecimiento de la conexión con el cliente, se procede a obtener la ruta absoluta del archivo o directorio a eliminar, para ello se comprueba que exista, después se evalúa si se trata de un directorio o un archivo y se procede con la eliminación. Finalmente, se le despliega al cliente que la eliminación del archivo o carpeta fue exitosa.

```

// Eliminar carpetas local y remotamente
public static void eliminarArchivo(DataInputStream dis, Socket cl) throws
IOException {
    // Obtenemos el directorio
    File f = new File("");
    String absoluta = f.getAbsolutePath();
    String path = dis.readUTF(); // Lectura directorio padre
}

```

```

String target = dis.readUTF(); // Lectura directorio destino
String ruta_archivos = absoluta + "\\\" + path + "\\\" + target;

File f2 = new File(ruta_archivos);
String output;
if (f2.exists()) {
    if (f2.isDirectory()) {
        eliminarCarpeta(f2);
        f2.delete();
        output = "Se eliminó el directorio";
    } else {
        if (f2.delete()) {
            output = "Se eliminó el archivo";
        } else {
            output = "No se pudo eliminar el archivo";
        }
    }
} else {
    output = "No se encontró el archivo a eliminar";
}
DataOutputStream dos = new DataOutputStream(cl.getOutputStream()); //
Flujo de salida
dos.writeUTF(output); // Envio mensaje de exito
dos.flush();
dos.close();
dis.close();
}

```

```

// Eliminar contenidos de una carpeta
private static void eliminarCarpeta(File f2) {
    File[] allContents = f2.listFiles();
    if (allContents != null) {
        for (File file : allContents) {
            if (file.isDirectory()) eliminarCarpeta(file);
            file.delete();
        }
    }
}

```

## 2.1.6 Renombrar archivos y carpetas

Para renombrar archivos o directorios del servidor se utiliza el método `renombrarArchivo()`, después del establecimiento de la conexión con el cliente, se procede a obtener la ruta absoluta del archivo o directorio a Finalmente, se le despliega al cliente que el renombramiento del archivo o carpeta fue exitoso

```

// Renombrar archivos/carpetas local o remotamente
public static void renombrarArchivo(DataInputStream dis, Socket cl) throws
IOException {
    File f = new File("");
    String absoluta = f.getAbsolutePath();
    String path = dis.readUTF(); // Lectura directorio a cambiar
}

```

```
String rename = dis.readUTF(); // Lectura directorio nombre nuevo
String ruta_nombre = absoluta + "\\\" + path;
String ruta_rename = absoluta + "\\\" + rename;

File f2 = new File(ruta_nombre);
File f3 = new File(ruta_rename);

boolean flag = f2.renameTo(f3);
String output;
if (flag) {
    output = "Archivo correctamente renombrado";
} else {
    output = "No se pudo renombrar.";
}
DataOutputStream dos = new DataOutputStream(cl.getOutputStream()); //
Flujo de salida
dos.writeUTF(output); // Envio mensaje de exito
dos.flush();
dos.close();
dis.close();
}
```

## 2.2 Socket Cliente

### 2.2.1 Parámetros de la comunicación

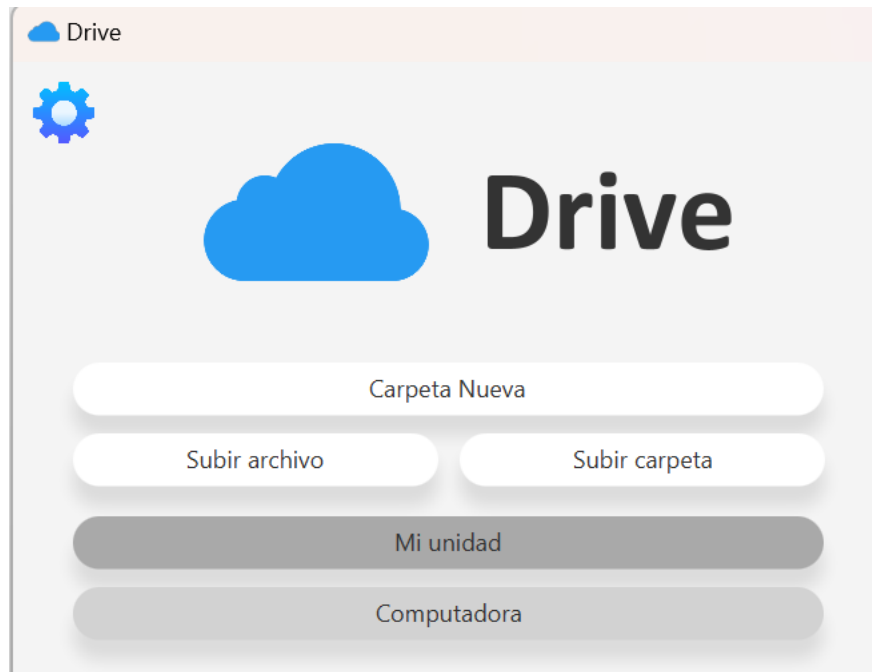
Para el socket en cliente, se instancia la clase de tipo Options, esta clase almacena todas las opciones que el usuario establezca en un cuadro de diálogo que se mostrará posteriormente. La clase cuenta con “getters” y “setters” para acceder a las variables encapsuladas que contendrán las últimas configuraciones seleccionadas por el usuario, de igual manera implementa la clase Serializable para poder guardar la clase en un archivo.

```
public class Options implements Serializable {  
  
    private String host, directory;  
  
    private int port, receiveBufferSize, sendBufferSize, timeout,  
lingerLong;  
    private boolean lingerStatus;  
  
    // Constructor  
    public Options() {  
        this.host = "localhost";  
        this.directory = "remoto";  
        this.port = 1201;  
        this.receiveBufferSize = 1500;  
        this.sendBufferSize = 1500;  
        this.timeout = 60;  
        this.lingerLong = 30;  
        this.lingerStatus = true;  
    }  
}
```

