

CAPÍTULO 4

GENERACIÓN DE SERVICIOS EN RED

Contenidos

Protocolos estándar de comunicaciones en red.

Librerías para comunicar con los protocolos estándar.

Crear clientes de protocolos estándar de comunicaciones en red.

Programación de servidores y clientes.

Objetivos

Utilizar librerías Java para implementar protocolos estándar de comunicación en red.

Programar clientes de protocolos estándar de comunicación en red.

Probar servicios de comunicación en red.

Crear servicios capaces de gestionar varios clientes.

RESUMEN DEL CAPÍTULO

En este capítulo utilizaremos **Apache Commons Net™**, que es una librería Java, para implementar protocolos estándar de comunicaciones y aprenderemos a crear clientes para acceder a servidores FTP, SMTP o POP3.

4.1. INTRODUCCIÓN

Los **servicios** son programas auxiliares utilizados en un sistema de computadoras para gestionar una colección de recursos y prestar su funcionalidad a los usuarios y aplicaciones. Por ejemplo, cuando enviamos un documento a una impresora que está formando parte de una red estamos usando un servicio de impresión, este servicio permite gestionar y compartir la impresora en la red. El único acceso que tenemos al servicio está formado por el conjunto de operaciones que ofrece, por ejemplo, un servicio de ficheros ofrece operaciones de lectura, escritura o borrado de ficheros.

Todos los servicios de Internet implementan una relación cliente-servidor, en este capítulo estudiaremos estos servicios y usaremos Java para programar clientes de los servicios de Internet que se usan más frecuentemente.

4.2. PROTOCOLOS ESTÁNDAR DE COMUNICACIÓN EN RED

El modelo TCP/IP esta compuesto por cuatro capas o niveles (véase Figura 4.1). La capa de aplicación define las aplicaciones de red y los servicios de Internet estándar que puede utilizar un usuario. Estos servicios utilizan la capa de transporte para enviar y recibir datos. Existen varios protocolos de capa de aplicación.

En la lista siguiente se incluyen ejemplos de protocolos de capa de aplicación:

- Conexión remota: Telnet.
- Correo electrónico: SMTP.
- Acceso a ficheros remotos: FTP, NFS, TFTP
- Resolución de nombres de ordenadores: DNS, WINS.
- World Wide Web: HTTP.

Capas TCP/IP	Capas y protocolos TCP/IP	
Aplicación	SMTP, Telnet, FTP, HTTP	NFS, SNMP, DNS
Transporte	TCP	UDP
Internet	IP	
Interfaz de Red	Protocolos de subred	

Figura 4.1. Protocolos y aplicaciones TCP/IP.

Todas las aplicaciones que implementan TCP/IP se basan en el modelo cliente-servidor.

TELNET (*Telecommunication Network*): Emulación de terminal; permite a un usuario acceder a una máquina remota y manejarla como si estuviese sentado delante de ella. Es el sistema empleado para arreglar fallos de máquinas remotas o para realizar consultas a distancia como por ejemplo para consultar los fondos de una biblioteca. Su principal problema es la seguridad ya que los nombres de usuario y contraseñas viajan por la red como texto plano.

SMTP (*Simple Mail Transfer Protocol*): Protocolo simple de transferencia de correo electrónico; es probablemente el servicio más popular entre los usuarios de la red. Este estándar especifica el formato exacto de los mensajes que un cliente en una máquina debe enviar al servidor en otra. Administra la transmisión de correo electrónico a través de las redes informáticas.

FTP (*File Transfer Protocol*): Protocolo de transferencia de ficheros; es un servicio confiable orientado a conexión que se utiliza para transferir ficheros de una máquina a otra a través de Internet. Los sitios FTP son lugares desde los que podemos descargar o enviar ficheros.

TFTP (*Trivial File Transfer Protocol*): Protocolo trivial de transferencia de ficheros; es un protocolo de transferencia muy simple semejante a una versión básica de FTP. Fue definido para aplicaciones que no necesitan tanta interacción entre cliente y servidor. A menudo se utiliza para transferir ficheros entre ordenadores en una red en los que no es necesaria una autenticación. Es un servicio no orientado a conexión que utiliza el protocolo UDP.

HTTP (*HyperText Transference Protocol*): Protocolo de Transferencia de Hipertexto; utilizado por los navegadores web para realizar peticiones a los servidores web y para recibir las respuestas de ellos. Es un protocolo que especifica los mensajes involucrados en un intercambio petición-respuesta, los métodos, argumentos y resultados y las reglas para representar todo ello en los mensajes.

NFS (*Network File System*): Sistema de ficheros de red, ha sido desarrollado por *Sun Microsystems* y permite a los usuarios el acceso en línea a ficheros que se encuentran en sistemas remotos, de esta forma el usuario accede a un fichero como si este fuera un fichero local.

SNMP (*Simple Network Management Protocol*): Protocolo simple de administración de red, es un protocolo utilizado para intercambiar información de gestión entre los dispositivos de una red. Permite a los administradores monitorear, controlar y supervisar el funcionamiento de la red.

DNS (*Domain Name System*): Sistema de nombres de dominio, es un sistema que usa servidores distribuidos a lo largo de la red para resolver el nombre de un host IP (nombre de ordenador + nombre de subdominio + nombre de dominio) en una dirección IP, de esta manera no hay que recordar y usar su dirección IP.

En este capítulo aprenderemos a crear clientes para acceder a diferentes servicios TCP/IP.

4.3. COMUNICACIÓN CON UN SERVIDOR FTP

FTP es una de las herramientas más útiles para el intercambio de ficheros entre diferentes ordenadores y es la forma habitual de publicación en Internet.

Para usar FTP para transferir ficheros entre dos ordenadores, cada uno debe tener un papel, es decir, uno debe ser el cliente FTP y el otro el servidor FTP. El cliente envía comandos al servidor (subir, bajar o borrar ficheros, crear un directorio) y el servidor los lleva a cabo. Podemos imaginarnos al servidor como un gran contenedor en el que podemos encontrar gran cantidad de ficheros y directorios.

Hay dos tipos fundamentales de acceso a través de FTP:

- **Acceso anónimo:** cuando la conexión con la máquina servidora la realiza un usuario sin autenticar y sin ningún tipo de privilegio en el servidor. En este caso al usuario es recluido a un directorio público donde sólo se le permite descargar ficheros.
- **Acceso autorizado:** el usuario que realiza la conexión con la máquina servidora está registrado y tiene ciertos privilegios en el servidor. En este caso, y una vez autenticado, el usuario es recluido a su directorio personal donde puede subir y bajar ficheros; normalmente se le asigna una cuota de espacio.

FTP utiliza dos conexiones TCP distintas, una conexión de control y otra de transferencia de datos. La primera se encarga de iniciar y mantener la comunicación entre el cliente y el servidor, la segunda se encarga de enviar datos entre cliente y servidor, esta existe únicamente cuando hay datos a transmitir.

Normalmente, cuando un cliente se conecta a un servidor FTP, el cliente emplea un puerto aleatorio pero el servidor se conecta en el **puerto 21**. Este puerto, **command port** o **control port**, es el puerto de comandos o canal de control por el que se transfieren las órdenes. Para la transferencia de datos no se utilizan los mismos puertos, el cliente obtiene un nuevo puerto y el servidor en el **proceso de transferencia de datos** usa normalmente el **puerto 20**.

En este proceso de comunicación entre cliente y servidor, el cliente puede actuar en modo activo o en modo pasivo:

- **Modo activo:** también llamado estándar o PORT. En modo activo, el servidor siempre crea el canal de datos en su puerto 20, mientras que en el lado del cliente el canal de datos se asocia a un puerto aleatorio mayor que el 1024, por ejemplo, el 1027. Véase Figura 4.2:
1. En el paso 1, el cliente manda un comando PORT al puerto de comandos del servidor (el 21) y le indica el número de puerto para el canal de datos (el 1027).
 2. En el paso 2, el servidor envía un ACK (acuse de recibo) de vuelta al puerto de comandos del cliente (el 1026).
 3. En el paso 3, el servidor inicia la conexión del puerto local de datos (el 20) con el puerto de datos especificado anteriormente, el 1027.
 4. Por último, el cliente envía un ACK de vuelta al servidor.

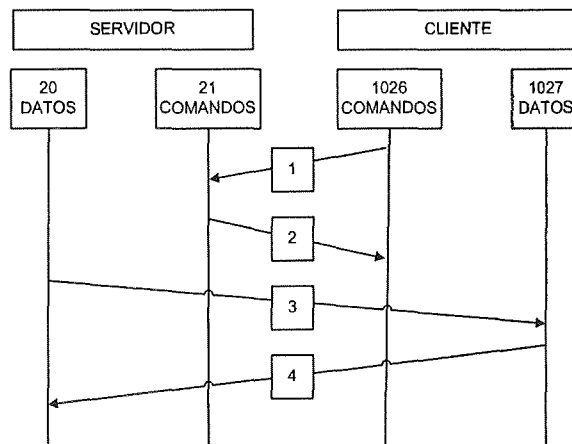


Figura 4.2. Modo activo

- **Modo pasivo:** o PASV, en este caso el cliente envía comandos tipo PASV, entonces el servidor FTP le indicará por el canal de control el puerto al que debe conectarse el cliente, debe ser mayor que 1024. Véase Figura 4.3:
1. En el paso 1, el cliente manda un comando PASV al puerto de comandos del servidor (el 21).
 2. En el paso 2, el servidor responde indicando al cliente el puerto por el que escuchará la conexión de datos, en este caso el puerto 2024.
 3. En el paso 3, el cliente inicia la conexión de datos desde el puerto de datos del cliente, al puerto de datos del servidor especificado anteriormente (el 2024).
 4. Por último, el servidor envía un ACK al puerto de datos del cliente.

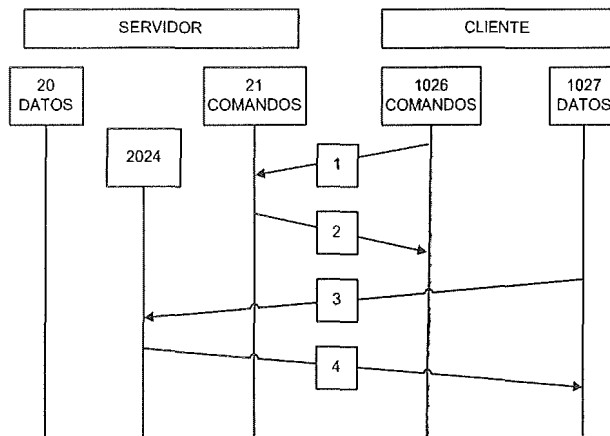


Figura 4.3. Modo pasivo

Tanto en el modo activo como en el modo pasivo el servidor utiliza el puerto 21 como puerto de comandos o canal de control. El modo activo tiene un problema de seguridad y es que se abre una conexión para datos a la máquina cliente desde fuera para adentro, y si el cliente está conectado a una red insegura como Internet puede ser atacado fácilmente. Normalmente, el uso de cortafuegos instalados en el equipo rechazará dichas conexiones. Para evitar estos problemas se desarrolló el modo pasivo.

4.3.1. JAVA PARA COMUNICAR CON UN SERVIDOR FTP

Existen librerías Java que nos permiten crear programas cliente para comunicar con un servidor FTP. **Apache Commons Net™** proporciona una librería de componentes que nos permite implementar el lado cliente de muchos protocolos básicos de Internet. La librería incluye soporte para protocolos como FTP, SMTP, Telnet, TFTP, etc.

A continuación, vamos a ver como acceder desde un programa cliente Java, a un servidor FTP, podremos conectarnos, listar los ficheros y directorios, subir ficheros, eliminarlos, etc. Necesitaremos la librería **commons-net-x.y.jar** que se puede descargar desde la URL de Apache Commons http://commons.apache.org/proper/commons-net/download_net.cgi; para los ejemplos se ha utilizado la versión 3.6.

La clase **FTPClient** encapsula toda la funcionalidad necesaria para almacenar y recuperar ficheros de un servidor FTP. Esta clase se encarga de todos los detalles de bajo nivel de la interacción con un servidor FTP. Para utilizarla primero es necesario realizar la conexión al servidor con el método **connect()**, comprobar el código de respuesta para ver si ha ocurrido algún error, realizar las operaciones de transferencia y cuando finalice el proceso, cerrar la conexión usando el método **disconnect()**. En el siguiente ejemplo realizamos una conexión a un servidor FTP (*ftp.rediris.es*), comprobamos si se ha realizado correctamente o no y cerramos la conexión:

```

import java.io.IOException;
import java.net.SocketException;

import org.apache.commons.net.ftp.*;

public class ClienteFTP1 {
    public static void main(String[] args) throws SocketException,
                                                IOException {

        FTPClient cliente = new FTPClient();
        String servFTP = "ftp.rediris.es"; // servidor FTP
        System.out.println("Nos conectamos a: " + servFTP);
        cliente.connect(servFTP);
    }
}
  
```

```

// respuesta del servidor FTP
System.out.print(cliente.getReplyString());

// código de respuesta
int respuesta = cliente.getReplyCode();
System.out.println("Respuesta: "+respuesta);

// comprobación del código de respuesta
if (!FTPReply.isPositiveCompletion(respuesta)) {
    cliente.disconnect();
    System.out.println("Conexión rechazada: " + respuesta);
    System.exit(0);
}

// desconexión del servidor FTP
cliente.disconnect();
System.out.println("Conexión finalizada.");
}
} // ..

```

La ejecución visualiza la siguiente información:

```

Nos conectamos a: ftp.rediris.es
220- Bienvenido al servicio de replicas de RedIRIS.
220- Welcome to the RedIRIS mirror service.
220 Only anonymous FTP is allowed here
Respuesta: 220
Conexión finalizada.

```

La clase **FTPReply** almacena un conjunto de constantes para códigos de respuesta FTP. Para interpretar el significado de los códigos se puede consultar la RFC 959 (<http://www.ietf.org/rfc/rfc959.txt>). Los nombres nemónicos usados para las constantes son transcripciones de las descripciones de los códigos de la RFC 959. El método **isPositiveCompletion(int respuesta)** devuelve *true* si un código de respuesta ha terminado positivamente. El código 220 significa que el servicio está preparado.

El protocolo FTP usa un esquema de códigos de respuesta donde cada uno de sus dígitos tiene un significado especial. Son números de tres dígitos en ASCII, el primer dígito indica si la respuesta es buena, mala o incompleta:

- **1yz:** Respuesta preliminar positiva, el servidor inició la acción solicitada.
- **2yz:** Respuesta de terminación positiva, el servidor terminó con éxito la acción solicitada.
- **3yz:** Respuesta intermedia positiva, el servidor aceptó el comando, pero la acción solicitada necesita más información.
- **4yz:** Respuesta de terminación negativa transitoria, el servidor no aceptó el comando, y la acción solicitada no ocurrió.
- **5yz:** Respuesta de terminación negativa permanente, el servidor no aceptó el comando y la acción solicitada no ocurrió.

En la siguiente tabla se muestran algunos los métodos de la clase **FTP** usados anteriormente:

MÉTODOS	MISIÓN
void connect(String host)	Abre la conexión con el servidor FTP indicado en <i>host</i> .
int getReplyCode()	Devuelve el valor entero del código de respuesta de la última respuesta FTP.
String getReplyString()	Devuelve el texto completo de la respuesta del servidor FTP.

