INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

Práctica 1 - Aplicación estilo DropBox

Unidad de aprendizaje: Aplicaciones para Comunicaciones de Red

Grupo: 3CM8

Alumnos(a):
Nicolás Sayago Abigail
Ramos Díaz Enrique

Profesor(a):
Moreno Cervantes Axel

19 de Febrero 2019

Índice

1. Introducción

En ésta práctica se implementará una aplicación estilo DropBox, que permita al usuario ver los archivos que tiene en su carpeta (en el servidor), así como subir y/o bajar archivos o carpetas a su carpeta del servidor, ya sea mediante una caja de diálogo o usando Drag Drop.

Se utilizarán sockets de flujo bloqueante para comunicar al cliente con el servidor, y el lenguaje de programación Java.

2. Desarrollo

Para la implementación de ésta práctica se utilizaron 3 clases en Java: DropBox, Cliente y Servidor.

2.1. Clase DropBox

Aquí básicamente se construyo la interfaz gráfica por medio de Java Swing que el usuario esta viendo constantemente y que representa su carpeta personal, junto a su contenido, en el servidor.

Ésta clase manda a llamar todos los métodos de la clase Cliente por medio de sus botones, para interactuar con el servidor, como lo son: descargar archivos y carpetas, subir archivos y carpetas, ver y navegar por los archivos de su carpeta, actualizar el cliente, etc.

El listado de archivos y carpetas de la carpeta personal del cliente, almacenada en el servidor, son desplegados por medio de un JList. Si se trata de un directorio, al dar doble clic sobre el mismo se abrirá este, mostrando su contenido.

Para descargar uno o múltiples archivos solo se deben seleccionar éstos en el JList y pulsar el botón ”Descargar”; se guardarán como un archivo ZIP.

```
1  public class DropBox extends JFrame implements ActionListener {
2   JButton BtnSubir, BtnActualizar, BtnDescargar;
3   static JList<String> archivos;
4   static DefaultListModel<String> modelo;
5   MouseListener mouseListener;
6   JPanel panelBotones;
7   static JProgressBar BarraProgreso;
8   JScrollPane scroll;
9
10  public DropBox() {
11   Container c = getContentPane();
12   c.setLayout(new BoxLayout(c, BoxLayout.Y_AXIS));
13
14   archivos = new JList<String>();
15         archivos.setSelectionMode(
16         ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
17
18         /*Añadimos la funcionalidad de doble clic para navegar por directorios*/
19         mouseListener = new MouseAdapter() {
20       public void mouseClicked(MouseEvent e) {
```

```
21            if (e.getClickCount() == 2) {
22                int index = archivos.locationToIndex(e.getPoint());
23                String nombreSeleccion = modelo.getElementAt(index);
24                //Revisamos que la seleccion sea un directorio
25                if (Cliente.tipoFile[index] == 1) {
26                 modelo.clear();
27                 Cliente.AbrirCarpeta(index);}}}};
```

2.2. Clase Cliente

Ésta clase es la encargada de solicitar todas las peticiones al servidor, y pasarle las respuestas a la clase DropBox, que las irá desplegando de forma gráfica al usuario.

Primero se inicializa el puerto del servidor, el host, y un String para la ruta de los directorios por los cuales vamos a navegar como objetos globales y estáticos.

```
1  /*Funciones del cliente que haran las peticiones que se requieran al servidor*/
2  public class Cliente {
3   private static int pto = 4321;
4   private static String host = "127.0.0.1";
5   private static String rutaDirectorios = "";
6   public static String sep = System.getProperty("file.separator");
7   public static int[] tipoFile;
```

Toda la funcionalidad importante de la interfaz de usuario DropBox se encuentra en los métodos de ésta clase (todos estáticos para ser invocados desde la clase DropBox), los cuales son:

2.2.1. void Actualizar()

Al hacer clic en el botón ”Actualizar - Inicio”, después de subir archivos o carpetas, o al abrir una carpeta, el cliente solicita al servidor por medio de un socket y enviando una bandera entera de valor 1, un arreglo de Strings con todo el contenido de su carpeta personal en el servidor, y los agregaba al JList de la clase DropBox para mostrarlos en la interfaz al usuario.

```
1  public static void Actualizar(){
2   try {
3    Socket cl = new Socket(host, pto);
4    DataOutputStream dos = new DataOutputStream(cl.getOutputStream());
       ↪  //OutputStream
5    //La bandera tiene el valor de 1 = Actualizar
6    dos.writeInt(1); dos.flush();
7
8    DataInputStream dis = new DataInputStream(cl.getInputStream()); // InputStream
9    int numArchivos = dis.readInt();
10   tipoFile = new int[numArchivos];
11
12   for(int i = 0; i < numArchivos; i++) {
13    String archivoRecibido = dis.readUTF();
14    DropBox.modelo.addElement(archivoRecibido);
```

```
    tipoFile[i] = dis.readInt();
   }//for

   dis.close(); dos.close(); cl.close();
   System.out.println("Carpeta del cliente actualizada.");
  }catch(Exception e) {
   e.printStackTrace();
  }//catch
}//Actualizar
```

2.2.2. void RecibirArchivos(nombresArchivos[], tamaño)

Después de seleccionar uno o múltiples archivos y/o carpetas en el JList en la interfaz gráfica de la clase DropBox y dar clic en el botón ”Descargar Archivos”, por medio de un socket se envía una bandera entera con valor 2 al servidor, y luego enviamos el número de archivos/directorios a descargar junto a su índice del JList para indicarle al servidor de qué archivos se tratan.

Se reciben un archivo ZIP con los archivos solicitados, leeyéndolo con ayuda del socket y de un flujo de entrada, y escribiéndolo en el disco duro del cliente byte a byte con ayuda de un flujo de salida.