Al iniciar el programa para el cliente se crea un objeto nuevo de tipo Options con las configuraciones predeterminadas

```
private static Options options;  
  
static {  
    options = new Options();  
}
```

Al dar clic en el icono de ajustes (engranaje azul esquina superior izquierda) se mostrará la ventana para modificar las opciones



Opciones

**Opciones**

Búfer de entrada

Búfer de escritura

1500

Temporizador de lectura

60

Linger Status

true

Linger Long

30

Algoritmo de Nagle

true

Aceptar Cancelar

Si el usuario le da al botón de “Aceptar”, se cambiarán los valores del objeto de tipo Options, se le establecerán los nuevos valores elegidos por el usuario

```
public static void guardarOpciones(String bEntrada, String bEscritura,
String lLong, String lStatus, String nagle,String temporizador ) {
    options.setSendBufferSize(Integer.parseInt(bEscritura));
    options.setReceiveBufferSize(Integer.parseInt(bEntrada));
    options.setTimeout(Integer.parseInt(temporizador));
    options.setLingerLong(Integer.parseInt(lLong));
    options.setLingerStatus(Boolean.parseBoolean(lStatus));
    options.setTcpNoDelay(Boolean.parseBoolean(nagle));
    System.out.println(options);
    Alert a = new Alert(Alert.AlertType.INFORMATION);
    a.setHeaderText(null);
    a.setContentText("Opciones guardadas..");
}
```

```
a.show();  
}
```

### 2.2.2 Subir archivos al servidor

Para subir un archivo al servidor se utiliza el método `subirArchivo(file f,String directory)` en donde se recibe como parámetro un archivo o directorio y los directorios padres. Posteriormente, se hacen las modificaciones en la comunicación que el cliente solicitó. Luego se obtiene el nombre del archivo, la ruta (absolute path) y tamaño, si se trata de un archivo se prepara este archivo estableciendo un flujo de salida, y enviando los correspondientes bytes del archivo, debido a que el archivo no se manda completo, para eso se tiene un buffer que limita la cantidad de bytes que se envían del cliente al servidor. Cuando se envía completamente el archivo se le notifica al cliente.

Por otro lado, para un directorio se procede a llamar recursivamente al método con cada uno de los archivos o directorios que se encuentren dentro. Finalmente, en ambos casos al finalizar el envío de archivos y directorios se procede a cerrar los respectivos flujos de entrada y salida. En la siguiente captura se puede observar lo que fue descrito anteriormente.

```
// Subir archivos y carpetas a la carpeta remota  
public static void subirArchivo(File f, String directory) {  
    try {  
        Socket cl = new Socket(options.getHost(), options.getPort());  
        cl.setReceiveBufferSize(options.getReceiveBufferSize());  
        cl.setSendBufferSize(options.getSendBufferSize());  
        cl.setSoTimeout(options.getTimeout());  
        cl.setSoLinger(options.getLingerStatus(), options.getLingerLong());  
  
        String nombre = f.getName();  
        String path = f.getAbsolutePath();  
        long tam = f.length();  
  
        DataOutputStream dos = new DataOutputStream(cl.getOutputStream());  
        // Flujo de salida  
        dos.writeInt(0); // Envio bandera  
        dos.flush();  
        dos.writeUTF(nombre); //Envio nombre del archivo  
        dos.flush();  
  
        if (f.isFile()) {  
            System.out.println("\nPreparandose para enviar archivo " + path  
+ " de " + tam + " bytes\n");  
  
            if (directory.isEmpty())  
                dos.writeUTF(options.getDirectory()); // Envio directorio
```



```

destino
        else
            dos.writeUTF(options.getDirectory() + "\\\" + directory); //
Envio directorio destino
            dos.flush();
            dos.writeLong(tam); // Envio tamaño del archivo
            dos.flush();
            dos.writeInt(1); // Envio flag tipo de archivo
            dos.flush();

            long enviados = 0;
            int l = 0, porcentaje = 0, buffer =
options.getSendBufferSize();
            DataInputStream dis = new DataInputStream(new
FileInputStream(path)); //Flujo de entrada
            while (enviados < tam) {
                byte[] b = new byte[buffer];
                l = dis.read(b);
                dos.write(b, 0, l); // dos.write(b);
                dos.flush();
                enviados = enviados + l;
                porcentaje = (int) ((enviados * 100) / tam);
                System.out.println("Enviado el " + porcentaje + " % del
archivo.Bytes enviados: " + enviados + "/" + tam);
            } // while
            System.out.println("Archivo " + f.getName() + " enviado...");
            dis.close();
        } //if
        else {
            if (directory.isEmpty())
                dos.writeUTF(options.getDirectory()); // Envio directorio
destino
            else
                dos.writeUTF(options.getDirectory() + "\\\" + directory); //
Envio directorio destino
                dos.flush();
                dos.writeLong(tam); //Envio tamaño del archivo
                dos.flush();
                dos.writeInt(0); // Envio flag tipo de archivo
                dos.flush();

                File folder = new File(f.getAbsolutePath());
                File[] files = folder.listFiles();
                assert files != null;
                for (File fi : files) {
                    subirArchivo(fi,directory+"\\\"+folder.getName());
                } // for
            } // else
            dos.close();
            cl.close();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}