En la siguiente tabla se muestran algunos métodos de la clase **FTPClient** (derivada de **FTP**) que utilizaremos para logearnos al servidor FTP, subir, bajar y eliminar ficheros, movernos de un directorio a otro, etc. Muchos de estos métodos devuelven un valor booleano, *true* si el método tuvo éxito y *false* en caso contrario:

MÉTODOS	MISIÓN
void disconnect()	Cierra la conexión con el servidor FTP y restaura los parámetros de conexión a los valores predeterminados.
boolean login (String user, String passwd)	Inicia sesión en el servidor FTP usando el nombre de usuario y la contraseña proporcionados. Devuelve <i>true</i> si se inicia la sesión con éxito, si no devuelve <i>false</i>
boolean logout ()	Sale del servidor FTP.
void enterLocalActiveMode()	Se establece el modo de conexión de datos actual en modo activo.
void enterLocalPassiveMode()	Se establece el modo de conexión de datos actual en modo pasivo.
String printWorkingDirectory ()	Devuelve el nombre de ruta del directorio de trabajo actual.
FTPFile [] listFiles()	Obtiene una lista de ficheros del directorio actual como un array de objetos FTPFile .
FTPFile [] listFiles(String path)	Obtiene una lista de ficheros del directorio indicado en <i>path</i> .
String[] listNames()	Obtiene una lista de ficheros del directorio actual como un array de cadenas.
FTPFile[] listDirectories ()	Obtiene la lista de los directorios que se encuentran en el directorio de trabajo actual.
FTPFile[] listDirectories(String parent)	Obtiene la lista de los directorios que se encuentran en el directorio de especificado en <i>parent</i> .
boolean changeWorkingDirectory (String pathname)	Cambia el directorio de trabajo actual de la sesión FTP al indicado en <i>pathname</i> .
boolean changeToParentDirectory ()	Cambia al directorio padre del directorio de trabajo actual.
boolean setFileType (int fileType)	Establece el tipo de fichero a transferir: ASCII_FILE_TYPE (fichero ASCII), BINARY_FILE_TYPE (imagen binaria), etc.
boolean storeFile (String nombre, InputStream local)	Almacena un fichero en el servidor con el nombre indicado tomando como entrada el InputStream , si el fichero existe lo sobrescribe.
boolean retrieveFile (String nombre, OutputStream local)	Recupera un fichero del servidor y lo escribe en el OutputStream dado.
boolean deleteFile (String pathname)	Elimina un fichero en el servidor FTP.

MÉTODOS	MISIÓN
boolean rename (String antiguo, String nuevo)	Cambia el nombre de un fichero del servidor FTP.
boolean removeDirectory (String pathname)	Elimina un directorio en el servidor FTP (si está vacío).
boolean makeDirectory (String pathname)	Crea un nuevo subdirectorio en el servidor FTP en el directorio actual.

Todos los métodos de comunicación con el servidor pueden lanzar *IOException*. El servidor FTP puede optar por cerrar antes de tiempo una conexión si el cliente ha estado inactivo durante más de un período de tiempo determinado (generalmente 900 segundos). La clase **FTPClient** detectará un cierre prematuro de la conexión con el servidor FTP y se puede producir la excepción *FTPConnectionClosedException*.

Lo más normal es conectar a un servidor FTP con un nombre de usuario y su clave. Para identificarnos usaremos el método **login()** que devuelve *true* si la conexión se realiza correctamente. Nos conectamos al servidor en modo pasivo usando el método **enterLocalPassiveMode()**. Para desconectarnos usamos el método **logout()**.

En el siguiente ejemplo nos conectamos al servidor FTP *ftp.rediris.es* utilizando un acceso anónimo. Nos conectamos como usuario *anonymous* y clave igual (aunque nos vale cualquier nombre) para mostrar la lista de ficheros del directorio actual. Usamos el método **listFiles()** que devuelve un array de la clase **FTPFile** con información de los ficheros y directorios encontrados; recorreremos el array visualizando el nombre del fichero o directorio y el tipo que puede ser fichero, directorio o enlace simbólico:

```
import java.io.*;
import org.apache.commons.net.ftp.*;

public class ClienteFTP2 {
    public static void main(String[] args) {
        FTPClient cliente = new FTPClient();
        String servFTP = "ftp.rediris.es";
        System.out.println("Nos conectamos a: " + servFTP);
        String usuario = "anonymous";
        String clave = "anonymous";
        try {
            cliente.connect(servFTP);
            cliente.enterLocalPassiveMode(); //modo pasivo

            boolean login = cliente.login(usuario, clave);
            if (login)
                System.out.println("Login correcto...");
            else {
                System.out.println("Login Incorrecto...");
                cliente.disconnect();
                System.exit(1);
            }
            System.out.println("Directorio actual: "
                               + cliente.printWorkingDirectory());

            FTPFile[] files = cliente.listFiles();
            System.out.println("Ficheros en el directorio actual:"
                               + files.length);
            //array para visualizar el tipo de fichero
```



```

String tipos[] = {"Fichero", "Directorio","Enlace simb."};

for (int i = 0; i < files.length; i++) {
    System.out.println("\t" + files[i].getName() + " => "
        + tipos[files[i].getType()]);
}
boolean logout = cliente.logout();
if (logout)
    System.out.println("Logout del servidor FTP...");
else
    System.out.println("Error al hacer Logout...");
//
cliente.disconnect();
System.out.println("Desconectado...");
} catch (IOException ioe) {
    ioe.printStackTrace();
}
}
} // ..

```

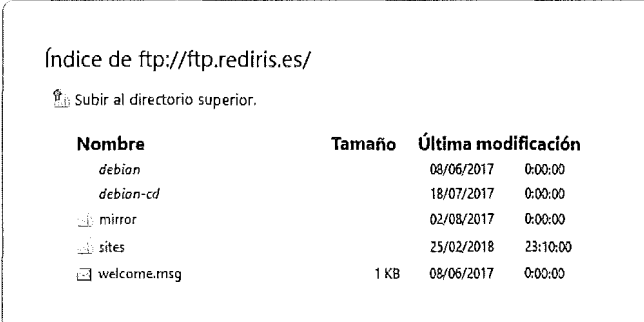
La ejecución muestra la siguiente salida (un punto designa el directorio actual, dos puntos seguidos designan el directorio de nivel superior al actual):

```


Nos conectamos a: ftp.rediris.es
Login correcto...
Directorio actual: /
Ficheros en el directorio actual:7
. => Directorio
.. => Directorio
debian => Enlace simb.
debian-cd => Enlace simb.
mirror => Directorio
sites => Directorio
welcome.msg => Fichero
Logout del servidor FTP...
Desconectado...

```

Si desde el navegador web escribimos *ftp.rediris.es* obtendremos también la lista de ficheros y directorios, véase Figura 4.4.



Índice de ftp://ftp.rediris.es/

 Subir al directorio superior.




Nombre	Tamaño	Última modificación
<i>debian</i>		08/06/2017 0:00:00
<i>debian-cd</i>		18/07/2017 0:00:00
 <i>mirror</i>		02/08/2017 0:00:00
 <i>sites</i>		25/02/2018 23:10:00
 <i>welcome.msg</i>	1 KB	08/06/2017 0:00:00

Figura 4.4. Directorio principal del sitio *ftp.rediris.es*.

Para movernos de un directorio a otro usamos el método **changeWorkingDirectory()**. Devuelve *true* si el directorio existe y *false* si no existe. En el ejemplo anterior si quiero mostrar el contenido de un directorio, primero he de situarme en él usando este método. Por ejemplo si

quiero ir al directorio *mirror/MySQL/Downloads* para obtener la lista de ficheros que hay escribo las siguientes órdenes:

```
String directorio="/mirror/MySQL/Downloads/";
if(cliente.changeWorkingDirectory (directorio))
    System.out.println("Dir Actual:"+ cliente.printWorkingDirectory());
else
    System.out.println("NO EXISTE EL DIRECTORIO: "+directorio);
FTPFile[] files = cliente.listFiles();
```

El siguiente ejemplo crea un directorio en el directorio actual (tenemos que tener permiso para poder hacerlo) y hacemos que sea el directorio de trabajo actual usando el método `changeWorkingDirectory()`:

```
String direc="NUEVODIREC";
if(cliente.makeDirectory (direc)){
    System.out.println("Directorio creado ....");
    cliente.changeWorkingDirectory (direc);
}
else
    System.out.println("NO SE HA PODIDO CREAR DIRECTORIO");
```

La clase `FTPFile` se utiliza para representar información acerca de los ficheros almacenados en un servidor FTP. Algunos métodos importantes son:

MÉTODOS	MISIÓN
<code>String getName()</code>	Devuelve el nombre del fichero.
<code>long getSize()</code>	Devuelve el tamaño del fichero en bytes.
<code>int getType()</code>	Devuelve el tipo del fichero, 0 si es un fichero (<code>FILE_TYPE</code>), 1 un directorio (<code>DIRECTORY_TYPE</code>) y 2 un enlace simbólico (<code>SYMBOLIC_LINK_TYPE</code>)
<code>String getUser()</code>	Devuelve el nombre del usuario propietario del fichero.
<code>boolean isDirectory()</code>	Devuelve <i>true</i> si el fichero es un directorio.
<code>boolean isFile()</code>	Devuelve <i>true</i> si es un fichero.
<code>boolean isSymbolicLink()</code>	Devuelve <i>true</i> si es un enlace simbólico.

ACTIVIDAD 4.1

Conéctate a *ftp.rediris.es* y visualiza los directorios del directorio raíz, entra después en cada directorio del directorio raíz mostrando la lista de ficheros y directorios que hay. Prueba en otros servidores FTP que admiten usuarios anónimos como *ftp.uv.es*, *ftp.freebsd.org*, *ftp.dit.upm.es*, *ftp.unavarra.es*, etc.

Instala el servidor FTP **Filezilla Server** en tu máquina local. Crea un directorio en el disco duro que sirva como servidor FTP. Crea un subdirectorio para cada usuario que crees y copia ficheros en ellos. Crea varios usuarios y asigna los permisos al directorio al que pueden acceder. Consulta el documento *INSTALACION_FilezillaServer.pdf* (que se encuentra en los recursos del capítulo) para instalar y configurar el servidor FTP. Puedes descargar **Filezilla Server** desde la URL: <https://filezilla-project.org/>.

Realiza la actividad anterior conectándote al servidor FTP local. El nombre del servidor es *localhost* y el nombre de usuario y clave alguno de los que hayas creado.

4.3.2. SUBIR FICHEROS AL SERVIDOR

Para los siguientes ejemplos necesitamos tener acceso a un servidor FTP. Podemos usar **Filezilla Server** o crearnos un hosting web gratuito que ofrezca servicio de FTP (<http://byethost.com/>, <https://www.hostinger.es/>, <https://www.000webhost.com/>, etc).

Para subir ficheros al servidor necesitamos un nombre de usuario, su clave, un espacio en el servidor y tener privilegios para ello. Primero nos conectamos al servidor y a continuación nos identificamos. Si todo va bien nos situamos en el directorio donde vamos a subir los ficheros; por ejemplo, supongamos que es un subdirectorio que cuelga del directorio raíz (/) y se llama NUEVODIREC:

```
String direc = "/NUEVODIREC";
cliente.changeWorkingDirectory(direc);
```

A continuación con el método **setFileType()** se indica el tipo de fichero a subir. Este tipo es una constante entera definida en la clase **FTP**. Se suele poner **BINARY_FILE_TYPE** que permite enviar ficheros de cualquier tipo:

```
cliente.setFileType(FTP.BINARY_FILE_TYPE);
```

Creamos un stream de entrada con los datos del fichero que vamos a subir (en el ejemplo el fichero se llama *EJEMPLO.pdf* y se ubica en la carpeta *D:\CLASE*) y se lo pasamos al método **storeFile()**, en el primer parámetro indicaremos el nombre que tendrá el fichero en el directorio FTP y en el segundo el **InputStream**, el método devuelve *true* si el proceso se ha realizado correctamente:

```
String archivo = "D:\\CLASE\\EJEMPLO.pdf";
BufferedInputStream in = new BufferedInputStream
                        (new FileInputStream(archivo));
if (cliente.storeFile("EJEMPLO.pdf", in))
    System.out.println("Subido correctamente... ");
else
    System.out.println("No se ha podido subir el fichero... ");
```

Por último, será necesario cerrar el flujo de entrada. El ejemplo completo se muestra a continuación, se sube un fichero al directorio NUEVODIREC, si el directorio no existe se crea. Al fichero se le da el mismo nombre en el servidor que el que tiene actualmente. El servidor FTP es *localhost* y el usuario y clave es *usuariol*:

```
import java.io.*;
import org.apache.commons.net.ftp.*;

public class SubirFichero {
    public static void main(String[] args) {
        FTPClient cliente = new FTPClient();
        String servidor = "localhost";
        String user = "usuariol";
        String pasw = "usuariol";

        try {
            System.out.println("Conectandose a " + servidor);
            cliente.connect(servidor);
            cliente.enterLocalPassiveMode();
            boolean login = cliente.login(user, pasw);

            cliente.setFileType(FTP.BINARY_FILE_TYPE);
            String direc = "/NUEVODIREC";
```

```

    if (login) {
        System.out.println("Login correcto");

        //SI EL DIRECTORIO NO EXISTE SE CREA
        if (!cliente.changeWorkingDirectory(direc)) {
            String directorio = "NUEVODIREC";

            if (cliente.makeDirectory(directorio)) {
                System.out.println("Directorio : " + directorio +
                                    " creado ...");
                cliente.changeWorkingDirectory(directorio);
            } else {
                System.out.println("No se ha podido crear el
                                    Directorio");
                System.exit(0);
            }
        }

        System.out.println("Directorio actual: " +
                            cliente.printWorkingDirectory());

        //STREAM DE ENTRADA CON EL FICHERO A SUBIR
        String archivo = "D:\\CLASE\\EJEMPLO.pdf";
        BufferedInputStream in = new BufferedInputStream
            (new FileInputStream(archivo));

        if (cliente.storeFile("EJEMPLO.pdf", in))
            System.out.println("Subido correctamente... ");
        else
            System.out.println("No se ha podido subir... ");

        in.close();           //cerrar flujo
        cliente.logout();      //logout del usuario
        cliente.disconnect();  //desconexión del servidor
    }

    } catch (IOException ioe) {
        ioe.printStackTrace();
    }
} // main
} // ..

```

Para renombrar un fichero se usa el método **rename(nombreantiguo, nombrenuevo)**. Devuelve *true* si renombra el fichero con éxito, en caso contrario devuelve *false*. Por ejemplo, renombro el fichero de nombre *EJEMPLO.pdf* que se encuentra en la carpeta */NUEVODIREC* a *EJEMPLO1.pdf*:

```

//Renombrar fichero
String direc = "/NUEVODIREC";
cliente.changeWorkingDirectory(direc);
if (cliente.rename("EJEMPLO.pdf", "EJEMPLO1.pdf"))
    System.out.println("Fichero renombrado... ");
else
    System.out.println("No se ha podido renombrar el Fichero... ");

```

Para eliminar un fichero usamos el método **deleteFile()**. Devuelve *true* si elimina el fichero y *false* si no lo elimina. Por ejemplo, para eliminar el fichero de nombre *EJEMPLO1.pdf*, ubicado en la carpeta */NUEVODIREC* escribo lo siguiente:

```
//eliminar fichero
String fichero = "/NUEVODIREC/EJEMPLO1.pdf";
if(cliente.deleteFile(fichero))
    System.out.println("Fichero eliminado... ");
else
    System.out.println("No se ha podido eliminar Fichero... ");
```

ACTIVIDAD 4.2

Crea un programa Java que te permita subir ficheros al servidor FTP local, al directorio raíz del usuario que se conecte. Utiliza la clase **JFileChooser** para seleccionar el fichero a subir de tu disco duro. Después de realizada la subida se debe mostrar un mensaje indicándolo. A continuación, muestra en consola el contenido del directorio raíz para ver si aparece el fichero recientemente subido. La Figura 4.5 muestra un momento de la ejecución del programa.

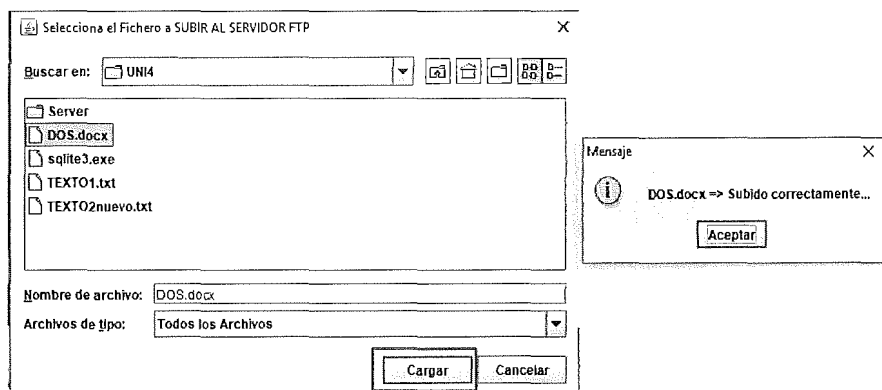


Figura 4.5. Ejecución Actividad 4.2

4.3.3. DESCARGAR FICHEROS DEL SERVIDOR

Para descargar un fichero del servidor en nuestro disco duro usamos el método **retrieveFile(String remote, OutputStream local)**. Necesitamos saber el directorio desde el que descargaremos el fichero. El método devuelve *true* si el proceso se realiza satisfactoriamente, en caso contrario devuelve *false*. Necesitaremos crear un stream de salida para escribir el fichero en nuestro disco duro.