El DropBox indicará la descarga satisfactoria de los archivos comprimidos, que se guardarán en el Escritorio de la computadora del cliente.

```
public static void RecibirArchivos(String[] nombresArchivos, int tama) {
 try {
  Socket cl = new Socket(host, pto);
  DataOutputStream dos = new DataOutputStream(cl.getOutputStream());
    ↪  //OutputStream
  DataInputStream dis = new DataInputStream(cl.getInputStream()); // InputStream
  //La bandera tiene el valor de 2 = Descargar seleccion
  dos.writeInt(2); dos.flush();
  dos.writeInt(tama); dos.flush();
  //Enviamos los indices de los archivos seleccionados
  String aux = "";
  for(int i = 0; i < tama; i++) {
   aux = nombresArchivos[i];
   dos.writeUTF(aux);
   dos.flush();
  }

  String nombre = System.getProperty("user.home");
  if((System.getProperty("os.name")).indexOf("Windows") != -1) {
   //Estamos en Windows
   nombre = nombre + sep + "Desktop" + sep;
  }
  else {
   //Estamos en Ubuntu
   nombre = nombre + sep + "Escritorio" + sep;
  }
  nombre = nombre + dis.readUTF();
  long tam = dis.readLong();
```

```
28    System.out.println("\nSe recibe el archivo " + nombre + " con " + tam +
        ↪  "bytes");
29    DataOutputStream dosArchivo = new DataOutputStream(new
        ↪  FileOutputStream(nombre)); // OutputStream
30
31    long recibidos = 0;
32    int n = 0, porciento = 0;
33    byte[] b = new byte[2000];
34
35    while(recibidos < tam) {
36     n = dis.read(b);
37     dosArchivo.write(b, 0, n);
38     dosArchivo.flush();
39     recibidos += n;
40     porciento = (int)((recibidos * 100) / tam);
41     System.out.println("\r Recibiendo el " + porciento + "% --- " + recibidos +
         ↪  "/" + tam + " bytes");
42    } // while
43
44    JOptionPane.showMessageDialog(null, "Se ha descargado el archivo " + nombre +
        ↪  " con tamanio: " + tam);
45    dos.close(); dis.close();
46    dosArchivo.close(); cl.close();
47   }catch(Exception e) {
48    e.printStackTrace();
49   }//catch
50  }
```

2.2.3. void AbrirCarpeta(indice)

Al hacer doble clic en un directorio dentro del JList que simboliza nuestra carpeta personal en el servidor, por medio de un socket de flujo bloqueante se envía una bandera entera con valor 3 al servidor, y luego el índice que corresponde al directorio seleccionado.

Se lee el numero de archivos contenidos en esa carpeta (apoyándose también del método Actualizar()), y toda su información se va añadiendo al JList de la clase DropBox, para darle a notar al usuario que el cliente abrió la carpeta.

```
1   public static void AbrirCarpeta(int indice){
2    try {
3     Socket cl = new Socket(host, pto);
4     DataOutputStream dos = new DataOutputStream(cl.getOutputStream());
        ↪  //OutputStream
5     //La bandera tiene el valor de 3 = AbrirCarpeta
6     dos.writeInt(3); dos.flush();
7     //Enviamos el indice en donde se encuentra la carpeta dentro del arreglo de
        ↪  Files[]
8     dos.writeInt(indice); dos.flush();
9
10    DataInputStream dis = new DataInputStream(cl.getInputStream()); // InputStream
11    int numArchivos = dis.readInt();
12    tipoFile = new int[numArchivos];
13
```

```
14    for(int i = 0; i < numArchivos; i++) {
15     String archivoRecibido = dis.readUTF();
16     DropBox.modelo.addElement(archivoRecibido);
17     tipoFile[i] = dis.readInt();
18    }//for
19
20    dis.close(); dos.close(); cl.close();
21    System.out.println("Nueva carpeta abierta: Request recibido.");
22   }catch(Exception e) {
23    e.printStackTrace();
24   }//catch
25  }
```

2.2.4. void SeleccionarArchivo()

Al hacer clic en el botón ”Subir Archivos” de la clase DropBox, éste método primero despliega un JFileChooser, para elegir desde nuestro ordenador (cliente) los archivos o directorios que deseemos subir al servidor.

Si se va a subir una carpeta, el cliente envía una bandera entera con valor 4 al servidor. Si se va a subir un archivo, la bandera es 0.

El cliente se encarga de decirle al servidor como irá guardando las carpetas o archivos en la carpeta personal, ubicada en éste último.

Una vez seleccionado uno o múltiples archivos, el cliente los enviará byte por byte con ayuda de un socket y un DataOutputStream.

El cliente DropBox indicará que la subida esta lista, y actualizará la interfaz, mostrando los nuevos archivos desde una petición al servidor.

```
1  public static void SeleccionarArchivos() {
2   try {
3           JFileChooser jf = new JFileChooser();
4           jf.setMultiSelectionEnabled(true);
5           jf.setFileSelectionMode(JFileChooser.FILES_AND_DIRECTORIES);
6           int r = jf.showOpenDialog(null);
7           if(r == JFileChooser.APPROVE_OPTION) {
8            rutaDirectorios = "";
9               File[] files = jf.getSelectedFiles();
10              for(File file : files) {
11               String rutaOrigen = file.getAbsolutePath();
12               // Tipo caso base: La primera vez que mandemos un archivo
13               // Siempre estará en la raíz del servidor
14               EnviarArchivo(file, rutaOrigen, file.getName());
15           }//for
16           DropBox.modelo.clear(); Actualizar();
17          }//if
18      }
19      catch(Exception e) {
20          e.printStackTrace();
21      }
22  }
```

2.3. Clase Servidor

Ésta clase es la encargada de atender las peticiones del cliente, procesarlas y enviarle las respuestas e información.

Primero se inicializa el directorio en donde se guardarán los archivos del cliente, el cual es **./serverP1**.

También inicializaremos un arreglo de tipo File para ir guardando toda la información de los archivos en esta localización y los de sus subdirectorios, un String con la ruta actual de trabajo en el servidor, y un contador para llevar un control con los archivos solicitados para su descarga en ZIPs numerados.

```
1  public class Servidor {
2   public static String sep = System.getProperty("file.separator");
3   private static String rutaServer = "." + sep + "serverP1" + sep;
4   private static File[] list;
5   private static String rutaActual = "";
6   private static int numVeces = 0;
```

2.3.1. Main

Aquí es donde todos las peticiones, métodos y respuestas interactúa.

Primero, creamos un ServerSocket, asignándole el mismo puerto que se asigno en el socket del cliente, y luego creamos un ciclo infinito para estar siempre escuchando a los clientes que se conecten en cualquier momento (creando un socket nuevo de respuesta con el método accept()).

También creamos los flujos de entrada y salida del socket.

```
1  public static void main(String[] args) {
2   try {
3    ServerSocket s = new ServerSocket(4321);
4    s.setReuseAddress(true);
5    System.out.println("Servidor de archivos iniciado, esperando cliente...");
6    // Espera clientes
7    for( ; ; ) {
8     Socket cl = s.accept();
9     System.out.println("\n\nCliente conectado desde " + cl.getInetAddress() + " "
       ↪  + cl.getPort());
10    DataOutputStream dos = new DataOutputStream(cl.getOutputStream());
       ↪  //OutputStream
11    DataInputStream dis = new DataInputStream(cl.getInputStream()); //
       ↪  InputStream
12    int bandera = dis.readInt();
```

Ahora bien, leemos una bandera de tipo entero, que nos ayuda a regular las acciones que tomará el servidor según la acción que se realizo en el cliente DropBox:

- **La bandera vale 0**: El cliente va a subir archivos desde su equipo, por lo que el servidor debe recibirlos y guardarlos en disco duro (en la carpeta antes mencionada).