```

## 2.2.3 Visualizar archivos

```
/* Visualizar carpetas (y su contenido), así como archivos almacenados
local y remotamente
*/
public static List<Directory> visualizarArchivos(String directory) {
    List<Directory> directories = new ArrayList<>();
    try {
        Socket cl = new Socket(options.getHost(), options.getPort());
        DataOutputStream dos = new DataOutputStream(cl.getOutputStream());

        dos.writeInt(1); // Envio de bandera opcion
        dos.flush();
        dos.writeUTF(directory); // Envio directorio destino
        dos.flush();

        ObjectInputStream ois = new ObjectInputStream(cl.getInputStream());
// Flujo de entrada
        directories = (List<Directory>) ois.readObject(); // Lectura
archivos/carpetas en formato

        ois.close();
        dos.close();
        cl.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
    return directories;
}
```

## 2.2.4 Eliminar archivos del servidor

```
// Eliminar carpetas local y remotamente
public static void eliminarCarpeta(String path, String target) {
    try {
        Socket cl = new Socket(options.getHost(), options.getPort());
        cl.setReceiveBufferSize(options.getReceiveBufferSize());
        cl.setSendBufferSize(options.getSendBufferSize());
        cl.setSoTimeout(options.getTimeout());
        cl.setSoLinger(options.getLingerStatus(), options.getLingerLong());

        DataOutputStream dos = new DataOutputStream(cl.getOutputStream());
// Flujo de salida
        dos.writeInt(3); // Envio bandera opcion
        dos.flush();
        dos.writeUTF(path); // Envio nombre del archivo dos.flush();
        dos.flush();
        dos.writeUTF(target); // Envio directorio destino

        DataInputStream dis = new DataInputStream(cl.getInputStream());
        String msg = dis.readUTF(); // Lectura mensaje de exito
    }
}
```

```

        Alert a = new Alert(Alert.AlertType.INFORMATION);
        a.setHeaderText(null);
        a.setContentText(msg);
        a.show();

        dis.close();
        dos.close();
        cl.close();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}

```

## 2.2.5 Renombrar archivos del servidor .

```

// Renombrar archivos/carpetas local o remotamente
public static void renombrarArchivo(String path, String rename) {
    try {
        Socket cl = new Socket(options.getHost(), options.getPort());
        cl.setReceiveBufferSize(options.getReceiveBufferSize());
        cl.setSendBufferSize(options.getSendBufferSize());
        cl.setSoTimeout(options.getTimeout());
        cl.setSoLinger(options.getLingerStatus(), options.getLingerLong());

        DataOutputStream dos = new DataOutputStream(cl.getOutputStream());
// Flujo de salida

        dos.writeInt(5); // Envio bandera opcion
        dos.flush();
        dos.writeUTF(path); // Envio nombre del archivo dos.flush();
        dos.flush();
        dos.writeUTF(rename); // Envio directorio destino
        dos.flush();

        DataInputStream dis = new DataInputStream(cl.getInputStream()); //
Flujo de entrada
        String msg = dis.readUTF(); // Lectura mensaje de exito
        Alert a = new Alert(Alert.AlertType.INFORMATION);
        a.setHeaderText(null);
        a.setContentText(msg);
        a.show();

        dis.close();
        dos.close();
        cl.close();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}

```

## 2.2.6 Descargar archivos

```

// Descargar archivos y carpetas desde la carpeta remota
public static void descargarArchivo(String nombre, String directory) {
    try{
        Socket cl = new Socket(options.getHost(), options.getPort());
        cl.setReceiveBufferSize(options.getReceiveBufferSize());
        cl.setSendBufferSize(options.getSendBufferSize());
        cl.setSoTimeout(options.getTimeout());
        cl.setSoLinger(options.getLingerStatus(), options.getLingerLong());

        DataOutputStream dos = new DataOutputStream(cl.getOutputStream());
// Flujo de salida
        dos.writeInt(4); // Envio bandera opcion
        dos.flush();
        dos.writeUTF(nombre); // Envio nombre del archivo
        dos.flush();
        dos.writeUTF(directory); // Envio directorio destino
        dos.flush();

        DataInputStream dis = new DataInputStream(cl.getInputStream()); //
Flujo de entrada
        String nombre_ = dis.readUTF(); // Lectura nombre del archivo
        String directory_ = dis.readUTF(); // Lectura directio destino
        long tam = dis.readLong(); // Lectura tamaño del archivo
        int flag = dis.readInt(); // Lectura flag tipo de archivo

        if (flag == 1) { // Es un archivo
            System.out.println("Comienza descarga del archivo " + nombre_ +
" de " + tam + " bytes\n");

            // Creamos directorio
            File f = new File("");
            String absoluta = f.getAbsolutePath();
            String ruta_archivos = absoluta + "\\\" + directory_ + "\\\";

            File f2 = new File(ruta_archivos);
            f2.mkdirs();
            f2.setWritable(true);

            dos = new DataOutputStream(new
FileOutputStream(ruta_archivos+"\\\"+nombre_)); // Flujo de salida

            long recibidos = 0;
            int l = 0, porcentaje = 0, buffer = 1500;

            while (recibidos < tam) {
                byte[] b = new byte[buffer];
                l = dis.read(b);
                dos.write(b, 0, l);
                dos.flush();
                recibidos = recibidos + l;
                porcentaje = (int) ((recibidos * 100) / tam);
                System.out.println("Recibido el " + porcentaje + " % del
archivo. Bytes leídos: " + recibidos + "/" + tam);
            } // while
            System.out.println("Archivo " + nombre_ + " recibido..");
        } // if
        else { // Es una carpeta