Por ejemplo para descargar el fichero de nombre *TEXT02.TXT*, que se ubica en la carpeta del servidor FTP */NUEVODIREC/* en nuestro disco duro en la carpeta *D\CAPIT4* y con nombre *TEXT02nuevo.txt*, escribo lo siguiente:

```
//descargar fichero
String direc = "/NUEVODIREC";
cliente.changeWorkingDirectory(direc);
//stream de salida para recibir el fichero descargado
BufferedOutputStream out = new BufferedOutputStream(
    new FileOutputStream("D:\\CAPIT4\\TEXT02nuevo.txt"));
if(cliente.retrieveFile("TEXT02.txt", out))
    System.out.println("Recuperado correctamente... ");
else
    System.out.println("No se ha podido descargar... ");
out.close();
```

ACTIVIDAD 4.3

Realiza un programa Java que se conecte al servidor FTP local (localhost) con un nombre de usuario y su clave. Se debe pedir por teclado el nombre de usuario y la clave.

Una vez conectado se mostrará en pantalla la lista de ficheros (solo los ficheros, los directorios no se mostrarán) del servidor FTP que podemos descargar. Al hacer clic en el fichero se debe mostrar su nombre y si pulsamos el botón *Descargar*, nos debe permitir descargar el fichero en nuestro disco duro. Después de la descarga se mostrará un mensaje indicando si se ha realizado correctamente o no. El botón *Salir* Finaliza la aplicación. La Figura 4.6 muestra un momento de la ejecución del programa:

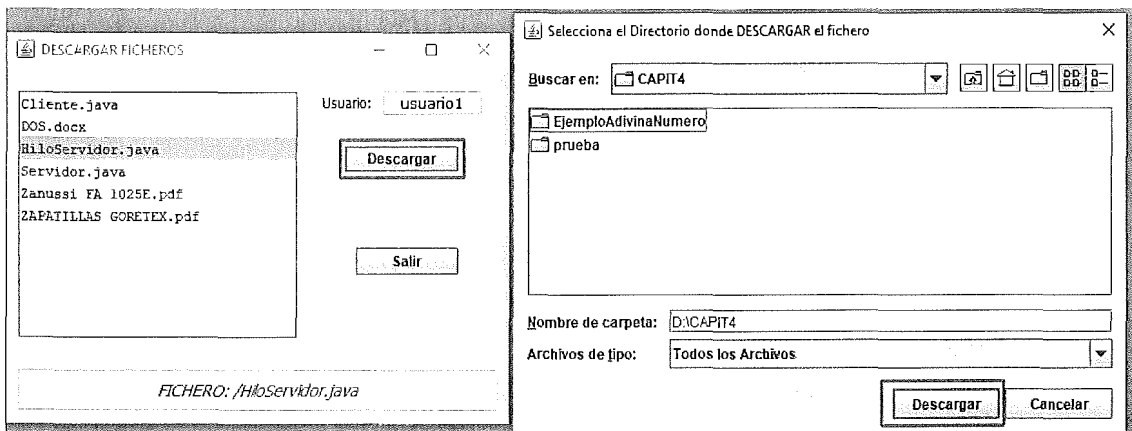


Figura 4.6. Ejecución Actividad 4.3

4.3.4. CREACIÓN DE UN CLIENTE FTP

A continuación, vamos a crear un cliente sencillo desde el que podremos subir, descargar y eliminar ficheros; y crear y eliminar directorios o carpetas en nuestro sitio FTP. Como servidor FTP usaremos *Filezilla Server* instalado en nuestra máquina local.

Para el ejemplo utilizaremos el usuario con nombre "*usuario1*" y clave "*usuario1*"; que tiene permisos sobre la carpeta *D:\MiServerFTP\usuario1* del servidor FTP. Dicha carpeta contiene ficheros y directorios.

NOTA: Para mostrar en la consola el contenido de los mensajes comando/respuesta que se van originando en la comunicación con el servidor FTP podemos usar el método *addProtocolCommandListener()* de la clase *SocketClient*. Se escribe la siguiente expresión antes de realizar la conexión al servidor:

```
cliente.addProtocolCommandListener
(new PrintCommandListener(new PrintWriter (System.out)));
```

La interfaz *ProtocolCommandListener* junto con la interfaz *PrintCommandListener* facilita esta tarea.

Los datos que necesitaremos para la conexión al servidor FTP son el nombre del servidor, el nombre del usuario y su clave. En el programa se han usado las siguientes variables para almacenar estos datos y se han asignado los siguientes valores: *servidor* = "*localhost*", *user* = "*usuario1*" y *pasw* = "*usuario1*". La pantalla inicial del cliente se muestra en la Figura 4.7.

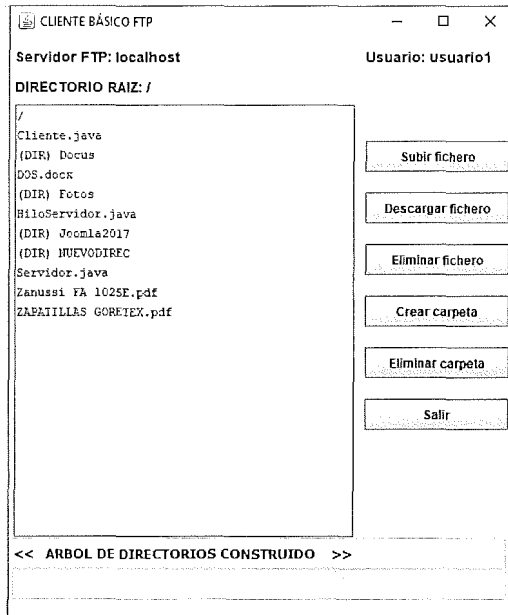


Figura 4.7. Pantalla inicial del cliente FTP básico.

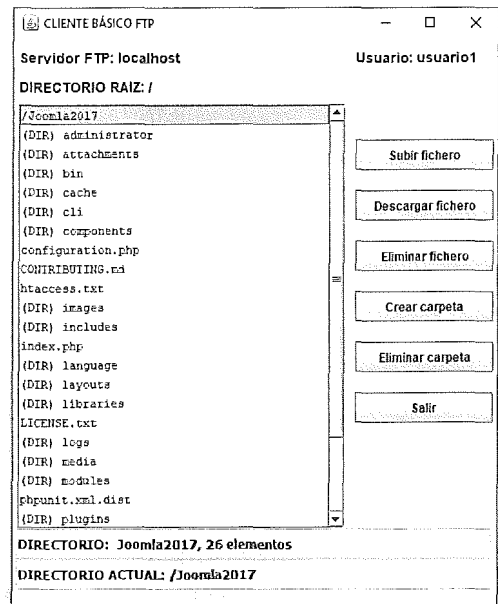


Figura 4.8. Contenido directorio Joomla2017.

En la pantalla se muestran 5 campos de texto no editables. En la parte superior se muestra el nombre del servidor, el usuario y el directorio raíz; en la parte inferior se muestran los mensajes que van surgiendo según vamos navegando por las carpetas.

A la derecha se muestran 6 botones que nos permitirán subir, descargar y eliminar ficheros; crear y eliminar carpetas y finalizar la aplicación. En el centro se muestra la lista de ficheros y carpetas del directorio actual. Estos se almacenan primero en un JList y después se visualizan.

Para distinguir un fichero de un directorio se ha añadido a la izquierda del directorio la palabra *(DIR)* seguida de un espacio en blanco. Al hacer clic en el directorio se visualiza automáticamente su contenido, por ejemplo, al hacer clic en *(DIR) Joomla2017* se muestra su contenido, véase Figura 4.8. El nombre del directorio será el primer elemento de la lista JList; al hacer clic de nuevo en él, se visualizará el contenido del directorio padre.

Se pueden observar en la Figura 4.9 los mensajes que aparecen en los campos de texto de la parte inferior. Al hacer clic en un directorio se muestra su nombre y el número de elementos que tiene, en el otro mensaje se muestra el nombre del directorio actual. Al hacer clic en un fichero se muestra su nombre y el directorio donde está, véase Figura 4.9.

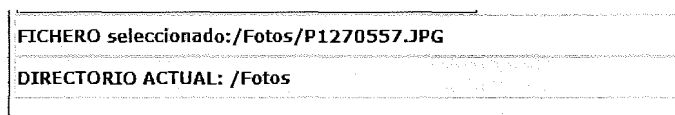


Figura 4.9. Mensajes al hacer clic en un fichero.

Utilizaremos la variable *direcInicial* para definir el directorio inicial del usuario a partir del cual realizaremos la navegación, *direcSelec* para saber en todo momento cual es el directorio de trabajo actual y *ficheroSelec* para saber el último fichero seleccionado (sobre el que hemos hecho clic). Se inicializan con los siguientes valores:

```
static String direcInicial = "/";
static String direcSelec = direcInicial;
static String ficheroSelec = "";
```

El código del programa es el siguiente, en primer lugar, se muestran los import, y los campos y variables que usa el programa:

```
import javax.swing.*;
import javax.swing.event.*;
import org.apache.commons.net.PrintCommandListener;
import org.apache.commons.net.ftp.*;
import java.io.*;
import java.awt.*;
import java.awt.event.*;

public class clienteFTPBasico extends JFrame {
    private static final long serialVersionUID = 1L;
    //Campos de cabecera parte superior
    static JTextField cab = new JTextField();
    static JTextField cab2 = new JTextField();
    static JTextField cab3 = new JTextField();

    //Campos de mensajes parte inferior
    private static JTextField campo = new JTextField();
    private static JTextField campo2 = new JTextField();

    //Botones
    JButton botonCargar = new JButton("Subir fichero");
    JButton botonDescargar = new JButton("Descargar fichero");
    JButton botonBorrar = new JButton("Eliminar fichero");
    JButton botonCreaDir = new JButton("Crear carpeta");
    JButton botonDelDir = new JButton("Eliminar carpeta");
    JButton botonSalir = new JButton("Salir");

    //Lista para los datos del directorio
    static JList listaDirec = new JList();

    //contenedor
    private final Container c = getContentPane();

    //Datos del servidor FTP - Servidor local
    static FTPClient cliente = new FTPClient();//cliente FTP
    String servidor = "localhost";
    String user = "usuario1";
    String pasw = " usuario1";
    boolean login;
    static String direcInicial = "/";

    //para saber el directorio y fichero seleccionado
    static String direcSelec = direcInicial;
    static String ficheroSelec = "";
```

En el constructor se inicializan las variables, se llena la lista con los nombres de ficheros y directorios del directorio inicial y se pinta la pantalla. El código se muestra a continuación. Se han incluido las líneas más importantes como son la conexión del cliente FTP, el establecimiento del directorio de trabajo inicial, la obtención de los ficheros de dicho directorio, el llenado de la lista de ficheros y directorios, y la colocación de la lista en la pantalla. Después se han incluido las acciones de respuesta cuando se pulsa un botón o en un elemento de la lista, más tarde se desarrollarán detalladamente:

```
public clienteFTPBasico() throws IOException {
    super("CLIENTE BÁSICO FTP");
```



```

//para ver los comandos que se originan
cliente.addProtocolCommandListener(new PrintCommandListener
    (new PrintWriter (System.out) ));
cliente.connect(servidor);          //conexión al servidor
cliente.enterLocalPassiveMode(); //modo pasivo
login = cliente.login(user, pasw);

//Se establece el directorio de trabajo actual
cliente.changeWorkingDirectory(direcInicial);

//Oteniendo ficheros y directorios del directorio actual
FTPFile[] files = cliente.listFiles();

//Construyendo la lista de ficheros y directorios
//del directorio de trabajo actual
llenarLista(files, direcInicial);

//preparar campos de pantalla
. . . . .
campo.setText("<< ARBOL DE DIRECTORIOS CONSTRUIDO >>");
cab.setText("Servidor FTP: "+servidor);
cab2.setText("Usuario: "+user);
cab3.setText("DIRECTORIO RAIZ: "+direcInicial);

//Preparación de la lista
//se configura el tipo de selección para que solo se pueda
//seleccionar un elemento de la lista
listaDirec.setSelectionMode(ListSelectionModel.
    SINGLE_SELECTION);
//barra de desplazamiento para la lista
JScrollPane barraDesplazamiento = new
    JScrollPane(listaDirec);
barraDesplazamiento.setPreferredSize
    (new Dimension(335, 420));
barraDesplazamiento.setBounds(new Rectangle
    (5, 65, 335, 420));

c.add(barraDesplazamiento);
c.setLayout(null);

//se añaden el resto de los campos de pantalla
. . . . .

setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
setSize(510, 600);
setVisible(true);

//ACCIONES AL PULSAR en la lista o en los botones
listaDirec.addListSelectionListener(new
    ListSelectionListener() {...} )
botonSalir.addActionListener(new ActionListener() {...});
botonCreaDir.addActionListener(new ActionListener() {...});
botonDelDir.addActionListener(new ActionListener() {...});
botonCargar.addActionListener(new ActionListener() {...});
botonDescargar.addActionListener(new ActionListener() {...});
botonBorrar.addActionListener(new ActionListener() {...});
} // ..fin constructor

```

Dentro del constructor se hace una llamada al método *llenarLista(files, direcInicial)*. Este método se encarga de llenar la lista JList con los nombres de ficheros y directorios y mostrarla en pantalla. Recibe como primer parámetro un objeto **FTPFile[]** con los datos de ficheros del directorio actual y el nombre del directorio de trabajo actual (parámetro *direc2*). Para añadir elementos al JList se crea un objeto *DefaultListModel*. El código es el siguiente:

```
private static void llenarLista(FTPFile[] files, String direc2) {
    if (files == null) return;

    //se crea un objeto DefaultListModel
    DefaultListModel modeloLista = new DefaultListModel();

    //se definen propiedades para la lista, color y tipo de fuente
    listaDirec.setForeground(Color.blue);
    Font fuente = new Font("Courier", Font.PLAIN, 12);
    listaDirec.setFont(fuente);

    //se eliminan los elementos de la lista
    listaDirec.removeAll();

    try {
        //se establece el directorio de trabajo actual
        cliente.changeWorkingDirectory(direc2);
    } catch (IOException e) {
        e.printStackTrace();
    }
    direcSelec = direc2; //directorio actual

    //se añade el directorio de trabajo al listmodel, primer elemento
    modeloLista.addElement(direc2);

    //se recorre el array con los ficheros y directorios
    for (int i = 0; i < files.length; i++) {
        if (!(files[i].getName()).equals("."))
            && !(files[i].getName()).equals("..")) {
            //nos saltamos los directorios . y ..
            //Se obtiene el nombre del fichero o directorio
            String f = files[i].getName();

            //Si es directorio se añade al nombre (DIR)
            if (files[i].isDirectory()) f = "(DIR) " + f;

            //se añade el nombre del fic o direc al listmodel
            modeloLista.addElement(f);
        } //fin if
    } //fin for
    try {
        //se asigna el listmodel al JList,
        //se muestra en pantalla la lista de ficheros y direc
        listaDirec.setModel(modeloLista);
    } catch (NullPointerException n) {
        ; //Se produce al cambiar de directorio
    }
} //Fin llenarLista
```

En lugar de añadir cadenas al JList, se podría haber definido una clase con un atributo (*files[i]* de tipo **FTPFile**) y sobrescribir el método *toString()* para que si se trata de fichero devuelva el

nombre y si es un directorio devuelva “(DIR) “ + nombre; y en este caso se añadirían objetos al JList.