- **La bandera vale 1**: El usuario actualizó su carpeta en la interfaz gráfica del cliente.
- **La bandera vale 2**: El cliente va a descargar archivos y/o carpetas seleccionadas en la interfaz gráfica, por lo que el servidor debe leerlos del disco, prepararlos, comprimirlos y enviarlos al cliente.

```
else if (bandera == 2) {
 //Descargar archivos -> El servidor prepara y envia archivos
 //Subir archivos -> El servidor recibe
 int tam = dis.readInt();
 String path = "Download" + numVeces + ".zip";
 path = rutaServer + path;
 System.out.println(""+path);
 File archivoZip = new File(path);
 System.out.println(""+archivoZip.getAbsoluteFile());
 crearZIP(dis, tam);
 if(archivoZip.exists()) {
  //System.out.println("Si existeee");
  System.out.println("La path del archivo esta en: " + path + " Con nombre:
   ↪  " + archivoZip.getName());
  EnviarArchivo(dos, archivoZip);
  // Lo elimino porque no debe estar en el servidor, solo lo hice
   ↪  temporalmente
  if(archivoZip.delete())
   System.out.println("Archivo temporal Download" + numVeces + ".zip
    ↪  eliminado");
 }
 numVeces++;
}
```

- **La bandera vale 3**: El usuario dio doble clic en un directorio de su carpeta en la interfaz gráfica del cliente.

```
else if (bandera == 3) {
 //Abrir carpeta -> El servidor envia los nombres de los contenidos de la
  ↪  carpeta seleccionada
 int ubicacionRuta = dis.readInt();
 //Bandera = 1. Se navega dentro de una carpeta
 String nuevaRuta = "" + list[ubicacionRuta].getAbsoluteFile();
 ActualizarCliente(cl, dis, nuevaRuta, 1);
}
```

- **La bandera vale 4**: El cliente va a subir carpetas desde su equipo, por lo que el servidor primero debe leer el nombre de éstas, crearlas dentro de la carpeta del servidor (en disco) y actualizar la ruta actual de trabajo.

```
else if(bandera == 4) {
 //Subir archivos -> El servidor recibe
 String rutaDirectorio = dis.readUTF();
 String path = rutaServer + rutaDirectorio;
 File archivosRuta = new File(path);
 if(!archivosRuta.exists()) {
  archivosRuta.mkdir();
 }
}
```

2.3.2. void RecibirArchivos(dis, nombre)

De forma muy similar a la clase Cliente, el servidor recibe los nombres y tamaños de los archivos enviados por el cliente por medio de un flujo de entrada del socket, los va leeyendo byte a byte y finalmente los escribe en la carpeta ./serverP1 en el disco duro por medio de un flujo de salida. (Para ver cómo se reciben carpetas, vaya a la sección de cuando la bandera vale 4 en el main.)

```
1  public static void RecibirArchivos(DataInputStream dis, String nombre) throws
    ↪  IOException {
2   long tam = dis.readLong();
3   String pathDestino = dis.readUTF();
4   nombre = rutaServer + pathDestino;
5   System.out.println("\nSe recibe el archivo " + nombre + " con " + tam +
     ↪  "bytes");
6   DataOutputStream dos = new DataOutputStream(new FileOutputStream(nombre)); //
     ↪  OutputStream
7   long recibidos = 0;
8   int n = 0, porciento = 0;
9   byte[] b = new byte[2000];
10
11  while(recibidos < tam) {
12   n = dis.read(b);
13   dos.write(b, 0, n); dos.flush();
14   recibidos += n;
15   porciento = (int)((recibidos * 100) / tam);
16   System.out.println("\r Recibiendo el " + porciento + "% --- " + recibidos +
      ↪  "/" + tam + " bytes");
17  } // while
18
19  System.out.println("\nArchivo " + nombre + " de tamanio: " + tam + "
     ↪  recibido.");
20  dos.close(); dis.close();
21 } // RecibirArchivos
```

2.3.3. void ActualizarCliente(cl, dis, rutaServer, bandera)

Aquí lo que hacemos es leer todo lo que contiene la ruta actual de trabajo (de forma predeterminada ./serverP1, pero puede ser una subcarpeta que se desea abrir desde el cliente), añadirlo a un arreglo de string, especificando y diferenciando si se trata de un archivo o un directorio, para su manejo en el cliente, y enviándolo por medio de un socket a éste.

No se envía ningún archivo o carpeta (en bytes), solo su nombre si se trata de un archivo o su ruta relativa si se trata de un directorio. Éste método es meramente informativo para que el cliente sepa que hay en su carpeta actualmente.

```
1  public static void ActualizarCliente(Socket cl, DataInputStream dis, String
    ↪  path, int bandera) throws IOException {
2   File archivosRuta = new File(path);
3   if(!archivosRuta.exists())
4    archivosRuta.mkdir();
5
```

```
 if(bandera == 1) {
  rutaActual = rutaActual + sep + archivosRuta.getName();
  System.out.println("Ubicacion: "+rutaActual);
 }

 list = archivosRuta.listFiles();
 DataOutputStream dos = new DataOutputStream(cl.getOutputStream()); //
   ↪  OutputStream
 dos.writeInt(list.length); dos.flush();
 String info = "";
 int tipo = 0;

    for (File f : list) {
        if (f.isDirectory()) {
            tipo = 1;
            if(bandera == 0) {//Ruta raiz - Inicio
             info = "." + sep + f.getName();
            }
            else {//Abrir ruta y concatenar
             info = "." + rutaActual + sep + f.getName();
            }
        }//if
        else {
         tipo = 2;
         if(bandera == 0){//Ruta raiz - Inicio
          info = f.getName();
         }
         else {//Abrir ruta y concatenar
          info = "." + rutaActual + sep + f.getName();
         }
        }//else
        dos.writeUTF(info); dos.flush();
        dos.writeInt(tipo); dos.flush();
        tipo = 0;
    }//for
    dos.close();
    System.out.println("Informacion enviada al cliente: Carpeta actualizada.");
}//Actualizar
```

2.3.4. void EnviarArchivo(dos, file)