```

```

        System.out.println("Comienza descarga del directorio " +
nombre_ + " de " + tam + " bytes");

        File file = new File("local\\Descargas\\"+directory_+"\\")+
nombre_);
        if (!file.exists()) {
            file.mkdir();
        }

        int nfiles = dis.readInt(); // Lectura numero archivos
contenidos en la carpeta
        for (int i = 0; i < nfiles; i++) {
            String nombreArchivo = dis.readUTF(); // Lectura nombre del
archivo
            String directoryName = dis.readUTF(); // Lectura directorio
destino
            String folderName = dis.readUTF(); // Lectura directorio
actual

descargarArchivo(nombreArchivo,directoryName+"\\")+folderName);
        } // for
    } // else

    dos.close();
    dis.close();
    cl.close();
} catch (IOException e) {
    throw new RuntimeException(e);
}
}

```

## 2.2.7 Crear carpeta

```

// Crear carpetas local y remotamente
public static void crearCarpeta(String path, String nombre) {
    try {
        Socket cl = new Socket(options.getHost(), options.getPort());
        cl.setReceiveBufferSize(options.getReceiveBufferSize());
        cl.setSendBufferSize(options.getSendBufferSize());
        cl.setSoTimeout(options.getTimeout());
        cl.setSoLinger(options.getLingerStatus(), options.getLingerLong());

        DataOutputStream dos = new DataOutputStream(cl.getOutputStream());
// Flujo de salida
        dos.writeInt(2);
        dos.flush();
        dos.writeUTF(nombre); //Envio nombre del archivo dos.flush();
        dos.flush();
        dos.writeUTF(path); // Sends the target directory
        dos.flush();

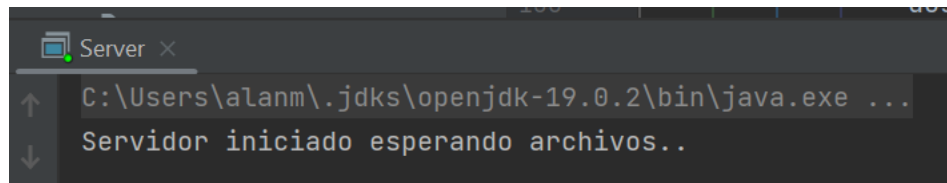
        dos.close();
        cl.close();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}

```

```
}  
}
```

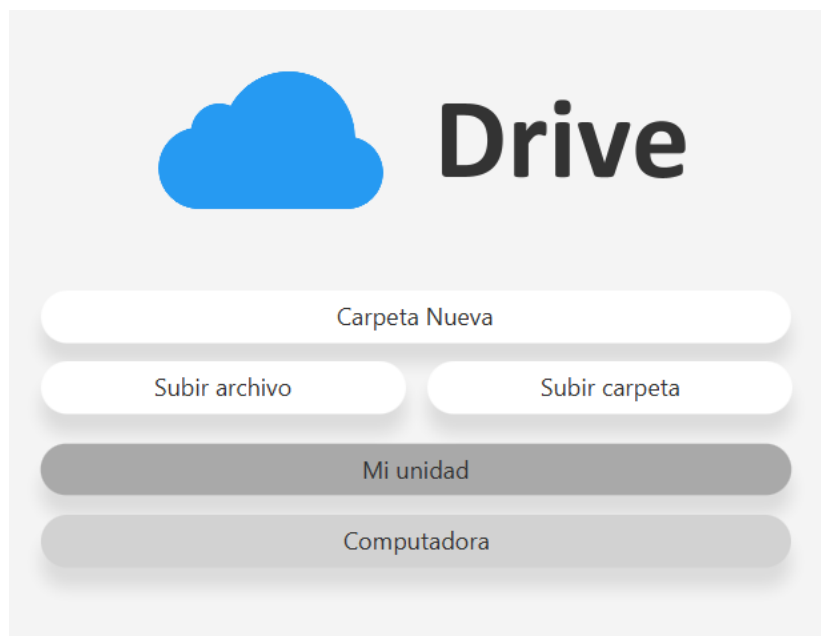
### 3. Pruebas

Primeramente se ejecuta la clase Servidor, para que esta quede en espera de las peticiones del cliente.