Para saber cuando se selecciona un elemento de la lista JList hay que implementar **ListSelectionListener** y desarrollar el método **valueChanged()** que recibe un objeto **ListSelectionEvent**. Para detectar si el usuario selecciona un elemento usamos el método **getValueIsAdjusting()** que devuelve *true* si el usuario ha pulsado sobre él. El código es el siguiente:

```
listaDirec.addListSelectionListener(new ListSelectionListener() {
    public void valueChanged(ListSelectionEvent lse) {
        if (lse.getValueIsAdjusting()) {
            ficheroSelec = "";

            //elemento que se ha seleccionado de la lista
            String fic = listaDirec.getSelectedValue().toString();
```

Con el método **getSelectedValue()** podemos saber el valor del elemento seleccionado. A continuación se comprueba si el elemento seleccionado es el primero de la lista (**listaDirec.getSelectedIndex() == 0**); recordemos que el primer elemento de la lista representa el directorio de trabajo actual. Si es así, necesitamos saber si es el directorio inicial. Si es el directorio inicial no se hace nada (no hay un directorio padre que mostrar); y si no lo es, hay que cambiar el directorio de trabajo actual al directorio padre del seleccionado. Será necesario llamar al método **llenarLista()** para llenar la JList con el contenido del nuevo directorio de trabajo:

```
if (listaDirec.getSelectedIndex() == 0) {
    //Se hace clic en el primer elemento del JList
    //se comprueba si es el directorio inicial
    if (!fic.equals(direcInicial)) {
        //si no estamos en el directorio inicial, hay que
        //subir al directorio padre
        try {
            cliente.changeToParentDirectory();
            direcSelec = cliente.printWorkingDirectory();
            FTPFile[] ff2 = null;
            //directorio de trabajo actual=directorio padre
            cliente.changeWorkingDirectory(direcSelec);
            //se obtienen ficheros y directorios
            ff2 = cliente.listFiles();
            campo.setText("");
            //se llena la lista con fich. del directorio padre
            llenarLista(ff2, direcSelec);
        } catch (IOException e) {e.printStackTrace();}
    }
}
```

Si se selecciona un elemento que no es el primero de la lista hay que comprobar si se trata de un fichero o de un directorio. Para saber si es un directorio se comprueba si los 6 primeros caracteres coinciden con la cadena “(DIR) “. En este caso se construye una cadena (**direcSelec2**), que contenga el nuevo directorio de trabajo (directorio actual + elemento seleccionado). Por ejemplo si el directorio actual es */Docus* y se pulsa en el directorio de nombre *carpeta*, se generará la cadena: */Docus/carpeta*. Si el directorio que se ha seleccionado está dentro del directorio / solo hay que añadir a éste el nombre del directorio. Para establecer el nuevo directorio de trabajo se pasa la cadena como argumento al método **changeWorkingDirectory()**. Una vez más, para visualizar el contenido de este directorio se llamará a **llenarLista()**:

```
//No se hace clic en el primer elemento del JList
```

```

//Puede ser un fichero o un directorio
else {
    if (fic.substring(0, 6).equals("(DIR) ")) {
        //SE TRATA DE UN DIRECTORIO
        try {
            fic = fic.substring(6);
            String direcSelec2 = "";
            if (direcSelec.equals("/"))
                direcSelec2 = direcSelec + fic;
            else
                direcSelec2 = direcSelec + "/" + fic;

            cliente.changeWorkingDirectory(direcSelec2);
            FTPFile[] ff2 = cliente.listFiles();
            campo.setText("DIRECTORIO: " + fic + ", "
                + ff2.length + " elementos");
            //directorio actual
            direcSelec = direcSelec2;
            //se llena la lista con datos del directorio
            llenarLista(ff2, direcSelec);
        } catch (IOException e2) {
            e2.printStackTrace();
        }
    }
}

```

Si se trata de un fichero se construye el nombre completo del mismo y se almacena en la variable *ficheroSelec*. Por ejemplo si seleccionamos un fichero de nombre *P1270557.JPG* ubicado en el directorio */Fotos*, el nombre completo que se genera es */Fotos/ P1270557.JPG*. Para generar el nombre completo de los ficheros contenidos en el directorio / solo hay que añadir a este el nombre del fichero:

```

else {
    // SE TRATA DE UN FICHERO
    ficheroSelec = direcSelec;
    if (direcSelec.equals("/"))
        ficheroSelec += fic;
    else
        ficheroSelec += "/" + fic;
    campo.setText("FICHERO seleccionado:" +
        ficheroSelec);
    ficheroSelec = fic;//me quedo con el nombre
} //fin else

} //fin else de fichero o directorio
campo2.setText("DIRECTORIO ACTUAL: " + direcSelec);
} //fin if inicial
}
}); // fin acción en JList

```

En los mensajes de la parte inferior de la pantalla se irán mostrando el nombre del fichero o directorio y el directorio actual. A continuación, se muestran las acciones de los botones. Empezamos por el botón **Salir**, la acción a realizar es desconectar el cliente y finalizar la aplicación:

```

botonSalir.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        try {
            cliente.disconnect();

```

```

    } catch (IOException e1) {
        e1.printStackTrace();
    }
    System.exit(0);
}
});

```

El botón *Subir fichero* sube un fichero al servidor FTP al directorio de trabajo actual. Al pulsar en el botón se muestra una ventana desde la que podremos navegar por nuestro ordenador y elegir el fichero a subir al servidor, véase Figura 4.10; se ha utilizado para ello la clase **JFileChooser** que proporciona una interfaz gráfica para seleccionar un fichero de una lista.

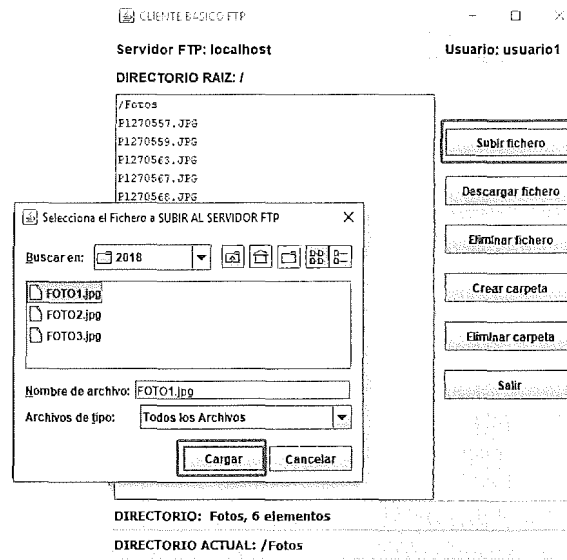


Figura 4.10. Subir un fichero.

Al pulsar en el botón *Cargar*, se muestra un mensaje indicando si el fichero se subió correctamente. Al pulsar en *Aceptar* se muestra en el directorio actual los ficheros anteriores más el nuevo fichero, véase Figura 4.11.

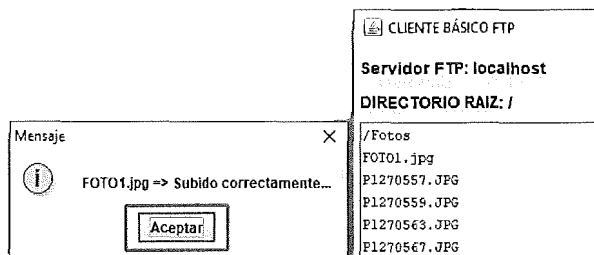


Figura 4.11. Mensaje de fichero subido correctamente.

La acción para este botón es la siguiente: primero se definen los objetos **JFileChooser** y **File**. La clase **JFileChooser** tiene el método **showDialog()** que devuelve un entero que indica si el usuario ha seleccionado un fichero, en el segundo parámetro se define un String con la etiqueta del botón para aceptar la selección (en el ejemplo es *Cargar*). Si el entero es igual a **JFileChooser.APPROVE_OPTION** es que el usuario ha seleccionado un fichero y ha pulsado el botón *Cargar*, entonces se llama al método **getSelectedFile()** para obtener un objeto **File**, que representa el fichero elegido:

```

botonCargar.addActionListener(new ActionListener() {

```

```

public void actionPerformed(ActionEvent e) {
    JFileChooser f = new JFileChooser();

    //solo se pueden seleccionar ficheros
    f.setFileSelectionMode(JFileChooser.FILES_ONLY);
    //título de la ventana
    f.setDialogTitle("Selecciona el Fichero a SUBIR
                                                                AL SERVIDOR FTP");

    //se muestra la ventana
    int returnVal = f.showDialog(f, "Cargar");
    if (returnVal == JFileChooser.APPROVE_OPTION) {
        //fichero seleccionado
        File file = f.getSelectedFile();
        //nombre completo del fichero
        String archivo = file.getAbsolutePath();
        //solo nombre del fichero
        String nombreArchivo = file.getName();
        try {
            SubirFichero(archivo, nombreArchivo);
        } catch (IOException el) { el.printStackTrace(); }
    }
}
}); //Fin boton subir

```

Una vez seleccionado se realiza una llamada al método *SubirFichero(archivo, nombreArchivo)* enviando como primer parámetro el nombre completo del fichero a subir (incluyendo el directorio) y como segundo parámetro el nombre del fichero. Dentro de este método se define el tipo de fichero a subir al servidor FTP (*FTP.BINARY_FILE_TYPE*), se crea un stream de entrada con los datos del fichero que se va a subir (nombre completo del fichero) y se pasa como argumento al método *storeFile()*, en el primer parámetro se indica el nombre que tendrá el fichero en el directorio FTP, antes será necesario establecer el directorio de trabajo actual:

```

private boolean SubirFichero
    (String archivo, String soloNombre) throws IOException {
    cliente.setFileType(FTP.BINARY_FILE_TYPE);
    BufferedInputStream in = new BufferedInputStream
        (new FileInputStream(archivo));

    boolean ok = false;
    //directorio de trabajo actual
    cliente.changeWorkingDirectory(direcSelec);
    if (cliente.storeFile(soloNombre, in)) {
        String s = " " + soloNombre + " => Subido correctamente... ";
        campo.setText(s);
        campo2.setText("Se va a actualizar el árbol de directorios...");
        JOptionPane.showMessageDialog(null, s);

        //obtener ficheros del directorio actual
        FTPFile[] ff2 = cliente.listFiles();
        //llenar la lista con los ficheros del directorio actual
        llenarLista(ff2, direcSelec);
        ok = true;
    } else
        campo.setText("No se ha podido subir... " + soloNombre);
    return ok;
} //.. SubirFichero

```

Una vez subido el fichero se muestra una ventanita indicándonos si se ha subido correctamente, se utiliza el método `JOptionPane.showMessageDialog()`. También se visualizará un mensaje en los campos de texto de la parte inferior de la pantalla. Después se llama al método `llenarLista()` para que muestre el directorio actual con el fichero que se acaba de subir.

Para **descargar un fichero** del servidor FTP será necesario seleccionarlo en el JList y pulsar el botón *Descargar fichero*. Se abre una ventana desde la que tendremos que seleccionar la carpeta donde descargarlo en nuestro ordenador, Figura 4.12. A continuación se pulsa en el botón *Descargar*. En una ventanita se mostrará un mensaje indicando si la operación se ha realizado correctamente, Figura 4.13.

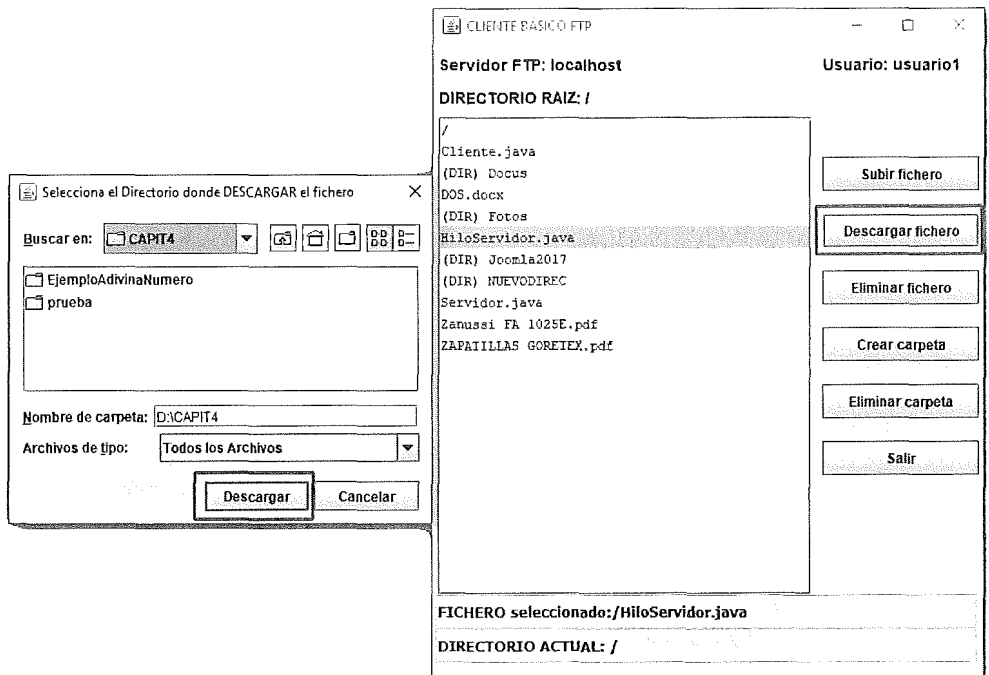


Figura 4.12. Descargar un fichero.

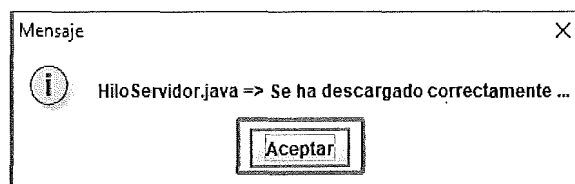


Figura 4.13. Mensaje de fichero descargado correctamente.

La acción para este botón es la siguiente: en primer lugar hay que construir el nombre completo del fichero seleccionado (es decir, nombre de directorio + nombre de fichero), que dependerá de si está en el directorio / o en otro directorio. Una vez construido se realiza una llamada al método `DescargarFichero(directorio + ficheroSelec, ficheroSelec)`:

```

botonDescargar.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        String directorio=direcSelec;
        if (!direcSelec.equals("/"))
            directorio = directorio + "/";
        if (!ficheroSelec.equals("")) {
            DescargarFichero(directorio + ficheroSelec ,ficheroSelec);
        }
    }
}

```

```

    }
}
}); // Fin botón descargar

```

El código del método *DescargarFichero()* se muestra a continuación. Se utilizará la clase **JFileChooser** para elegir la carpeta donde descargar el fichero, los pasos son similares a la carga del fichero. Para almacenar el fichero en nuestro ordenador crearemos un stream de salida sobre el directorio seleccionado + el nombre de fichero a descargar. Mediante el método **retrieveFile()** recuperaremos el fichero del servidor para almacenarlo en nuestro disco duro:

```

private void DescargarFichero(String NombreCompleto,
                               String nombreFichero) {
    String archivoyCarpetaDestino = "";
    String carpetaDestino = "";
    JFileChooser f = new JFileChooser();

    //solo se pueden seleccionar directorios
    f.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
    //título de la ventana
    f.setDialogTitle("Selecciona el Directorio donde DESCARGAR el
                               fichero");

    int returnVal = f.showDialog(null, "Descargar");
    if (returnVal == JFileChooser.APPROVE_OPTION) {
        File file = f.getSelectedFile();
        //obtener carpeta de destino
        carpetaDestino = (file.getAbsolutePath()).toString();
        //construimos el nombre completo que se creará en nuestro disco
        archivoyCarpetaDestino = carpetaDestino + File.separator
                                   + nombreFichero;

        try {
            cliente.setFileType(FTP.BINARY_FILE_TYPE);
            BufferedOutputStream out = new BufferedOutputStream(
                new FileOutputStream(archivoyCarpetaDestino));
            if (cliente.retrieveFile(NombreCompleto, out))
                JOptionPane.showMessageDialog(null, nombreFichero
                    + " => Se ha descargado correctamente ...");
            else
                JOptionPane.showMessageDialog(null, nombreFichero
                    + " => No se ha podido descargar ...");
            out.close();
        } catch (IOException e1) {e1.printStackTrace();}
    }
} // ..DescargarFichero

```

Al finalizar se muestra un mensaje indicándonos si el proceso se ha realizado correctamente.