El servidor lee el archivo ZIP (que contiene ya a los archivos y carpetas solicitadas por el cliente), lo carga byte a byte desde el disco con ayuda de un flujo de entrada, y lo envía al cliente con ayuda de un socket y un flujo de salida.

```
public static void EnviarArchivo(DataOutputStream dos, File f) {
 try {
  String nombre = f.getName();
  long tam = f.length();
  String path = f.getAbsolutePath();
  System.out.println("\nSe envia el archivo " + nombre + " con " + tam + "
   ↪  bytes");
```

```
7     DataInputStream disArchivo = new DataInputStream(new FileInputStream(path));
        ↪  // InputStream
8     //Se envia info de los archivos
9     dos.writeUTF(nombre); dos.flush();
10    dos.writeLong(tam); dos.flush();
11
12    long enviados = 0;
13    int n = 0, porciento = 0;
14    byte[] b = new byte[2000];
15
16    while(enviados < tam) {
17        n = disArchivo.read(b);
18        dos.write(b, 0, n); dos.flush();
19        enviados += n;
20        porciento = (int)((enviados * 100) / tam);
21        System.out.println("\r Enviando el " + porciento + "% --- " + enviados +
            ↪  "/" + tam + " bytes");
22    } //while
23    System.out.println("\nArchivo " + nombre + " de tamanio: " + tam + "
        ↪  enviado.");
24    disArchivo.close(); dos.close();
25   } // try
26   catch(Exception e) {
27    e.printStackTrace();
28   }
29  } // Enviar archivo
```