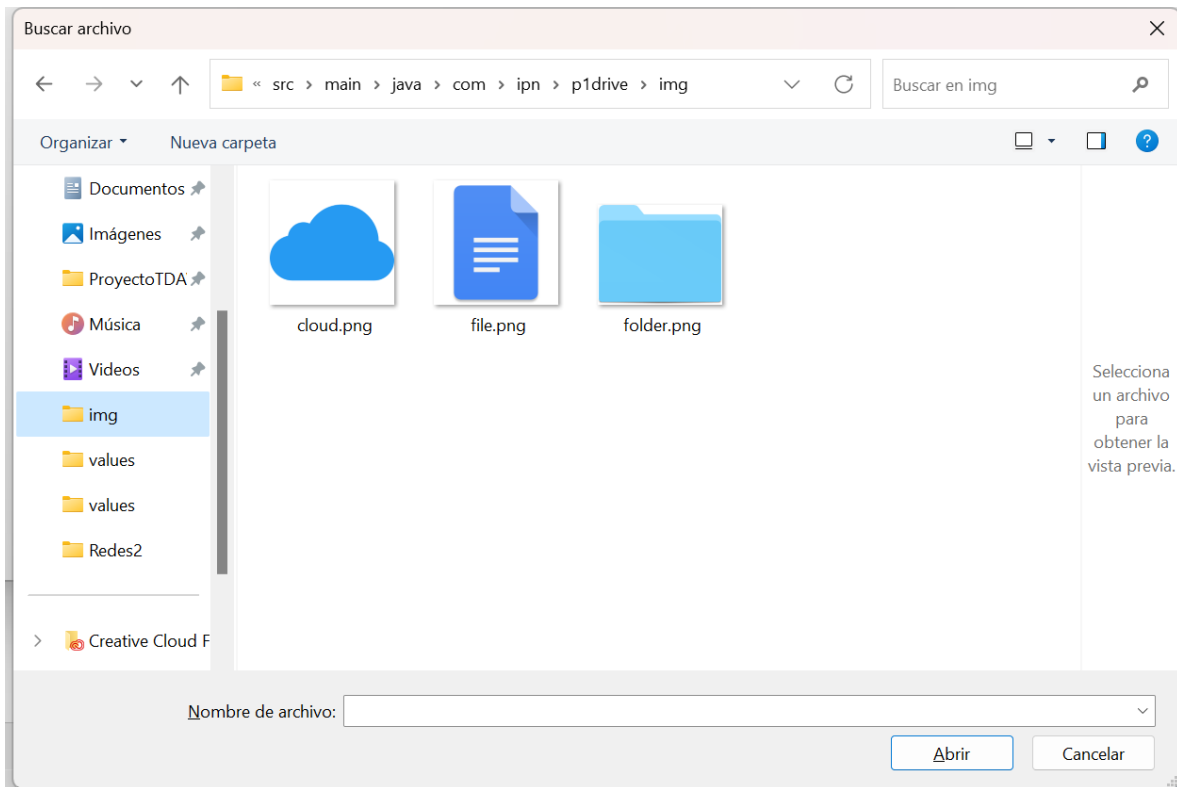


```
Server x
C:\Users\alanm\.jdk\openjdk-19.0.2\bin\java.exe ...
Servidor iniciado esperando archivos..
```

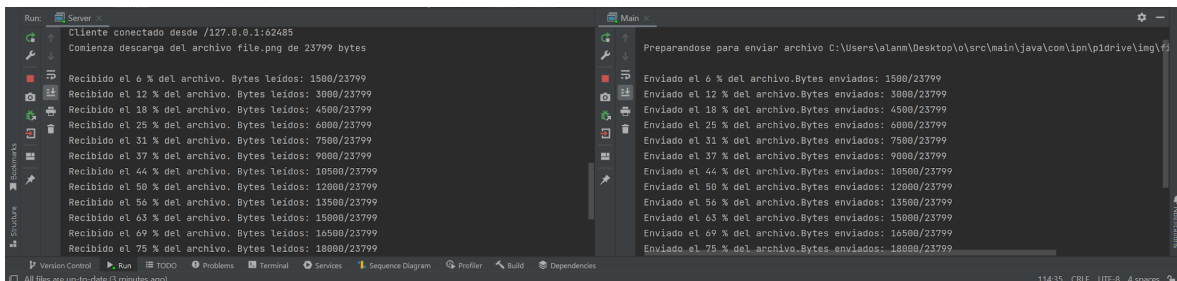
Después se ejecuta la clase Cliente. Al ejecutar se le despliega al cliente un menú en el que podrá elegir una de las siguientes acciones: Carpeta Nueva, Subir archivo (Servidor), Subir carpeta (Servidor), Mi unidad (Carpeta remota), Computadora (Carpeta local).



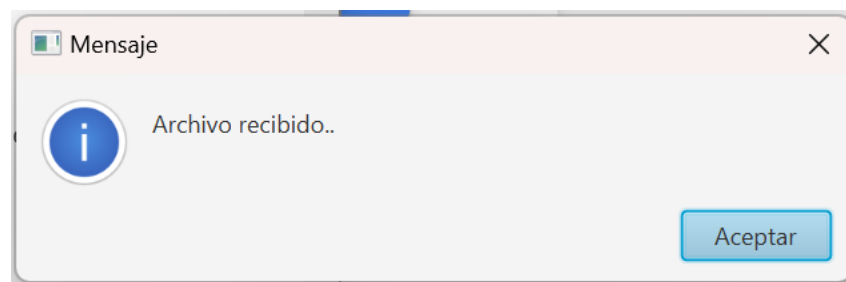
Si el cliente desea subir archivos al servidor se abre el FileChooser en donde podrá seleccionar el archivo que desee enviar al servidor.



Inmediatamente después de dar clic en Abrir comienzan a enviarse el archivo a la carpeta del Servidor. Como se ve en la siguiente captura el cliente puede observar los archivos que se fueron enviando al servidor. Mientras que en el servidor se comienzan a recibir cada uno de los archivos o carpetas.