Para **eliminar un fichero** del servidor FTP será necesario seleccionarlo y pulsar el botón *Eliminar fichero*. Se abre una ventana que nos pregunta si deseamos borrar el fichero seleccionado, Figura 4.14. Al pulsar el botón *Sí* se eliminará el fichero, se visualizará una ventanita indicándonos si el proceso se ha realizado correctamente. Inmediatamente se visualizará el directorio actual sin el fichero eliminado.

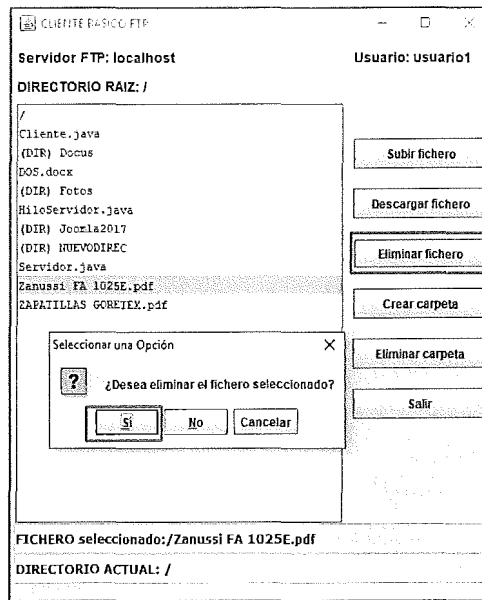


Figura 4.14. Eliminar un fichero.

La acción para este botón es similar a la acción del botón para descargar fichero: en primer lugar hay que construir el nombre completo del fichero seleccionado y después se realiza una llamada al método *BorrarFichero* (*directorio + ficheroSelec, ficheroSelec*):

```

botonBorrar.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        String directorio=direcSelec;
        if (!direcSelec.equals("/"))
            directorio =directorio +"/";
        if (!ficheroSelec.equals(""))
            BorrarFichero(directorio + ficheroSelec, ficheroSelec);
    }
}); //boton borrar

```

El código del método *BorrarFichero()* es el siguiente, en primer lugar se muestra un *JOptionPane.showConfirmDialog* que nos pide confirmación para el borrado. Devuelve un valor entero, si es igual a *JOptionPane.OK_OPTION* es que hemos pulsado en el botón *Sí* para confirmar el borrado del fichero; en este caso se realiza una llamada al método *deleteFile()* llevando como argumento el nombre completo del fichero a eliminar; se mostrará un mensaje indicando si el borrado se ha realizado correctamente. Después se llama al método *llenarLista()* para que muestre el directorio actual sin el fichero que se ha eliminado:

```

private void BorrarFichero(String NombreCompleto, String
                                nombreFichero) {

    //pide confirmación
    int seleccion = JOptionPane.showConfirmDialog(null,
        "¿Desea eliminar el fichero seleccionado?");
    if(seleccion==JOptionPane.OK_OPTION) {
        try {
            if (cliente.deleteFile (NombreCompleto)) {
                String m= nombreFichero + " => Eliminado correctamente... ";
                JOptionPane.showMessageDialog(null, m);
                campo.setText(m);
                //directorio de trabajo actual
                cliente.changeWorkingDirectory(direcSelec);
            }
        }
    }
}

```

```

//obtener ficheros del directorio actual
FTPFile[] ff2 = cliente.listFiles();
//llenar la lista con los fichero del directorio actual
llenarLista(ff2, direcSelec);
} else
    JOptionPane.showMessageDialog(null, nombreFichero
        + " => No se ha podido eliminar ...");
} catch (IOException e1) {e1.printStackTrace();}
}
} // ..BorrarFichero

```

Para **crear una carpeta** en el directorio seleccionado del servidor FTP se pulsa el botón *Crear carpeta*. Se abre una ventanita desde la que tendremos que escribir su nombre, Figura 4.15.

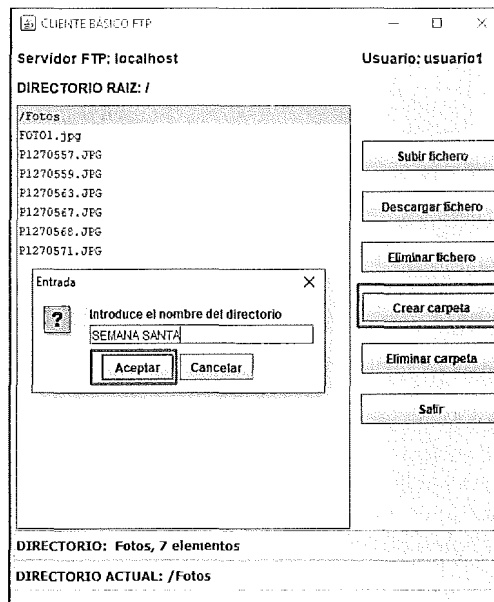


Figura 4.15. Crear una carpeta.

Al pulsar en el botón *Aceptar* se muestra un mensaje indicándonos si se ha creado correctamente, tendremos que pulsar el botón *Aceptar* para continuar; se visualiza la lista de ficheros del directorio actual con la nueva carpeta, Figura 4.16.

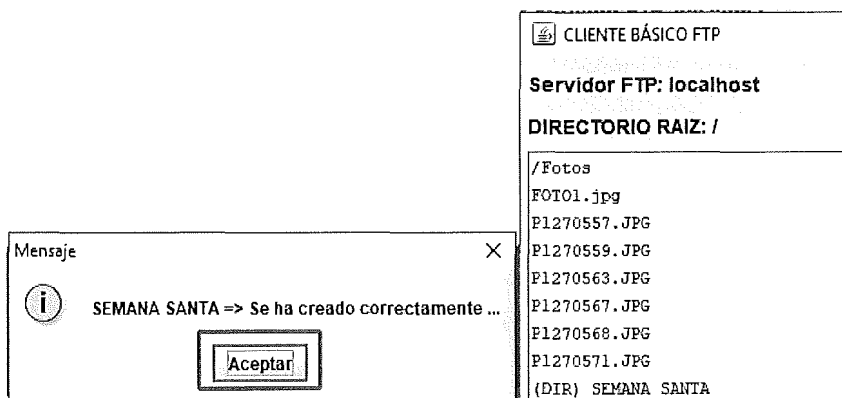


Figura 4.16. Mensaje de carpeta creada correctamente.

La acción para este botón es la siguiente: en primer lugar, se muestra una ventana de diálogo (*JOptionPane.showInputDialog*) desde la que escribimos el nombre del directorio, por defecto aparece como nombre *carpeta*. Una vez que tenemos el nombre se construye el nombre completo del directorio a crear (variable *directorio*). Se utiliza el método **makeDirectory(directorio)** para crear el directorio; una vez creado se visualiza un mensaje con *JOptionPane.showMessageDialog()* indicando si se ha creado correctamente, a continuación se llama al método *llenarLista()* para que muestre la lista de ficheros y directorios del directorio actual actualizada:

```

botonCreaDir.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        String nombreCarpeta = JOptionPane.showInputDialog(null,
            "Introduce el nombre del directorio", "carpeta");
        if (!(nombreCarpeta==null)) {
            String directorio=direcSelec;
            if (!direcSelec.equals("/")) directorio = directorio +"/";

            //nombre del directorio a crear
            directorio += nombreCarpeta.trim();//quita blancos a der e izd

            try {
                if (cliente.makeDirectory(directorio)) {
                    String m= nombreCarpeta.trim()+ " => Se ha creado
                        correctamente ...";
                    JOptionPane.showMessageDialog(null, m);
                    campo.setText(m);
                    //directorio de trabajo actual
                    cliente.changeWorkingDirectory(direcSelec);

                    //obtener ficheros del directorio actual
                    FTPFile[] ff2 = cliente.listFiles();
                    //llenar la lista
                    llenarLista(ff2, direcSelec);
                } else
                    JOptionPane.showMessageDialog(null,
                        nombreCarpeta.trim()+ " => No se ha podido crear ...");

                } catch (IOException e1) {e1.printStackTrace();}
            }//if
        }
    });//.. botonCreaDir

```

Para **eliminar una carpeta** en el directorio seleccionado del servidor FTP se pulsa el botón *Eliminar carpeta*. El proceso es similar al de crear una carpeta, se muestra una ventanita desde la que escribiremos el nombre de carpeta a eliminar y se pulsa el botón *Aceptar*. Solo se podrán eliminar carpetas si están vacías. Se mostrará un mensaje indicando si se ha podido eliminar o no la carpeta. Para eliminar el directorio se usa el método **removeDirectory(directorio)**. La acción para este botón es la siguiente:

```

botonDelDir.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        String nombreCarpeta = JOptionPane.showInputDialog(null,
            "Introduce el nombre del directorio a eliminar", "carpeta");
        if (!(nombreCarpeta==null)) {
            String directorio=direcSelec;
            if (!direcSelec.equals("/"))

```

```

        directorio =directorio +"/";
//nombre del directorio a eliminar
directorio += nombreCarpeta.trim() ;

try {
    if (cliente.removeDirectory(directorio)) {
        String m= nombreCarpeta.trim()+
            " => Se ha eliminado correctamente ...";
        JOptionPane.showMessageDialog(null, m);
        campo.setText(m);
        cliente.changeWorkingDirectory(direcSelec);

        FTPFile[] ff2 = cliente.listFiles();
        //llenar la lista
        llenarLista(ff2, direcSelec);
    } else
        JOptionPane.showMessageDialog(null,
            nombreCarpeta.trim() +
            " => No se ha podido eliminar ...");

    } catch (IOException e1) {e1.printStackTrace();}
} //if
}); //...botonDelDir

```

Por último la función *main()* presenta el siguiente aspecto:

```

public static void main(String[] args) throws IOException {
    new clienteFTPBasico();
} // .fin main

```

4.4. COMUNICACIÓN CON UN SERVIDOR SMTP

SMTP (*Simple Mail Transfer Protocol*) es el protocolo estándar de Internet para el intercambio de correo electrónico. Funciona con comandos de texto que se envían al servidor SMTP (por defecto, al puerto 25). A cada comando que envía el cliente le sigue una respuesta del servidor compuesta por un número y un mensaje descriptivo. Las especificaciones de este protocolo se definen en la RFC 2821.

4.4.1. INSTALACIÓN DE UN SERVIDOR DE CORREO ELECTRÓNICO

Un servidor SMTP es un programa que permite enviar correo electrónico a otros servidores SMTP. Vamos a utilizar el servidor SMTP **Mercury Mail** que proporciona el paquete XAMPP.

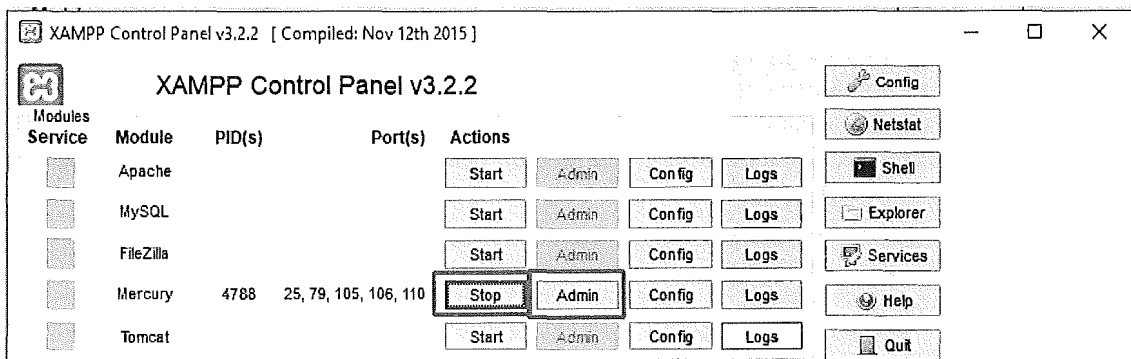


Figura 4.17. Inicio de los servicios de Mercury Mail.

Abrimos el panel de control de XAMPP y pulsamos en el botón *Start* que aparece al lado de Mercury para iniciar los servicios. Se debe marcar en verde y al lado debe aparecer el PID y los puertos usados por **Mercury Mail** (véase Figura 4.17). Pulsamos en el botón *Admin* para configurar el servidor. Se pulsa en la opción de menú *Configuration/Protocol Modules*, se configuran los parámetros como se muestra en la Figura 4.18 y se pulsa el botón *OK*.

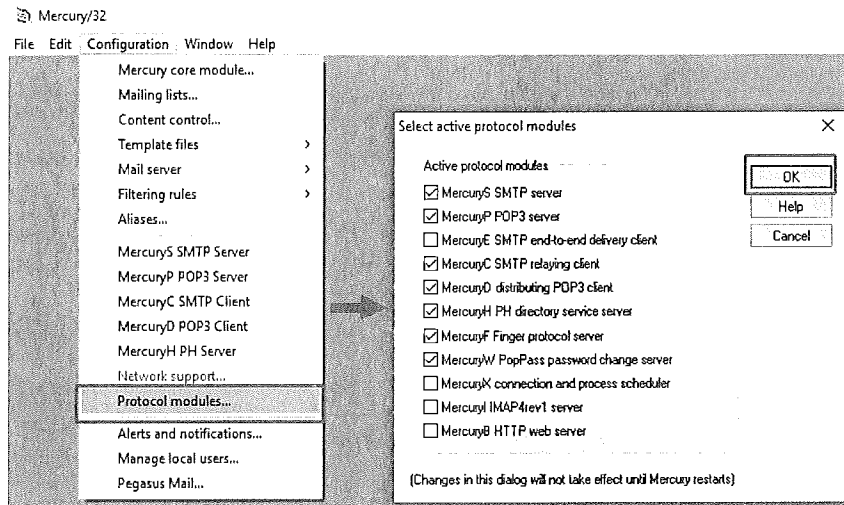


Figura 4.18. Mercury Mail Protocolo modules.

Una vez configurados estos parámetros se cierra Mercury y se vuelve a abrir. A continuación, se configura el cliente SMTP. Se pulsa en la opción *Configuration/MercuryC SMTP Client*. Para mandar emails al exterior necesitamos los datos de un correo exterior, en la aplicación se ha usado el correo de gmail: USUARIO@gmail.com, por tanto en el campo *Smart host name* se escriben los datos de SMTP de gmail para correos salientes: **smtp.gmail.com**; en puerto escribimos 587, luego se elige *SSL encryption via STARTTLS command*. En *Login username* se pone la cuenta de correo de gmail y en *Password* la contraseña del correo gmail. Se pulsa el botón *OK*, véase Figura 4.19.

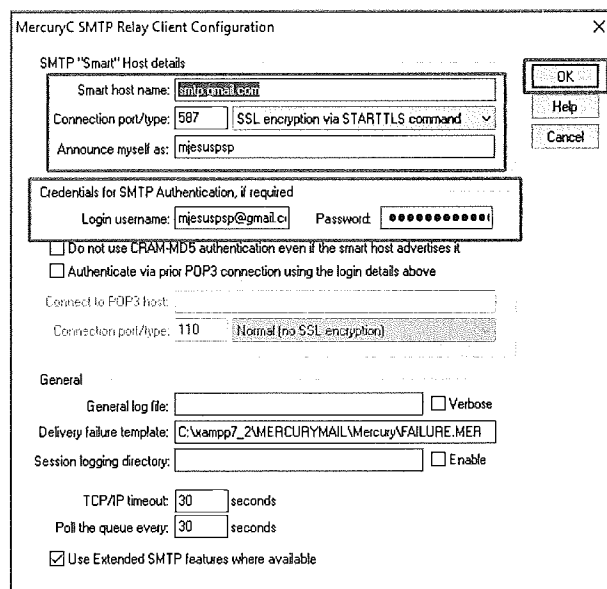


Figura 4.19. MercuryC SMTP Client.