3. Pruebas

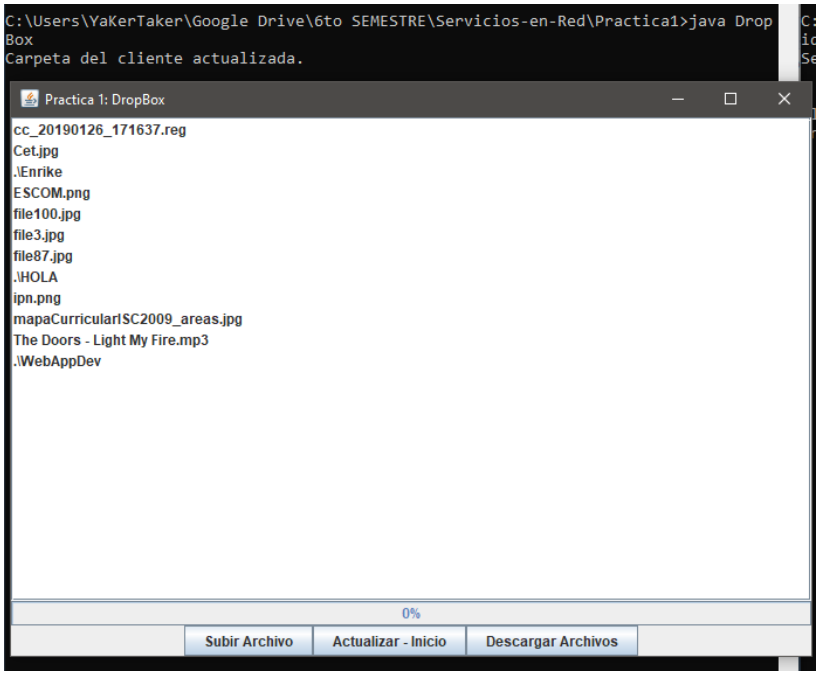

Figura 1: Cliente DropBox

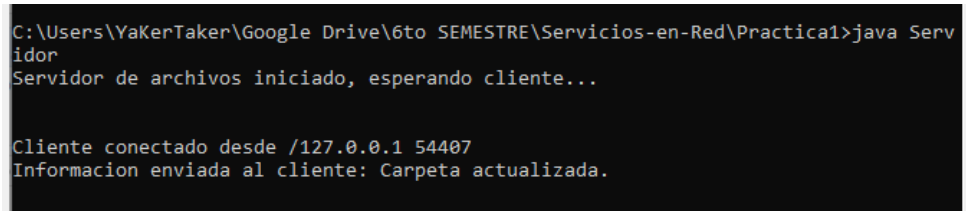

Figura 2: Servidor

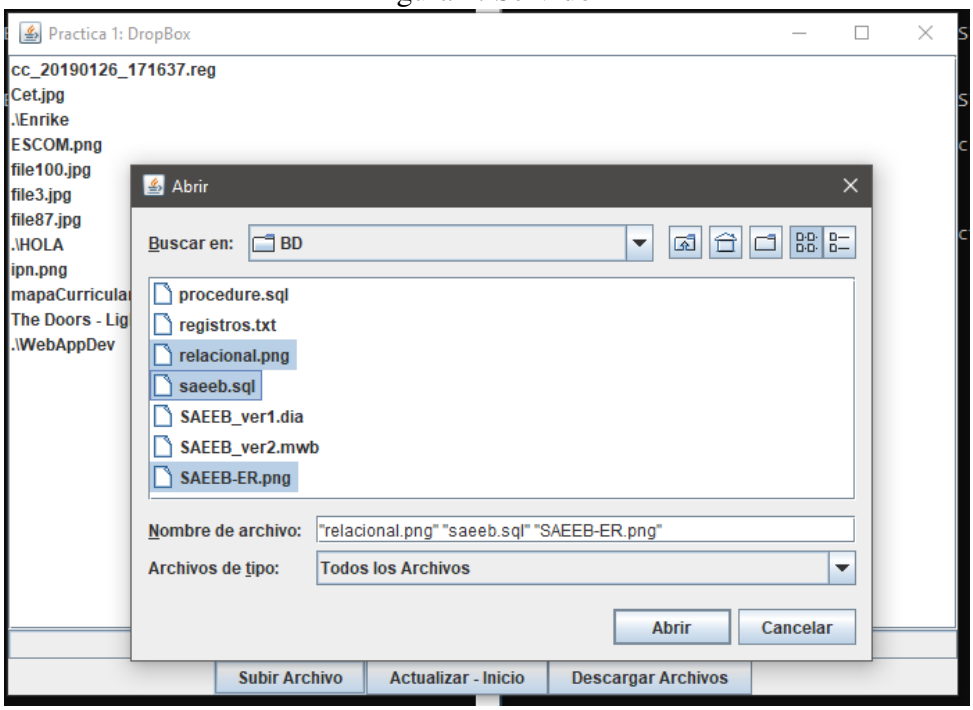

Figura 3: Seleccionar archivos para subir

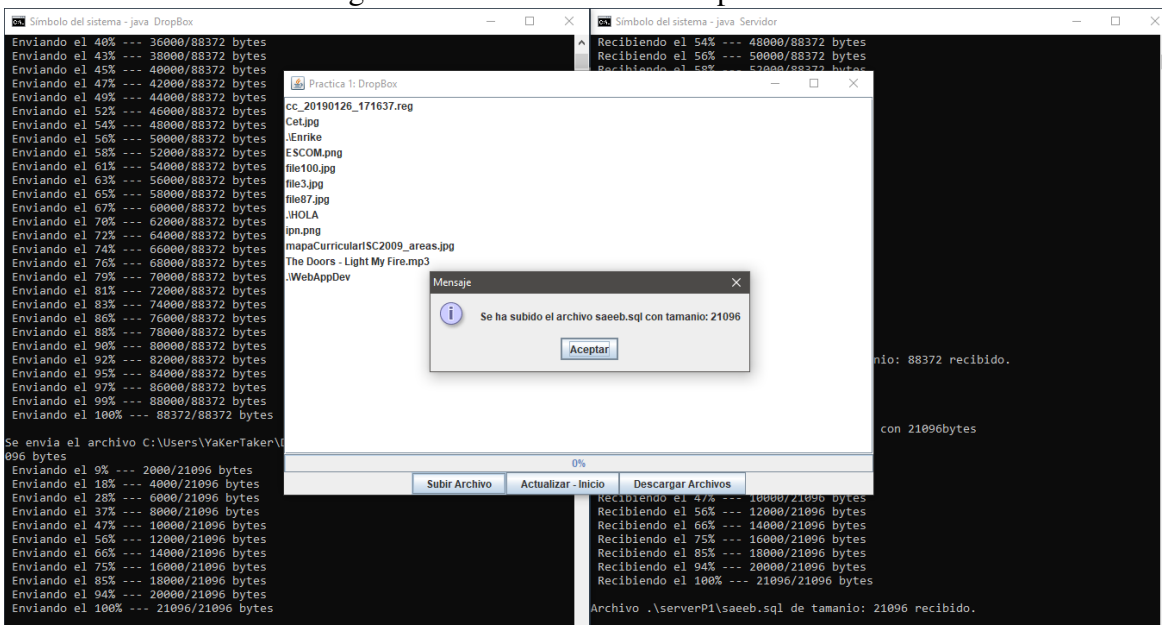

Figura 4: Archivos subidos al servidor

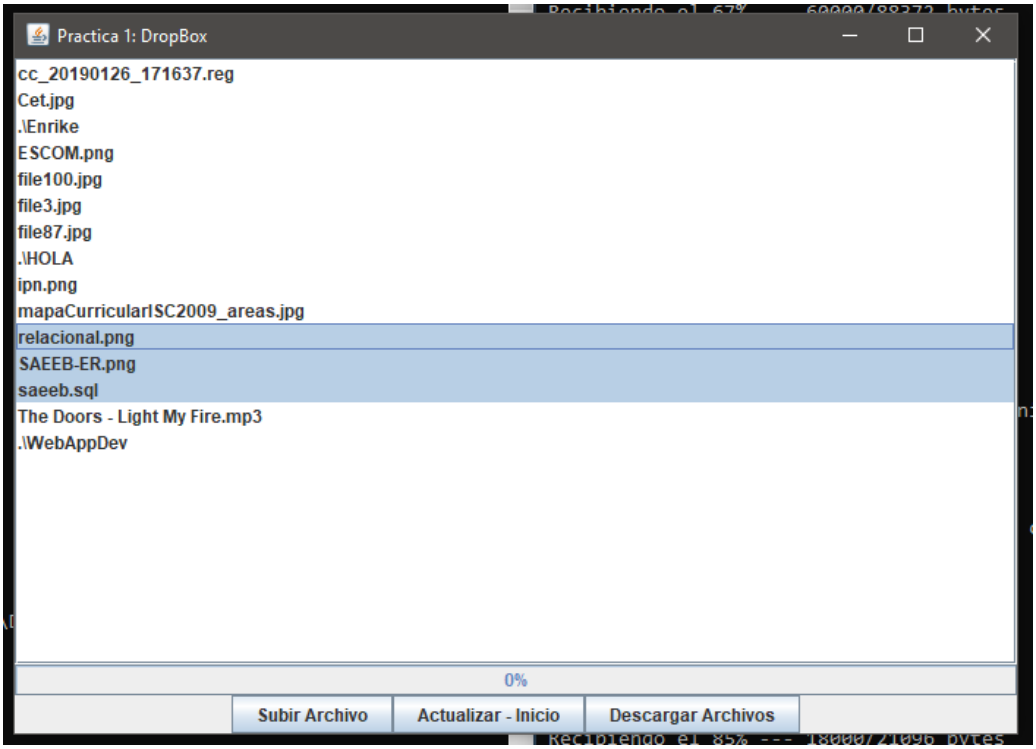

Figura 5: Carpeta del cliente actualizada

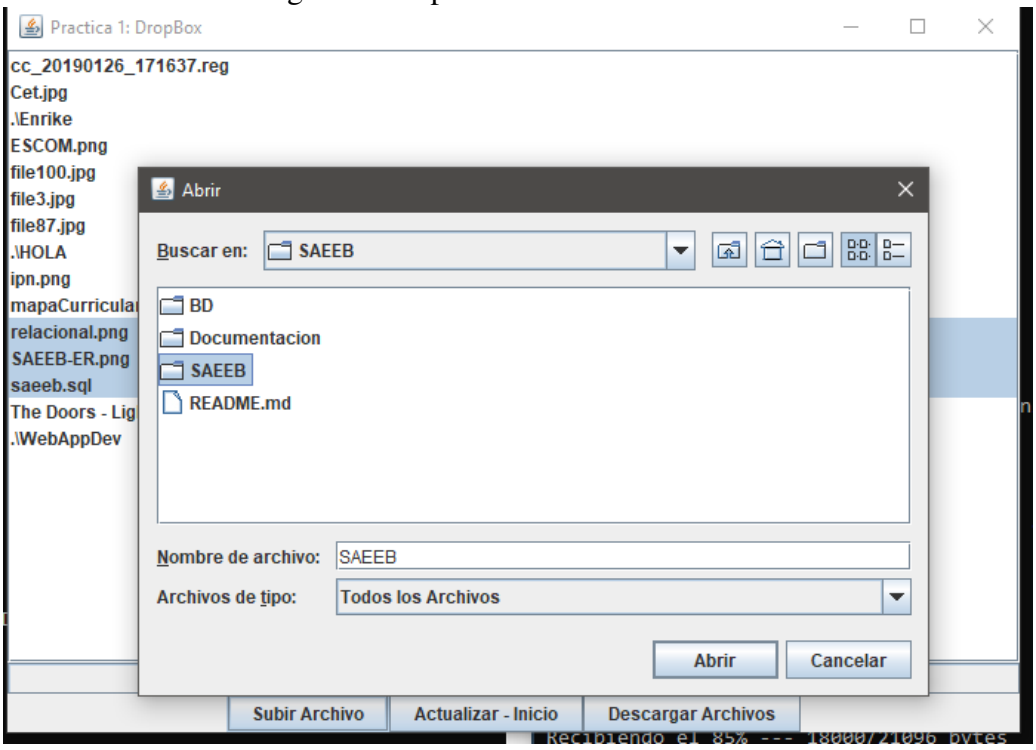

Figura 6: Subir carpetas

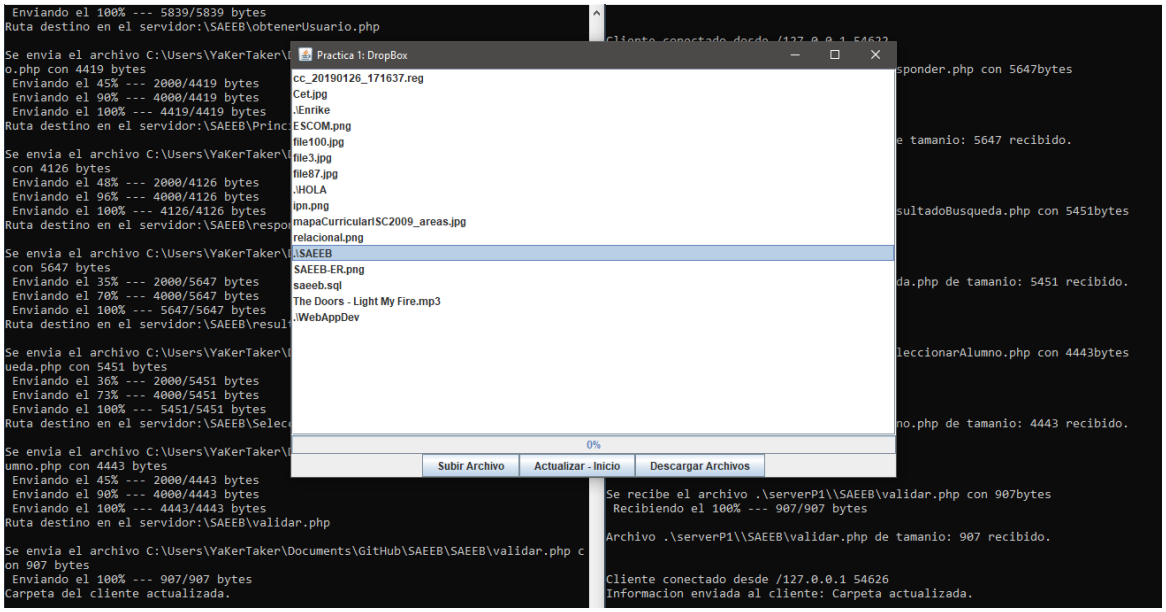

Figura 7: Carpeta subida al servidor

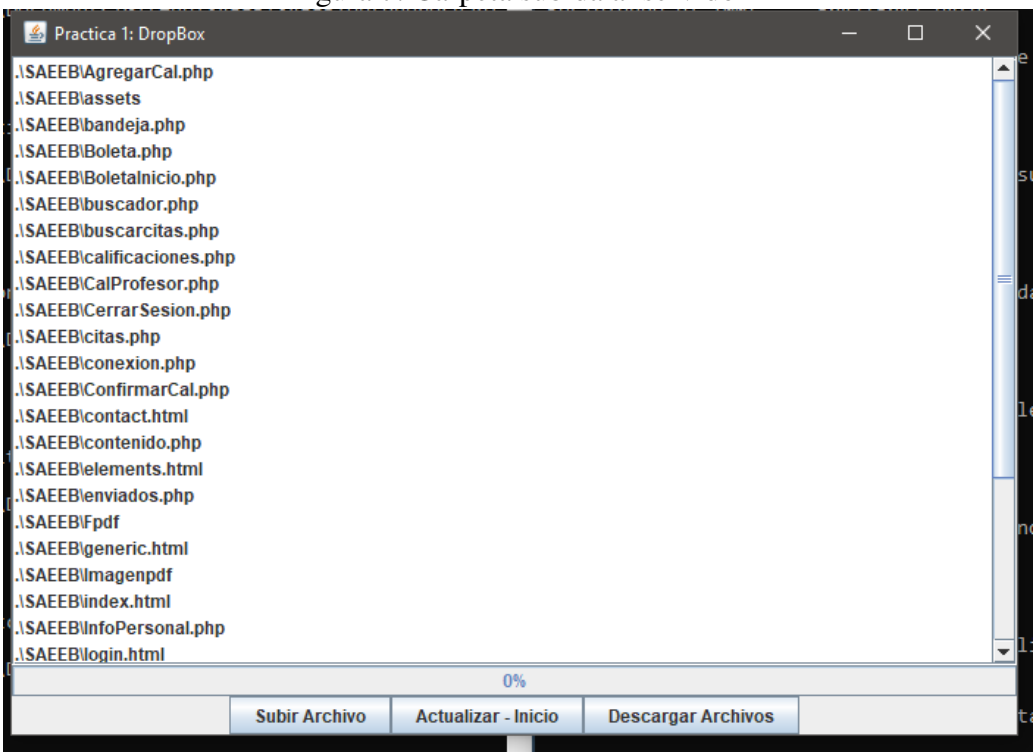

Figura 8: Abrir y navegar por carpetas desde el cliente

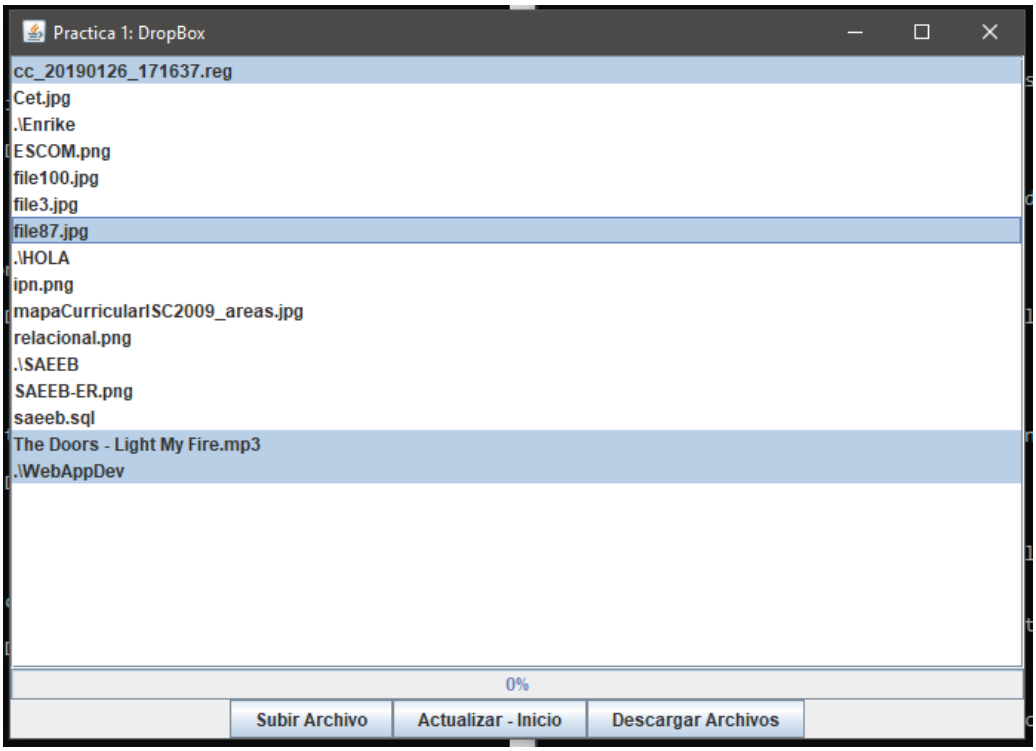

Figura 9: Seleccionar archivos del servidor para descargar

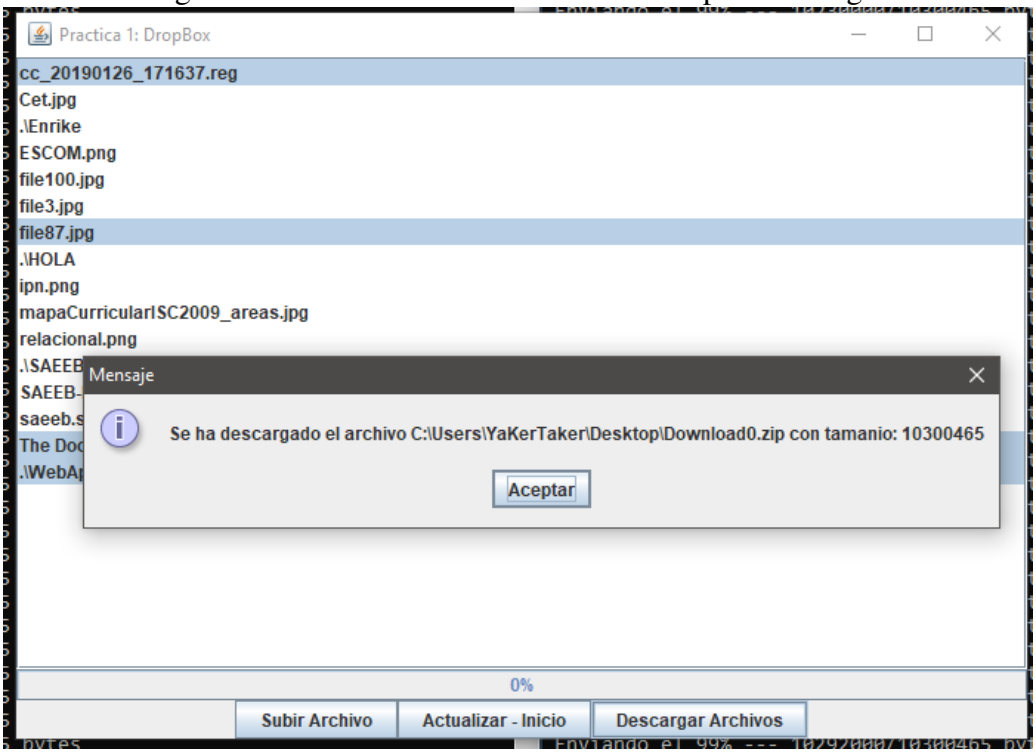

Figura 10: Archivos comprimidos en un ZIP descargados

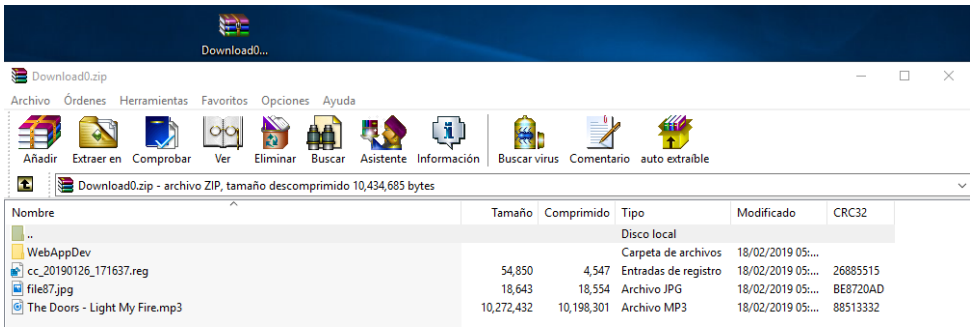

Figura 11: Archivo ZIP

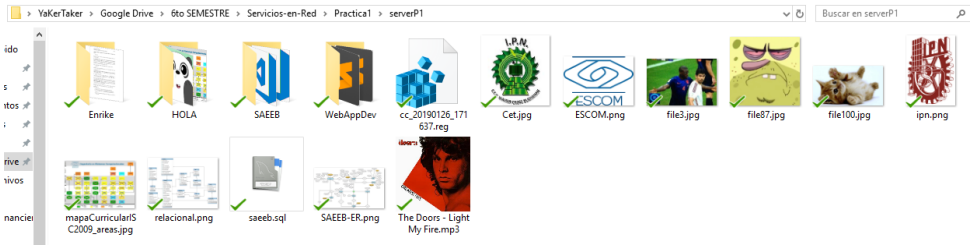

Figura 12: Carpeta personal del cliente en el servidor

```
 Enviando el 99% --- 10236000/10300465 bytes
 Enviando el 99% --- 10238000/10300465 bytes
 Enviando el 99% --- 10240000/10300465 bytes
 Enviando el 99% --- 10242000/10300465 bytes
 Enviando el 99% --- 10244000/10300465 bytes
 Enviando el 99% --- 10246000/10300465 bytes
 Enviando el 99% --- 10248000/10300465 bytes
 Enviando el 99% --- 10250000/10300465 bytes
 Enviando el 99% --- 10252000/10300465 bytes
 Enviando el 99% --- 10254000/10300465 bytes
 Enviando el 99% --- 10256000/10300465 bytes
 Enviando el 99% --- 10258000/10300465 bytes
 Enviando el 99% --- 10260000/10300465 bytes
 Enviando el 99% --- 10262000/10300465 bytes
 Enviando el 99% --- 10264000/10300465 bytes
 Enviando el 99% --- 10266000/10300465 bytes