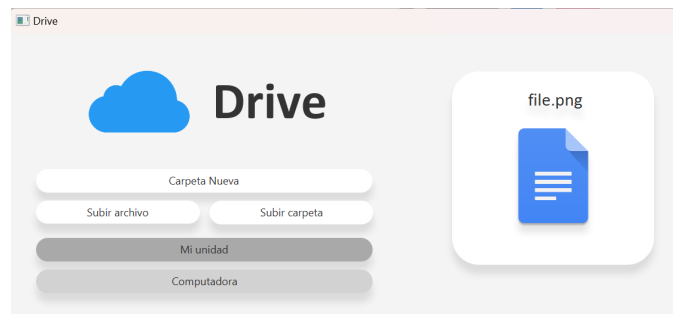


Al finalizar se manda mensaje de éxito.

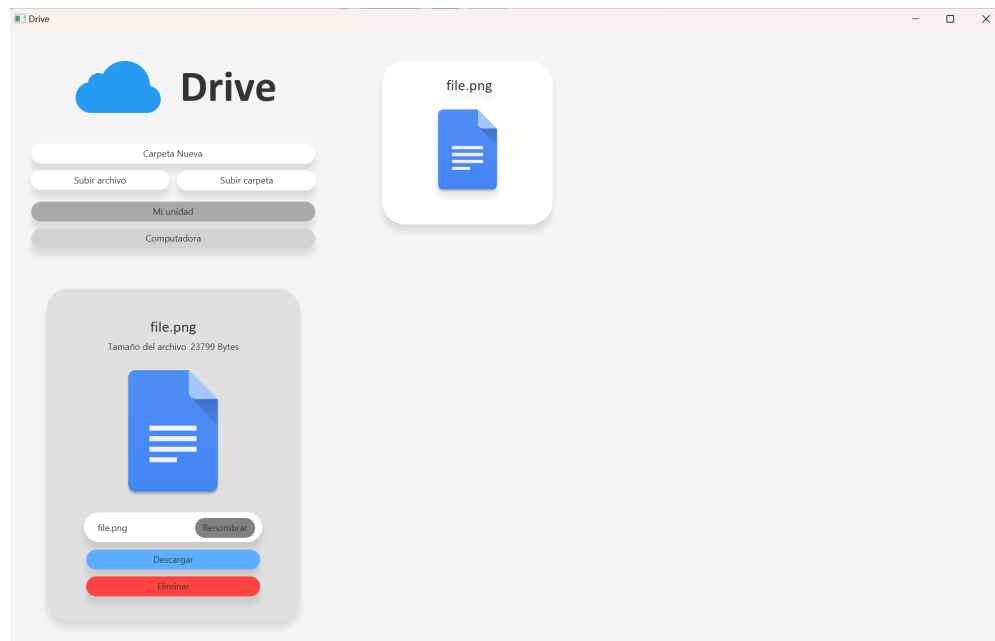




Y se actualiza la interfaz principal con una lista con los archivos y carpetas que existen en la carpeta remota.

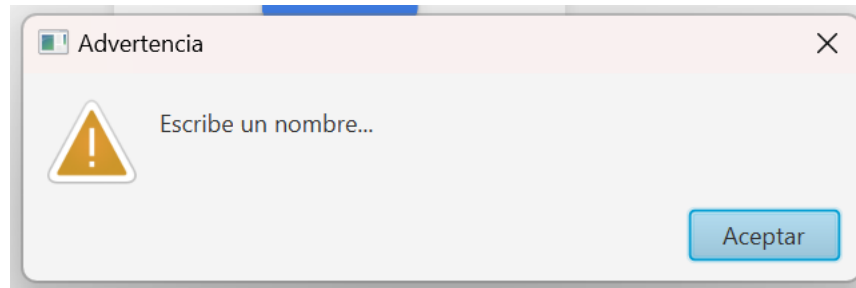


Si el cliente desea renombrar, descargar (remoto a local) y eliminar debe hacer click sobre la Card (cuadrado blanco) del archivo), lo cual mostrará debajo del botón Computadora el nombre del archivo o carpeta, su tamaño, el campo de texto y botón para Renombrar, botón de Descargar y botón de Eliminar

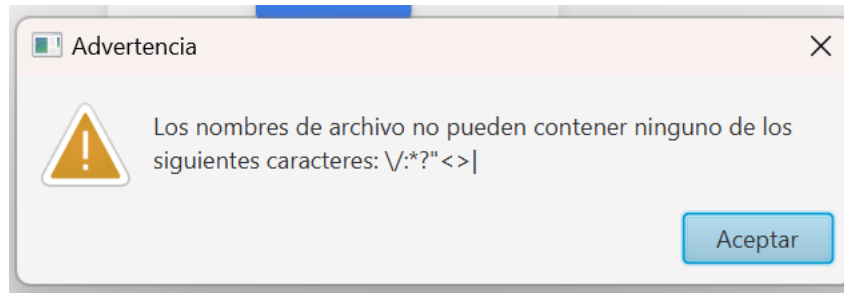


Para renombrar deberá modificar el campo de texto con algún nombre válido; en caso de no ser válido enviará mensajes sobre los errores; y para finalizar dar click al botón Renombrar el cual mandará mensaje de éxito.