Desde la opción de menú **Configuration/Manage local users** comprobamos que tenemos los usuarios *Admin* y *postmaster* con permisos de administrador, véase Figura 4.20. Desde la opción de menú **Configuration/MercuryS SMTP Server** y desde la pestaña *Connection Control* desmarcamos la casilla *Do not permit SMTP relaying of non-local mail*, y pulsamos el botón *Aceptar*, véase Figura 4.21.

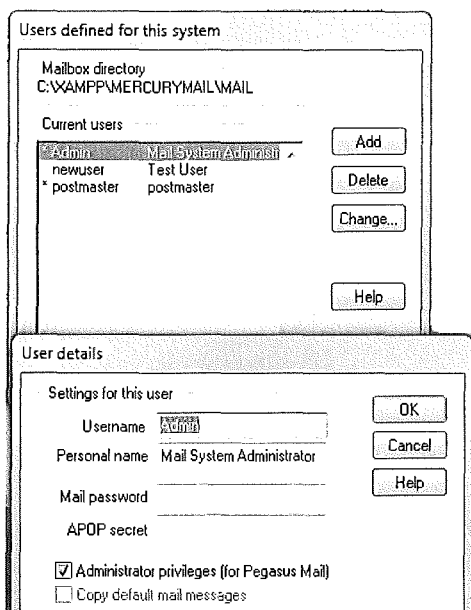


Figura 4.20. Manage local users.

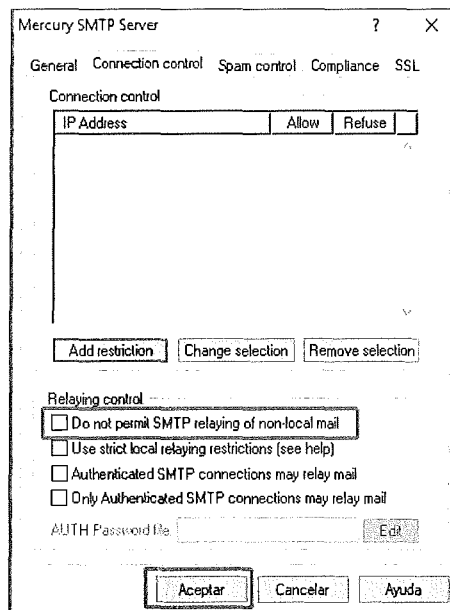


Figura 4.21. MercuryS SMTP Server.

Seguidamente se modifica el archivo **php.ini** que se localiza en la carpeta *C:\xampp\php*. Dentro de la sección **[mail function]** comprobamos que las líneas siguientes están así:

```
SMTP = localhost
```

```
smtp_port = 25
```

Y añadimos la siguiente línea:

```
sendmail_from = postmaster@localhost
```

Una vez hecho esto reiniciamos Mercury. Para probarlo, desde la opción de menú **File/Send mail message** enviamos un email a un correo externo.

Normalmente cuando creamos una cuenta de correo en un proveedor de servicios de internet, el proveedor nos proporciona los datos del servidor POP3 (o IMAP) y del servidor SMTP. Estos son necesarios para configurar clientes de correo como Microsoft Outlook, Mozilla Thunderbird, Eudora, etc. El primero se utiliza para recibir los mensajes (es decir, para configurar el correo entrante) y el segundo para enviar nuestros mensajes (configurar el correo saliente). Podemos usar el servidor SMTP que hemos instalado para enviar correos externos.

4.4.2. USO DE TELNET PARA COMUNICAR CON EL SERVIDOR SMTP

Vamos a ver a continuación como enviar un correo electrónico de forma manual usando el cliente Telnet al puerto 25 del servidor SMTP. Hemos de tener instalado en nuestra máquina el cliente telnet. Para comprobar si lo tenemos activado nos vamos a la línea de comandos del DOS (en sistemas Windows) y ejecutamos el comando telnet. Si no está activado, en Windows 10 pulsamos en el botón Inicio/ Configuración/ Aplicaciones/ Programas y características/ Activar y

desactivar las características de Windows, marcamos la casilla **Cliente Telnet** y pulsamos *Aceptar*, véase Figura 4.22.

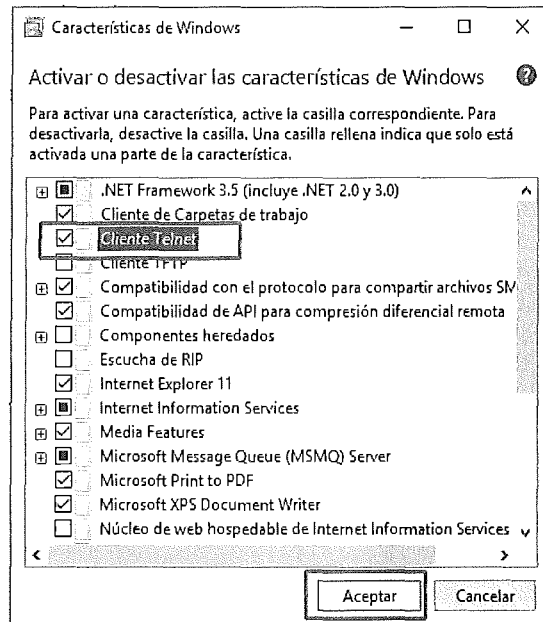


Figura 4.22. Activar el Cliente Telnet en Windows

Algunos de los comandos que usaremos se muestran en la siguiente tabla:

COMANDOS	MISIÓN
HELO o EHLO	Se utiliza para abrir una sesión con el servidor.
MAIL FROM: <i>origen</i>	A la derecha se indica quien envía el mensaje, por ejemplo: remitente@servidor.com
RCPT TO: <i>destino</i>	A la derecha se indica el destinatario del mensaje, por ejemplo: destinatario@servidor.com
DATA <i>mensaje</i>	Se utiliza para indicar el comienzo del mensaje, éste finalizará cuando haya una línea únicamente con un punto.
QUIT	Cierra la sesión.
HELP	Muestra la lista de comandos SMTP que el servidor admite.

Nos vamos a la línea de comandos del DOS y escribimos **telnet**. A continuación, escribimos: **open localhost 25**. El servidor nos responde con la siguiente línea:

```
220 localhost ESMTP server ready.
```

Normalmente responde con un número de 3 dígitos donde cada uno tiene un significado especial (como en FTP). Por ejemplo, los números que empiezan en 2 (220, 250, ...) indican que la acción se ha completado con éxito; los que empiezan por 3 (354) indican que el comando ha sido aceptado, pero la acción solicitada está suspendida a la espera de recibir más información, se usa en grupo de secuencias de comandos (**DATA**). Los que empiezan por 4 indican que el comando no fue aceptado, pero se puede volver a escribir de nuevo. Los que empiezan por 5

indican que el comando no ha sido aceptado y la acción no se ha realizado, por ejemplo, *502 Unknown command*.

Empezamos a escribir los comandos, primero se abre la sesión con **HELO**, a continuación, escribimos el origen (**MAIL FROM:**) y el destino del mensaje (**RCPT TO:**), por cada línea que vamos escribiendo el servidor nos responde. Por último, mediante el comando **DATA** enviamos el mensaje, para finalizar el mensaje escribimos una línea únicamente con un punto. Para finalizar escribimos **QUIT**. En el siguiente ejemplo se escribe un correo a la dirección *mariajesusramos@brianda.net* que se usará como origen y destino, la Figura 4.23 muestra la secuencia completa:

```
HELO ↵
250 localhost Hello, .
MAIL FROM: mariajesusramos@brianda.net ↵
250 Sender OK - send RCPTs.
RCPT TO: <mariajesusramos@brianda.net> ↵
250 Recipient OK - send RCPT or DATA.
DATA ↵
354 OK, send data, end with CRLF.CRLF
Subject: Prueba de correo ↵
Hola ↵
este es un mensaje de correo ↵
enviado usando Telnet. ↵

Adios. ↵
. ↵
250 Data received OK.
QUIT ↵
```

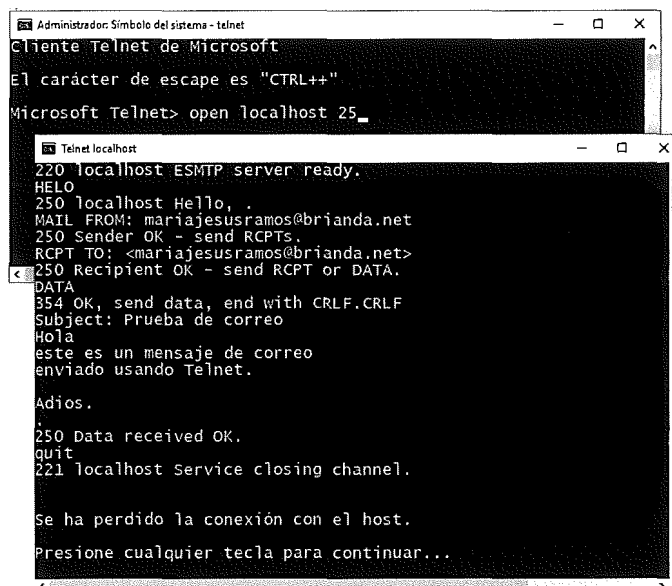


Figura 4.23. Envío de correo mediante Telnet.

Con el comando **DATA** empieza el cuerpo del mensaje. Si el servidor responde con el mensaje *354 OK, send data, end with CRLF.CRLF*, se puede empezar a escribir el cuerpo que puede contener las siguientes cabeceras: *Date*, *Subject*, *To*, *Cc* y *From* (se pueden escribir en mayúsculas o minúsculas).

Una vez realizados los pasos anteriores comprobamos si se ha recibido el correo, véase la Figura 4.24. Aunque en **MAIL FROM** pongamos una dirección de correo origen, la dirección que aparecerá en el destino es la que tenemos configurada con MERCURY.

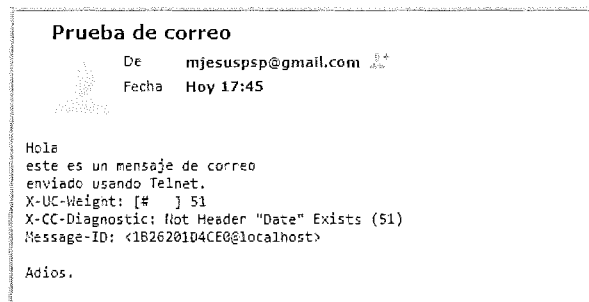


Figura 4.24. Recepción y visualización del correo enviado.

Al usar la cuenta de Gmail para configurar el servidor SMTP local y enviar correos, Google nos avisará para que revisemos el intento de inicio de sesión bloqueado, en este caso hemos de permitir el acceso a nuestra cuenta a aplicaciones menos seguras, véase Figura 4.25.

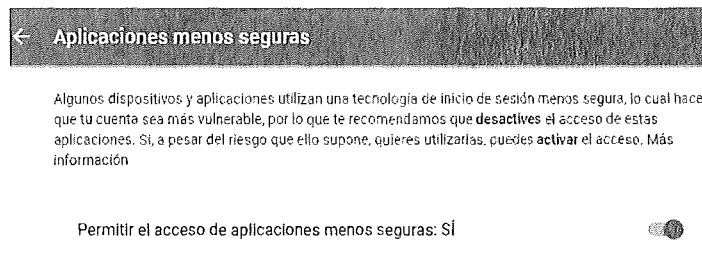


Figura 4.25. Permitir acceso a aplicaciones menos seguras.

4.4.3. JAVA PARA COMUNICAR CON UN SERVIDOR SMTP

La librería **Apache Commons Net™** proporciona la clase **SMTPClient** (extiende **SMTP**) que encapsula toda la funcionalidad necesaria para enviar ficheros a través de un servidor SMTP. Esta clase se encarga de todos los detalles de bajo nivel de interacción con un servidor SMTP. Al igual que con todas las clases derivadas de **SocketClient**, antes de hacer cualquier operación es necesario conectarse al servidor y una vez finalizada la interacción con el servidor es necesario desconectarse. Una vez conectados es necesario comprobar el código de respuesta SMTP para ver si la conexión se ha realizado correctamente.

El siguiente ejemplo realiza una conexión al servidor SMTP local (por defecto se conecta al puerto 25, en el método **connect()** no es necesario indicarlo) y después se desconecta:

```
import java.io.IOException;
import org.apache.commons.net.smtp.*;

public class ClienteSMTP1 {
    public static void main(String[] args) {
        SMTPClient client = new SMTPClient();
        try {
            int respuesta;
            //NOS CONECTAMOS AL PUERTO 25
            client.connect("localhost");

            System.out.print(client.getReplyString());
            respuesta = client.getReplyCode();
```

```

        if (!SMTPReply.isPositiveCompletion(respuesta)) {
            client.disconnect();
            System.err.println("CONEXION RECHAZADA.");
            System.exit(1);
        }

        // REALIZAR ACCIONES, por ejemplo enviar un correo

        // NOS DESCONECTAMOS
        client.disconnect();

    } catch (IOException e) {
        System.err.println("NO SE PUEDE CONECTAR AL SERVIDOR.");
        e.printStackTrace();
        System.exit(2);
    }

} //main
} // ..

```

La salida que se muestra al ejecutar el programa es:

```
220 localhost ESMTP server ready.
```

La clase **SMTPReply** (similar a **FTPReply**) almacena un conjunto de constantes para códigos de respuesta SMTP. Para interpretar el significado de los códigos se puede consultar la RFC 2821 (<http://tools.ietf.org/html/rfc2821>). El método **isPositiveCompletion(int respuesta)** devuelve *true* si un código de respuesta ha terminado positivamente. Los métodos **getReplyString()** y **getReplyCode()** son métodos de la clase **SMTP** y son similares a los vistos en la clase **FTP**.

SMTPClient presenta dos tipos de constructores:

CONSTRUCTOR	MISIÓN
SMTPClient()	Constructor por defecto.
SMTPClient(String codificación)	Se establece una codificación en el constructor (<i>BASE64</i> , <i>BINARY</i> , <i>8BIT</i> , etc.)

La clase utiliza el método **connect()** de la clase **SocketClient** para conectarse al servidor; y el método **disconnect()** de la clase **SMTP** para desconectarse del servidor SMTP. Para conectarnos a un servidor SMTP cuyo puerto de escucha sea distinto al 25, tendríamos que indicar en la conexión el número de puerto: **connect(host, puerto)**.

Algunos métodos de esta clase son:

MÉTODOS	MISIÓN
boolean addRecipient(String address)	Añade la dirección de correo de un destinatario usando el comando RCPT .
boolean completePendingCommand()	Este método se utiliza para finalizar la transacción y verificar el éxito o el fracaso de la respuesta del servidor.
boolean login()	Inicia sesión en el servidor SMTP enviando el comando HELO .
boolean login(String hostname)	Igual que la anterior, pero envía el nombre del host como argumento.

MÉTODOS	MISIÓN
boolean <code>logout()</code>	Finaliza la sesión con el servidor enviando el comando QUIT .
Writer <code>sendMessageData()</code>	Envía el comando DATA para después enviar el mensaje de correo. La clase Writer se usará para escribir secuencias de caracteres, como la cabecera y el cuerpo del mensaje.
boolean <code>sendShortMessageData(String message)</code>	Método útil para envío de mensajes cortos.
boolean <code>sendSimpleMessage(String remitente, String[] destinatarios, String message)</code>	Un método útil para el envío de un correo electrónico corto. Se especifica el remitente, los destinatarios y el mensaje.
boolean <code>sendSimpleMessage(String remitente, String destinatario, String message)</code>	Igual que el anterior pero el mensaje sólo va dirigido a un destinatario.
boolean <code>setSender(String address)</code>	Se especifica la dirección del remitente usando el comando MAIL .
boolean <code>verify(String username)</code>	Compruebe que un nombre de usuario o dirección de correo electrónico es válido (envía el comando VRFY para comprobarlo, tiene que estar soportado por el servidor).