 Enviando el 99% --- 10268000/10300465 bytes
 Enviando el 99% --- 10270000/10300465 bytes
 Enviando el 99% --- 10272000/10300465 bytes
 Enviando el 99% --- 10274000/10300465 bytes
 Enviando el 99% --- 10276000/10300465 bytes
 Enviando el 99% --- 10278000/10300465 bytes
 Enviando el 99% --- 10280000/10300465 bytes
 Enviando el 99% --- 10282000/10300465 bytes
 Enviando el 99% --- 10284000/10300465 bytes
 Enviando el 99% --- 10286000/10300465 bytes
 Enviando el 99% --- 10288000/10300465 bytes
 Enviando el 99% --- 10290000/10300465 bytes
 Enviando el 99% --- 10292000/10300465 bytes
 Enviando el 99% --- 10294000/10300465 bytes
 Enviando el 99% --- 10296000/10300465 bytes
 Enviando el 99% --- 10298000/10300465 bytes
 Enviando el 99% --- 10300000/10300465 bytes
 Enviando el 100% --- 10300465/10300465 bytes

Archivo Download0.zip de tamanio: 10300465 enviado.
Archivo temporal Download0.zip eliminado


Cliente conectado desde /127.0.0.1 54643
Informacion enviada al cliente: Carpeta actualizada.
```

Figura 13: Enviando ZIP desde el servidor

```
 Recibiendo el 99% --- 10232000/10300465 bytes
 Recibiendo el 99% --- 10234000/10300465 bytes
 Recibiendo el 99% --- 10236000/10300465 bytes
 Recibiendo el 99% --- 10238000/10300465 bytes
 Recibiendo el 99% --- 10240000/10300465 bytes
 Recibiendo el 99% --- 10242000/10300465 bytes
 Recibiendo el 99% --- 10244000/10300465 bytes
 Recibiendo el 99% --- 10246000/10300465 bytes
 Recibiendo el 99% --- 10248000/10300465 bytes
 Recibiendo el 99% --- 10250000/10300465 bytes
 Recibiendo el 99% --- 10252000/10300465 bytes
 Recibiendo el 99% --- 10254000/10300465 bytes
 Recibiendo el 99% --- 10256000/10300465 bytes
 Recibiendo el 99% --- 10258000/10300465 bytes