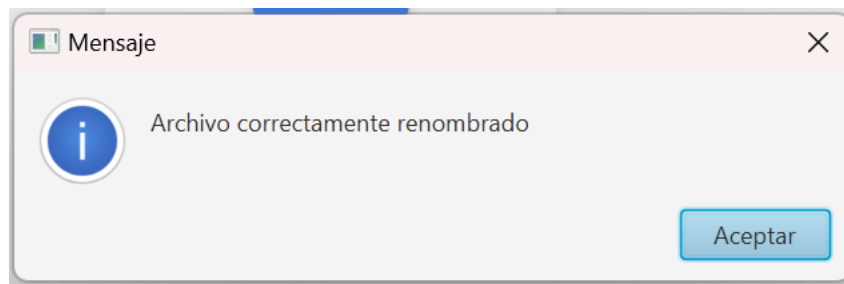
Mensaje en caso de estar vacío el campo.



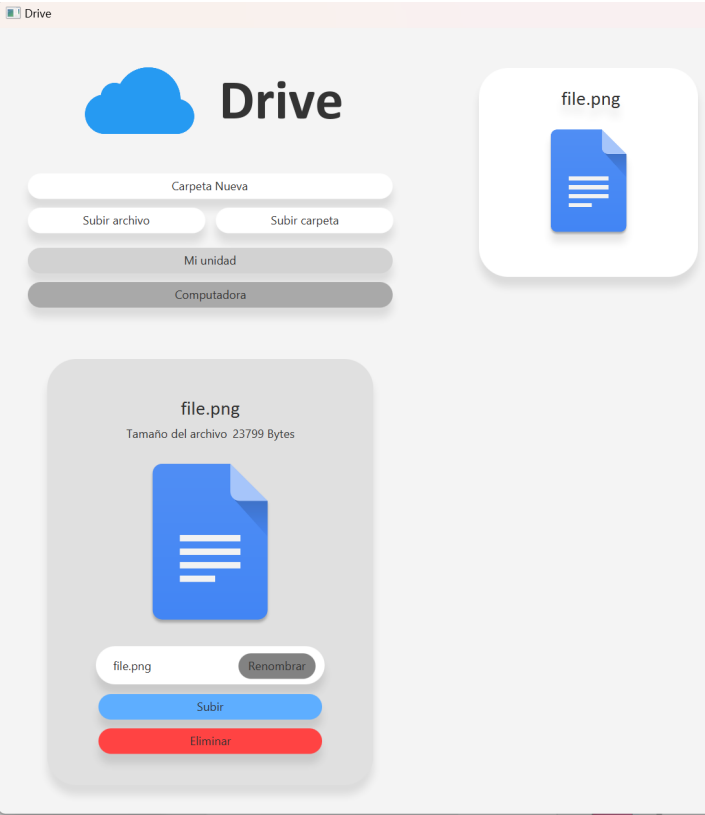
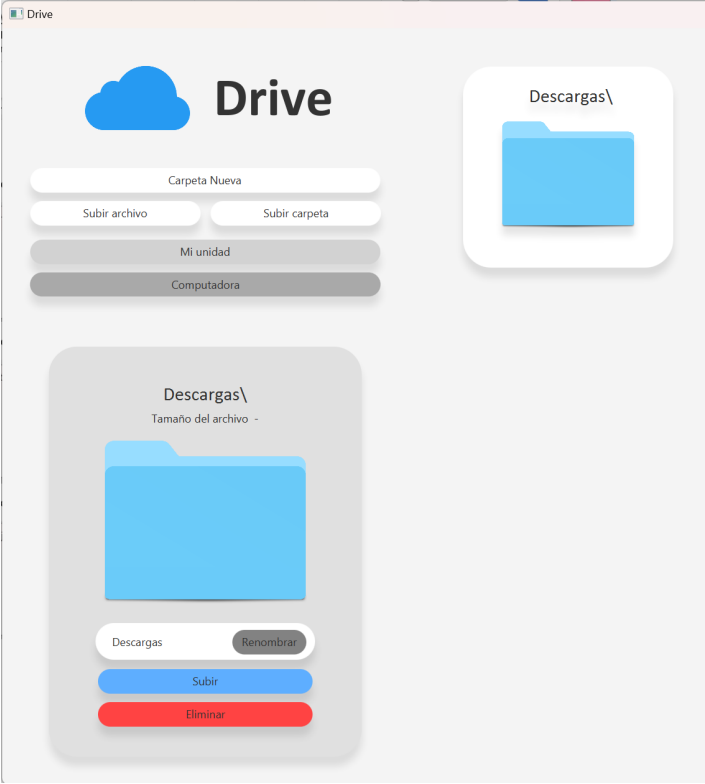
Mensaje en caso de que contenga un carácter especial.