Para enviar un simple mensaje a un destinatario podemos escribir las siguientes líneas:

```
// REALIZAR ACCIONES
client.login(); // inicio de sesión HELO
String destinatario = "mariajesusramos@brianda.net";
String mensaje = "Hola. \nEnviando saludos.\nChao.";
String remitente = "yo@localhost.es";

if (client.sendSimpleMessage(remitente, destinatario, mensaje))
    System.out.println("Mensaje enviado a " + destinatario);
else
    System.out.println("No se ha podido enviar");

client.logout();// final de sesión QUIT
```

Si todo va bien, se muestra un mensaje en consola:

```
220 localhost ESMTP server ready.
Mensaje enviado a mariajesusramos@brianda.net
```

Si consultamos el correo recibido vemos que aparece sin asunto, véase Figura 4.26.

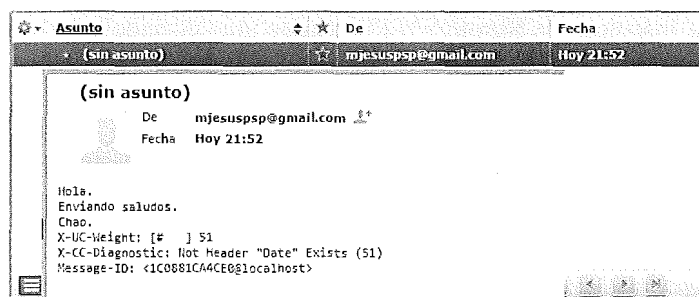


Figura 4.26. Recepción de correo sin asunto.

Cuando usamos MERCURY es importante consultar los mensajes que devuelve el servidor SMTP. Desde la ventana **Mercury SMTP Server** los podemos ver, y si se produce algún error aquí se mostrará. En la Figura 4.27 se muestra un mensaje de error (*554 Invalid HELO format*) y el envío correcto del mensaje anterior.

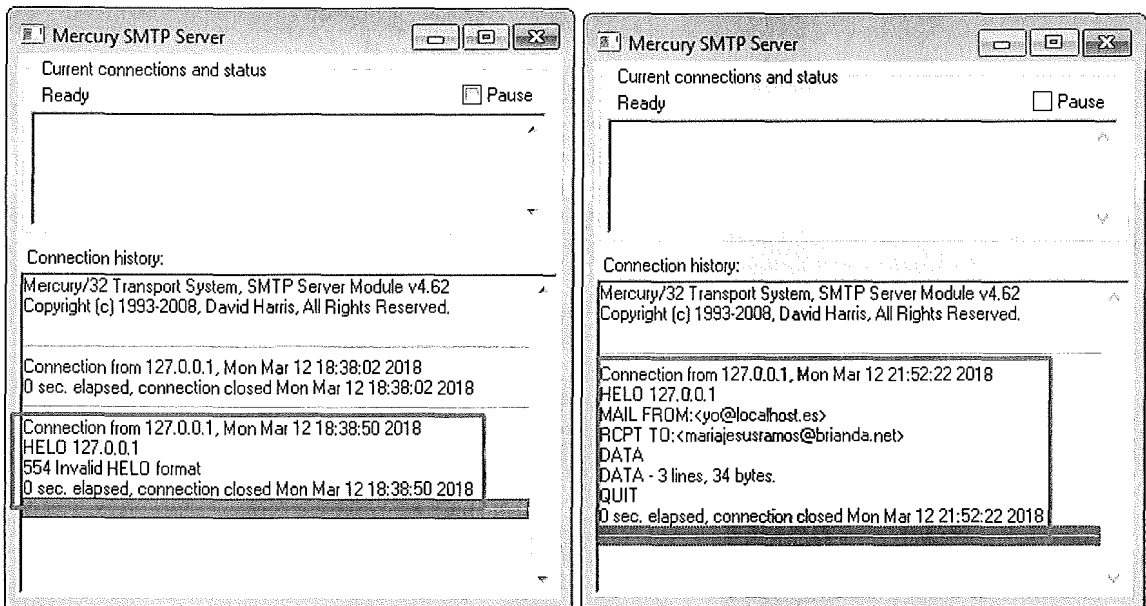


Figura 4.27. Mercury SMTP Server.

Para evitar el error *554 Invalid HELO format* hemos de modificar el fichero **transfltr.mer** que se encuentra en la carpeta `C:\xampp\MercuryMail`. Lo editamos con un editor de textos y buscamos la siguiente línea al final del archivo:

```
H, "[EHeh][EHeh]LO +[0-9]+.[0-9]+.[0-9]+.[0-9]*", R, "554 Invalid HELO format"
```

La comentamos agregando `#` al comienzo de la línea, guardamos los cambios y reiniciamos el servidor de correo. Si deseamos que MERCURY sea compatible con RFC, podemos agregar esta línea encima de la comentada: `H, "[EHeh] [EHeh] LO + [0-9] +. [0-9] +. [0-9] +. [0-9] +.", X, ""`.

La clase **SimpleSMTPHeader** se utiliza para la construcción de una cabecera mínima aceptable para el envío de un mensaje de correo electrónico. El constructor es el siguiente:

```
SimpleSMTPHeader(String from, String to, String subject)
```

Crea una nueva instancia de **SimpleSMTPHeader** inicializándola con los valores dados en los siguientes campos de cabecera:

- **from**: valor del campo de cabecera *from*, dirección de correo origen.
- **to**: valor del campo de cabecera *to*, dirección de correo destino.
- **subject**: valor del campo de cabecera *subject*, asunto del mensaje.

Proporciona los siguientes métodos:

MÉTODOS	MISIÓN
void addCC(String address)	Agrega una dirección de correo electrónico a la lista CC.
void addHeaderField(String headerField, String value)	Agrega un campo de encabezado arbitrario indicado en <i>headerField</i> , con el valor dado en <i>value</i> .
String toString()	Convierte el SimpleSMTPHeader a una cadena que contiene el encabezado con el formato correcto, incluyendo la línea en blanco para separar la cabecera del cuerpo del correo.

Desde el siguiente código se envía un correo a dos destinatarios (almacenados en las variables *destino1* y *destino2*), el texto se encuentra en la variable *mensaje*. Mediante la clase **SimpleSMTPHeader** se construye la cabecera y se agrega al campo CC el segundo destinatario. Mediante el método **setSender()** establecemos el remitente y mediante el método **addRecipient()** añadimos los destinatarios del mensaje, en este caso son dos (ejemplo en la clase *ClienteSMTP2.java*):

```

client.login(); //inicia sesión
String remitente = "yo@localhost.es";
String destino1 = "alumnouni5@gmail.com";
String destino2 = "mariajesusramos@brianda.net";
String asunto = "Prueba de SMTPClient";
String mensaje = "Hola. \nEnviando saludos.\nChao.";

//se crea la cabecera
SimpleSMTPHeader cabecera =
    new SimpleSMTPHeader(remitente, destino1, asunto);
cabecera.addCC(destino2);

//establecer el correo de origen
client.setSender(remitente);

//añadir correos de destino
client.addRecipient(destino1);
client.addRecipient(destino2);

```

Después se crea un objeto **Writer** para escribir el mensaje. Con el método **sendMessageData()** se envía el comando **DATA**, después se escribe la cabecera del correo y a continuación el cuerpo. Luego se cierra el stream. Por último se comprueba si el correo se ha enviado correctamente mediante el método **completePendingCommand()** y se cierra la sesión:

```

//se envia DATA
Writer writer = client.sendMessageData();
if(writer == null) { //fallo
    System.out.println("FALLO AL ENVIAR DATA.");
    System.exit(1);
}

//pintamos cabecera
System.out.println(cabecera.toString());

writer.write(cabecera.toString()); //primero escribo cabecera
writer.write(mensaje); //luego mensaje
writer.close(); //se cierra stream

if(!client.completePendingCommand()) { //fallo

```

```

        System.out.println("FALLO AL FINALIZAR LA TRANSACCIÓN.");
        System.exit(1);
    }
    client.logout();//Finaliza sesión

```

El programa muestra la siguiente salida en pantalla con la cabecera construida:

```

220 localhost ESMTP server ready.
Date: Tue, 13 Mar 2018 09:10:13 +0100
From: yo@localhost.es
To: alumnouni5@gmail.com
Cc: mariajesusramos@brianda.net
Subject: Prueba de SMTPClient

```

En la Figura 4.28 se muestran los correos recibidos por los dos destinos, se puede ver el aspecto de la cabecera del mensaje (si no se ven los correos pueden estar en la carpeta de correo spam):

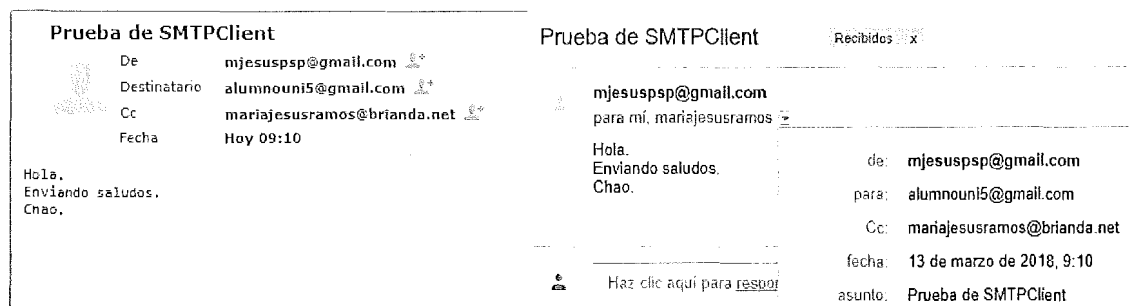


Figura 4.28. Recepción de correo en los dos destinos.

Hasta ahora hemos utilizado el servidor SMTP sin necesidad de autenticarnos. La **autenticación SMTP** se configura con el fin de elevar los niveles de seguridad y eficacia del servicio de correo electrónico y con el objetivo de prevenir que nuestra dirección de correo sea utilizada sin autorización, evitando el posible envío de correos no deseados a otras personas con fines perjudiciales. La autenticación se realiza a través de la verificación de su nombre de usuario y su contraseña.

Apache Commons Net™ proporciona la clase **AuthenticatingSMTPClient** (extiende **SMTPSClient**) con soporte de autenticación al conectarnos al servidor SMTP. La clase **SMTPSClient** proporciona soporte SMTP sobre el protocolo **SSL** (*Secure Socket Layer* – capa de conexión segura). **SSL** es un protocolo criptográfico empleado para realizar conexiones seguras entre un cliente y un servidor. **TLS** (*Transport Layer Security* - seguridad de la capa de transporte) es el protocolo sucesor de **SSL**.

STARTTLS es una extensión que nos permite cifrar las comunicaciones entre el cliente y el servidor, de tal forma que aunque se use el protocolo SMTP que por defecto no usa cifrado, todos los mensajes enviados y recibidos usando **STARTTLS** estarán cifrados. Cuando el servidor permite utilizar **STARTTLS**, se lo comunica al cliente y el cliente le responde para iniciar una sesión **TLS** y posteriormente intercambiar los correos electrónicos, que ya estarán cifrados en tránsito¹. Si nuestro servidor SMTP acepta **STARTTLS**, podemos estar seguros de que todo estará encriptado.

¹ <https://www.redeszone.net/2016/03/25/este-es-el-estado-del-despliegue-de-smtp-con-seguridad-starttls-segun-el-estudio-yahoo/>

El comando **STARTTLS** se incorporó como un nuevo mecanismo a SMTP a través del RFC 3207 para incorporar privacidad a la transmisión del correo.

La clase **SMTPSClient** (extiende **SMTPClient**) proporciona varios constructores. Usar uno u otro dependerá de los datos que nos proporcione el servidor SMTP en el que tengamos nuestra cuenta de correo:

- **SMTPSClient()**: establece el modo de seguridad en explícito, es decir *false*.
- **SMTPSClient (booleano implícito)**: establece el modo de seguridad en modo implícito si la variable booleana es *true* y en modo explícito si la variable booleana es *false*.

En modo implícito, la negociación **SSL/TLS** comienza justo después de que se haya establecido la conexión. En modo explícito (primer constructor), la negociación **SSL/TLS** se inicia cuando el usuario llama al método **execTLS()** y el servidor acepta el comando.

Por ejemplo, uso en modo implícito:

```
SMTPSClient c = new SMTPSClient(true);
c.connect("servidorsmtp", 25);
//resto de comandos aquí
```

Uso en modo explícito, por ejemplo el servidor **smtp.gmail.com**, que usa el puerto 587, soporta autenticación y **STARTTLS**:

```
SMTPSClient c = new SMTPSClient();
c.connect("smtp.gmail.com", 587);
if (c.execTLS()) {
    //resto de comandos aquí
}
```

Alguno de los métodos que usaremos después para realizar la autenticación SMTP son:

MÉTODOS	MISIÓN
void setKeyManager(KeyManager newKeyManager)	Para obtener el certificado para la autenticación se usa la interface KeyManager . Con este método se establece la clave para llevar a cabo la autenticación. Las KeyManager se pueden crear usando un KeyManagerFactory .
boolean execTLS()	Ejecuta el comando STARTTLS . La palabra clave STARTTLS se usa para indicarle al cliente SMTP que el servidor SMTP esta en disposición de negociar el uso de TLS . Devuelve <i>true</i> si el comando y la negociación han tenido éxito.

La **Autenticación SMTP (SMTP AUTH)** es una extensión del SMTP mediante el cual un cliente SMTP puede iniciar sesión utilizando un mecanismo de autenticación elegido entre los admitidos por el servidor SMTP. La clase **AuthenticatingSMTPClient** proporciona soporte de autenticación SMTP (RFC4954) al conectarnos al servidor SMTP.

Presenta varios constructores. Utilizaremos el constructor por defecto:

- **public AuthenticatingSMTPClient() throws NoSuchAlgorithmException**, para crear un nuevo cliente de autenticación SMTP. Puede lanzar *NoSuchAlgorithmException*, esta excepción se produce cuando un algoritmo criptográfico, habiéndose solicitado, no está disponible en el entorno.

Algunos de los métodos de la clase son los siguientes:

MÉTODOS	MISIÓN
boolean auth(AuthenticatingSMTPClient.AUTH_METHOD method, String username, String password)	Se envía el comando AUTH con el método seleccionado para autenticarse en el servidor SMTP, se envía el nombre del usuario y su clave.
int ehlo(String hostname)	Método para enviar el comando ESMTP (<i>Extended SMTP</i> - SMTP extendido) EHLO al servidor, devuelve el código de respuesta.

Los valores para el método de autenticación son: **CRAM_MD5** la contraseña se envía encriptada, **LOGIN** y **PLAIN** donde la contraseña se envía sin encriptar, como texto plano (no importa ya que la autenticación se va a realizar sobre un canal cifrado) y **XOAUTH** es un mecanismo de autenticación SASL que se basa en firmas OAuth.

En el siguiente ejemplo vamos a usar el servidor SMTP de Gmail. Los datos que proporciona son los siguientes:

- servidor de correo saliente (SMTP) - requiere TLS o SSL: **smtp.gmail.com**,
- puerto para TLS/STARTTLS: 587
- y puerto para SSL: 465.