 Recibiendo el 99% --- 10260000/10300465 bytes
 Recibiendo el 99% --- 10262000/10300465 bytes
 Recibiendo el 99% --- 10264000/10300465 bytes
 Recibiendo el 99% --- 10266000/10300465 bytes
 Recibiendo el 99% --- 10268000/10300465 bytes
 Recibiendo el 99% --- 10270000/10300465 bytes
 Recibiendo el 99% --- 10272000/10300465 bytes
 Recibiendo el 99% --- 10274000/10300465 bytes
 Recibiendo el 99% --- 10276000/10300465 bytes
 Recibiendo el 99% --- 10278000/10300465 bytes
 Recibiendo el 99% --- 10280000/10300465 bytes
 Recibiendo el 99% --- 10282000/10300465 bytes
 Recibiendo el 99% --- 10284000/10300465 bytes
 Recibiendo el 99% --- 10286000/10300465 bytes
 Recibiendo el 99% --- 10288000/10300465 bytes
 Recibiendo el 99% --- 10290000/10300465 bytes
 Recibiendo el 99% --- 10292000/10300465 bytes
 Recibiendo el 99% --- 10294000/10300465 bytes
 Recibiendo el 99% --- 10296000/10300465 bytes
 Recibiendo el 99% --- 10298000/10300465 bytes
 Recibiendo el 99% --- 10300000/10300465 bytes
 Recibiendo el 100% --- 10300465/10300465 bytes
Carpeta del cliente actualizada.
Nueva carpeta abierta: Request recibido.
Nueva carpeta abierta: Request recibido.
Nueva carpeta abierta: Request recibido.
Carpeta del cliente actualizada.
```

Figura 14: Cliente recibiendo el ZIP

4. Posibles mejoras

- Poner los archivos a descargar/subir en una cola de espera, para así evitar problemas de lectura o transferencia de datos y asignar tiempos.
- Mejorar la navegación entre carpetas, pues únicamente podemos ir entrando en directorios, pero no podemos regresar a aquellos que ya visitamos (solo a la raíz).
- Agregar un elemento visual, como una barra de progreso, que indique el estado de las cargas o descargas de archivos.