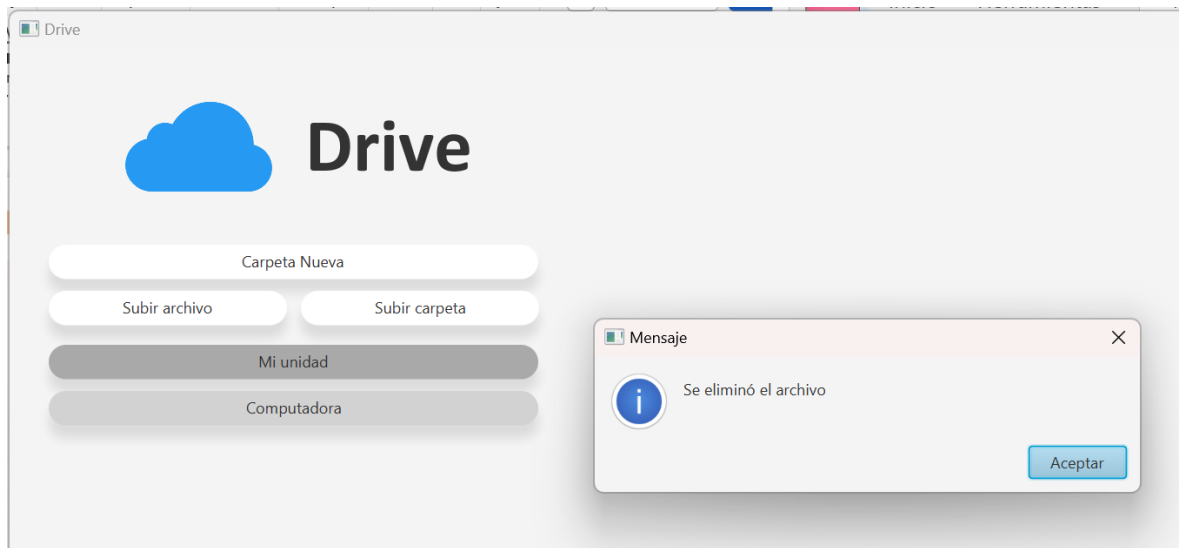
Mensaje de éxito.



Para descargar se da click en el botón Descargar, el cual creara la carpeta Descargas\ en la carpeta local (para mostrar su contenido se deberá hacer doble click en la Card) y subirá el archivo o carpeta que se seleccionó para descargar.



Para eliminar se da click en el botón Eliminar el cual mandará mensaje de éxito y actualizará la lista de archivos y directorios.



## 4. Preguntas

1. ¿Qué tipo de archivos se enviaron más rápido?

Se observó que los archivos con extensión .txt y los archivos con extensión .bin se enviaron más rápido.

2. ¿Cuál fue el número máximo de archivos que fue posible enviar a la vez?

Se probó con diferente cantidad de archivos y no se tuvo ningún inconveniente en la recepción y envío de archivos.

3. ¿Cuál fue el tamaño de archivo más grande que se pudo transferir? ¿por qué?

La práctica se probó con archivos de diferente tamaño y todos los archivos se pudieron transferir, sin embargo, esto se debe a la partición o tratamiento que se les hacen a los archivos. Debido a que el archivo que se desea enviar se envía por partes.

4. ¿Qué es el orden de red?

El orden de red es garantizar que los datos que se desean enviar se reciban en la misma secuencia en la que se transmiten

5. ¿Por qué razón es importante utilizar el orden de red al enviar los datos a través de un socket?

Es importante porque en las aplicaciones de red orientadas a conexión se necesita establecer la secuencia en que los datos serán enviados y recibidos, para evitar que los paquetes de datos lleguen antes o después, incluso que no lleguen. TCP dispone de una ventana deslizante en el emisor y en el receptor, de tal forma que, si recibimos un segmento que no está en orden, automáticamente “esperará” hasta que llegue el segmento que falta, o si no, pedirá una retransmisión únicamente del segmento que falte.

6. Si deseamos enviar archivos de tamaño muy grande, ¿qué cambios sería necesario hacer con respecto a los tipos de datos usados para medir el tamaño de los archivos, así como para leer bloques de datos del archivo?

Para ello se necesita llevar a cabo la segmentación del archivo, en nuestro caso el archivo se segmenta en bytes, así se garantiza el envío de archivos de gran tamaño.

## 5. Conclusiones

Con el desarrollo de esta práctica comprendimos cómo se podría implementar una aplicación para el envío de múltiples archivos, incluidos directorios. Diariamente utilizamos servicios de transferencia de archivos e incluso esos archivos son de un gran tamaño, lo que necesitamos es transferir estos archivos de manera confiable, en orden y sin duplicados, para esto se requiere una aplicación que cubra todas las necesidades descritas, para ello se pueden utilizar sockets de flujo. Al utilizar sockets de flujo se garantiza la entrega de datos confiable.

Ahora bien, en cuanto a la arquitectura Cliente-Servidor utilizada aprendimos que el servidor debe exponer un mecanismo que permite a los clientes conectarse, que por lo general es TCP/IP, esta comunicación permitirá una comunicación continua y bidireccional, de tal forma que el cliente puede enviar y recibir datos del servidor y viceversa. En este sentido, las dos partes son mutuamente dependientes, pues una sin la otra no tendría motivo de ser. Finalmente, se comprobó que la arquitectura Cliente-Servidor es considerada una arquitectura distribuida debido a que el servidor y el cliente se encuentran distribuidos en diferentes equipos (aunque podrían estar en la misma máquina) y se comunican únicamente por medio de la red o Internet.

## Bibliografía

- Tanenbaum, Andrew. "Redes de Computadoras". Cuarta Edición, Pearson Prentice Hall, 2003.
- What is a Socket?. Docs.oracle.com. (2019). Retrieved 23 February 2022, from: <https://docs.oracle.com/javase/tutorial/networking/sockets/definition.html>.
- What is a TCP/IP Socket Connection?. Ibm.com. (2022). Retrieved 23 February 2022, from [https://www.ibm.com/docs/en/zvse/6.2?topic=SSB27H\\_6.2.0/fa2ti\\_intro\\_socket\\_program.html](https://www.ibm.com/docs/en/zvse/6.2?topic=SSB27H_6.2.0/fa2ti_intro_socket_program.html).