- Descomprimir el ZIP una vez recibido por el cliente, así como evitar comprimir cuando solo se trata de un único archivo.
- Aumentar el alcance para permitir muchos clientes, cada uno con su respectiva carpeta, conectados al mismo tiempo al servidor.
- Manejar cuentas de usuario, y compartir archivos entre estos.
- Historial de versiones.
- Añadir la funcionalidad Drag and Drop para arrastrar archivos al cliente y subirlos automáticamente, o tomarlos del cliente y arrastrarlos a algún directorio de nuestro equipo.
- Mejorar la presentación de los archivos en el cliente, añadiendo más detalles como tamaño, fecha de creación, etc.
- Abrir una vista previa del archivo cuando es seleccionado.

5. Conclusiones

5.1. Ramos Diaz Enrique

El trabajo de comunicación y transmisión de archivos no fue problema, pues la clases Socket y ServerSockets (sockets de flujo bloqueante) en Java hacen todo éste proceso con métodos muy simples de utilizar y comprender.

El verdadero reto fue el manejo de directorios y subdirectorios, tanto para subir como para descargar archivos, ya debiamos tener un control del directorio de trabajo actual, y de distinguins entre archivos y carpetas cuando el usuario las seleccionaba en el cliente. Hablando del cliente, también hubo algo de complejidad en emular un explorador de archivos como el de Windows o Google Drive al abrir carpetas, ver sus contenidos y navegar por ellas (siendo la única excepción regresar a la ruta anterior).

Nos apoyamos de métodos como recursividad, condicionales, y otros métodos de clases como el de compresión de archivos a descargar en un ZIP, eligiendo esta forma para manejar los archivos y directorios que se descargan desde el servidor.