En el ejemplo (*ClienteSMTP3.java*) se definen las variables con el nombre y la clave del usuario que se autentica en el servidor, el nombre del servidor y el puerto utilizado (en este caso el 587). Se crea una instancia de la clase **AuthenticatingSMTPClient** para crear el cliente SMTP seguro. Mediante la clase **KeyManagerFactory** creamos las **KeyManager** para establecer un canal de comunicación seguro (se incluyen los *import* necesarios para estas clases):

```
import java.io.IOException;
import java.io.Writer;
import java.security.InvalidKeyException;
import java.security.KeyStoreException;
import java.security.NoSuchAlgorithmException;
import java.security.UnrecoverableKeyException;
import java.security.spec.InvalidKeySpecException;
import javax.net.ssl.KeyManager;
import javax.net.ssl.KeyManagerFactory;
import org.apache.commons.net.smtp.*;

public class ClienteSMTP3 {
    public static void main(String[] args) throws
        NoSuchAlgorithmException, UnrecoverableKeyException,
        KeyStoreException, InvalidKeyException, InvalidKeySpecException {

        //se crea cliente SMTP seguro
        AuthenticatingSMTPClient client = new AuthenticatingSMTPClient();

        //datos del usuario y del servidor
        String server = "smtp.gmail.com";
        String username = "correo@gmail.com";
        String password = "clavedelusuario";
        int puerto = 587;
        String remitente = "correo@gmail.com";
```



```

try {
    int respuesta;

    // Creación de la clave para establecer un canal seguro
    KeyManagerFactory kmf = KeyManagerFactory
        .getInstance(KeyManagerFactory.getDefaultAlgorithm());
    kmf.init(null, null);
    KeyManager km = kmf.getKeyManagers()[0];

```

A continuación se utiliza el método **connect()** para realizar la conexión al servidor SMTP y se establece la clave para la comunicación segura. Antes de continuar se comprueba el código de respuesta generado:

```

//nos conectamos al servidor SMTP
client.connect(server, puerto);
System.out.println("1 - " + client.getReplyString());
//se establece la clave para la comunicación segura
client.setKeyManager(km);

respuesta = client.getReplyCode();
if (!SMTPReply.isPositiveCompletion(respuesta)) {
    client.disconnect();
    System.err.println("CONEXIÓN RECHAZADA.");
    System.exit(1);
}

```

Se envía el comando **EHLO** mediante el método **ehlo()** y se ejecuta el método **execTLS()**. Si tiene éxito se realiza todo el proceso empezando por la autenticación del usuario mediante el método **auth()**. Como método de autenticación se usa **LOGIN** (*AuthenticatingSMTPClient.AUTH_METHOD.LOGIN*). Después se preparan las cabeceras y el texto del mensaje, se añaden el emisor y destinatario del mensaje y se escribe en el **Writer**:

```

//se envía el commando EHLO
client.ehlo(server); // necesario
System.out.println("2 - " + client.getReplyString());

//NECESITA NEGOCIACIÓN TLS - MODO NO IMPLICITO
//Se ejecuta el comando STARTTLS y se comprueba si es true
if (client.execTLS()) {
    System.out.println("3 - " + client.getReplyString());

    //se realiza la autenticación con el servidor
    if (client.auth(AuthenticatingSMTPClient.AUTH_METHOD.PLAIN,
        username, password)) {
        System.out.println("4 - " + client.getReplyString());
        String destino1 = "mariajesusramos@brianda.net";
        String asunto = "Prueba de SMTPClient con GMAIL";
        String mensaje = "Hola. \nEnviando saludos.\nUsando\nGMAIL.\nChao.";

        //se crea la cabecera
        SimpleSMTPHeader cabecera = new SimpleSMTPHeader(remitente,
            destino1, asunto);
        //el nombre de usuario y el email de origen coinciden
        client.setSender(remitente);
        client.addRecipient(destino1);
        System.out.println("5 - " + client.getReplyString());

        //se envia DATA

```

```

Writer writer = client.sendMessageData();
if (writer == null) { // fallo
    System.out.println("FALLO AL ENVIAR DATA.");
    System.exit(1);
}

writer.write(cabecera.toString()); //cabecera
writer.write(mensaje); //luego mensaje
writer.close();
System.out.println("6 - " + client.getReplyString());

boolean exito = client.completePendingCommand();
System.out.println("7 - " + client.getReplyString());

if (!exito) { // fallo
    System.out.println("FALLO AL FINALIZAR TRANSACCIÓN.");
    System.exit(1);
} else
    System.out.println("MENSAJE ENVIADO CON ÉXITO.....");

} else
    System.out.println("USUARIO NO AUTENTICADO.");
} else
    System.out.println("FALLO AL EJECUTAR STARTTLS.");

} catch (IOException e) {
    System.err.println("Could not connect to server.");
    e.printStackTrace();
    System.exit(1);
}
try {
    client.disconnect();
} catch (IOException f) {f.printStackTrace();}

System.out.println("Fin de envío.");
System.exit(0);
} //main
} // ..ClienteSMTP3

```

La salida generada al ejecutar el programa mostrando las respuestas que envía el servidor es la siguiente:

```

1 - 220 smtp.gmail.com ESMTP 2sm73594wmk.29 - gsmt
2 - 250-smtp.gmail.com at your service, [2.153.35.64]
250-SIZE 35882577
250-8BITMIME
250-STARTTLS
250-ENHANCEDSTATUSCODES
250-PIPELINING
250-CHUNKING
250 SMTPUTF8

3 - 220 2.0.0 Ready to start TLS

4 - 235 2.7.0 Accepted

5 - 250 2.1.5 OK 2sm73594wmk.29 - gsmt

```

```
6 - 354 Go ahead 2sm73594wmk.29 - gsmtpp
7 - 250 2.0.0 OK 1520942797 2sm73594wmk.29 - gsmtpp
```

MENSAJE ENVIADO CON ÉXITO.....
Fin de envío.

Algunos códigos de respuesta son: 220: indica que el servicio está preparado, 250: solicitud aceptada, 235: autenticación aceptada, 354: acepta la entrada de datos.

Si el servidor SMTP no requiere negociación con el comando TLS hay que quitar el **if (client.execTLS())**, se realizarían las operaciones de dentro del if sin llamar a ese método. Sin negociación TLS se muestra la siguiente salida:

```
1 - 220 smtp.jccm.es ESMTP
2 - 250-smtp.jccm.es
250-8BITMIME
250-SIZE 31457280
250-AUTH PLAIN LOGIN
250 AUTH=PLAIN LOGIN
4 - 235 #2.0.0 OK Authenticated
5 - 250 recipient <mariajesusramos@brianda.net> ok
6 - 354 go ahead
7 - 250 ok: Message 1782861 accepted
```

MENSAJE ENVIADO CON ÉXITO.....
Fin de envío.

ACTIVIDAD 4.4

Realiza un programa Java que permita a cualquiera que lo ejecute enviar un correo electrónico. Todos los datos que se necesitan se introducirán por teclado: el servidor SMTP, si necesita negociación TLS, el puerto, nombre de usuario, su password, el correo del remitente (no siempre coincide con el nombre de usuario) y destinatario, asunto y el mensaje, que permitirá varias líneas y cuando se introduzca un * finalizará el texto del mensaje:

```
Introduce servidor SMTP.....:
Necesita negociación TLS (S, N)?:
Introduce usuario.....:
Introduce password.....:
Introduce puerto.....:
Introduce correo del remitente..:
Introduce correo destinatario...:
Introduce asunto.....:
Introduce el mensaje, finalizará cuando se introduzca un * :
```

Se deben mostrar los mensajes de cada acción con el servidor SMTP. Se debe mostrar un mensaje indicando si la negociación se ha establecido o no. No se puede enviar un mensaje vacío. Mostrar más mensajes que consideréis necesarios para aclarar la situación.

Realiza el ejercicio propuesto 2.

4.4.4. ACCESO A LOS MENSAJES DE UN SERVIDOR SMTP

En el correo electrónico se utilizan otros protocolos, además del SMTP, para funciones adicionales. Entre los más utilizados están:

- **MIME** (*Multipurpose Internet Mail Extensions* - extensiones multipropósito de correo de internet): define una serie de especificaciones para expandir las capacidades limitadas del correo electrónico y en particular para permitir la inserción de documentos (como imágenes, sonido y texto) en un mensaje. Su versión segura se denomina **S/MIME**.
- **POP** (*Post Office Protocol* - protocolo de oficina de correo): proporciona acceso a los mensajes de los servidores SMTP. En general, cuando hacemos referencia al término **POP**, nos estamos refiriendo a **POP3** que es la última versión. Este es el que usaremos en este apartado.
- **IMAP** (*Internet Message Access Protocol* - protocolo de acceso a mensajes de Internet): permite acceder a los mensajes de correo electrónico almacenados en los servidores SMTP. Permite que los usuarios accedan a su correo desde cualquier equipo que tenga una conexión a Internet. Tiene alguna ventaja sobre POP, por ejemplo, los mensajes continúan siempre almacenados en el servidor, cosa que no ocurre con POP o los usuarios pueden organizar los mensajes en carpetas. La versión actual es la 4, **IMAP4**.

A POP3 se le asigna el puerto 110. El Servidor POP3 es el servidor de correo electrónico entrante y utiliza en general el puerto 110. Al igual que SMTP, funciona con comandos de texto. Algunos de ellos se muestran en esta tabla:

COMANDOS	MISIÓN
USER login	A la derecha se escribe el login de la cuenta de correo.
PASS contraseña	A la derecha se escribe la contraseña de la cuenta de correo.
STAT	Muestra el número de mensajes de la cuenta.
LIST	Listado de mensajes (numero - tamaño total del mensaje).
RETR número-mensaje	Obtiene el mensaje cuyo número coincida con el indicado a la derecha.
DELE número-mensaje	Borra el mensaje indicado.
QUIT	Cierra la conexión.
TOP número-mensaje n	Muestra las 'n' primeras líneas del mensaje indicado.

4.4.5. USO DE TELNET PARA COMUNICAR CON EL SERVIDOR POP

Podemos hacer Telnet al puerto 110 para interactuar con el servidor POP. Para probarlo vamos a crear un usuario en nuestro servidor local de correo. Desde la opción de menú **Configuration/ Manage local users** pulsamos el botón **Add** y a continuación rellenamos los datos del usuario, véase Figura 4.29. En el ejemplo se crea el usuario de nombre *usu1* y clave igual.

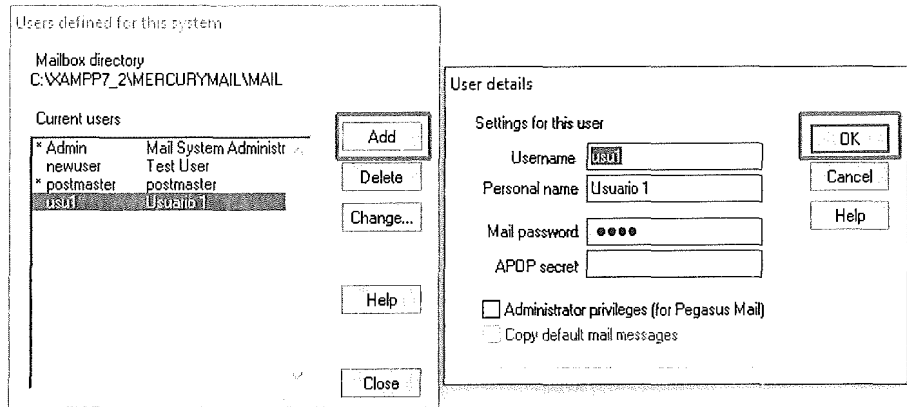


Figura 4.29. Creación de un usuario de correo.

Desde la opción de menú **Configuration/ MercuryP POP3 Server** desmarcamos las casillas *Mark successfully-retrieved mail as Read* y *Offer only unread mail to connected clients*, véase Figura 4.30, y pulsamos el botón *Aceptar*. Esto hará que los mensajes no desaparezcan del buzón una vez recuperados, así podremos recuperarlos siempre que queramos.

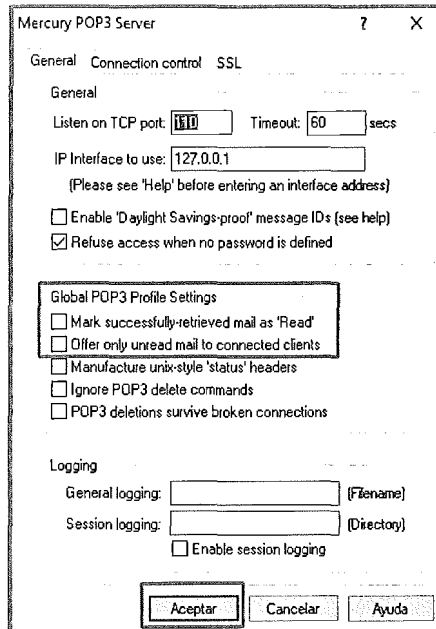


Figura 4.30. Mercury POP3 Server.

Ahora usando los comandos SMTP por medio de Telnet al puerto 25, enviamos varios mensajes al usuario creado *usu1*, por ejemplo:

```
telnet localhost 25 ↵
220 localhost ESMTP server ready.
HELO ↵
250 localhost Hello, .
MAIL FROM: yo@localhost.es ↵
250 Sender OK - send RCPTs.
RCPT TO: <usu1> ↵
250 Recipient OK - send RCPT or DATA.
DATA ↵
```

222 Programación de servicios y procesos

```
354 OK, send data, end with CRLF.CRLF
Subject: Probando ↵
```

```
Esto es una prueba de un correo ↵
de varias líneas ↵
enviado desde Telnet. ↵
```

```
Fin. ↵
. ↵
250 Data received OK.
QUIT ↵
221 localhost Service closing channel.
```

Ahora usando los comandos **POP3** por medio de Telnet al puerto 110, recuperamos uno de los mensajes enviados a *usul*. En primer lugar, se envía el comando **USER** con el nombre de usuario, a continuación, se envía **PASS** con la clave. El comando **STAT** nos muestra el número de mensajes del usuario y el tamaño. El comando **LIST** nos muestra la lista de mensajes (en este caso hay 2 mensajes), el comando **RETR 2** obtiene el mensaje con número 2:

```
telnet localhost 110 ↵
+OK <556829734.16351@localhost>, POP3 server ready.
USER usul ↵
+OK usul is known here.
PASS usul ↵
+OK Welcome! 2 messages (35661 bytes)
STAT ↵
+OK 2 35660
LIST ↵
+OK 2 messages, 35661 bytes
1 35243
2 418
.
RETR 2 ↵
+OK Here it comes...
Received: from spooler by localhost (Mercury/32 v4.62); 13 Mar 2018
21:54:12 +0100
X-Envelope-To: <usul>
Return-path: yo@localhost.es
Received: from (127.0.0.1) by localhost (Mercury/32 v4.62) ID
MG000001;
    13 Mar 2018 21:53:18 +0100
Subject: Probando
X-UC-Weight: [#    ] 51
X-CC-Diagnostic: Not Header "Date" Exists (51)
```

```
Esto es una prueba de un correo
de varias líneas
enviado desde Telnet.
```

```
Fin.
.
QUIT ↵
+OK localhost Server closing down.
```

4.4.6. JAVA PARA COMUNICAR CON UN SERVIDOR POP3

Apache Commons Net™ proporciona varias clases para acceder a servidores POP3:

- La clase **POP3Client**: implementa el lado cliente del protocolo POP3 de Internet definido en la RFC 1939.
- **POP3SClient**: POP3 con soporte SSL, extiende **POP3Client**.
- **POP3MessageInfo**: se utiliza para devolver información acerca de los mensajes almacenados en el servidor POP3.

La clase **POP3Client** presenta un único constructor. Algunos de los métodos que usaremos en los ejemplos son:

MÉTODOS	MISIÓN
boolean deleteMessage(int messageId)	Elimina el mensaje con número <i>messageId</i> del servidor POP3. Devuelve <i>true</i> si la operación se realizó correctamente.
POP3MessageInfo listMessage(int messageId)	Lista el mensaje indicado en el parámetro <i>messageId</i> .
POP3MessageInfo[] listMessages()	Obtiene un array con información de todos los mensajes.
POP3MessageInfo listUniqueIdentifier(int messageId)	Obtiene la lista de un único mensaje.
boolean login(String username, String password)	Inicia sesión en el servidor POP3 enviando el nombre de usuario y la clave. Devuelve <i>true</i> si la operación se realizó correctamente.
boolean logout()	Finaliza la sesión con el servidor POP3. Devuelve <i>true</i> si la operación se realizó correctamente.
Reader retrieveMessage(int messageId)	Recupera el mensaje con número <i>messageId</i> del servidor POP3.
Reader retrieveMessageTop(int messageId, int numLines)	Igual que el anterior pero sólo el número de líneas especificadas en el parámetro <i>numLines</i> . Para recuperar la cabecera del mensaje <i>numLines</i> debe ser 0.

La clase **POP3SClient** presenta varios constructores, usar uno u otro dependerá de los datos que nos proporcione el servidor POP en el que tengamos nuestra cuenta de correo. En los ejemplos usaremos los más básicos:

- el constructor sin parámetros **POP3SClient()**, modo explícito y
- el constructor con un parámetro booleano **POP3SClient(boolean implicit)**.

Si se selecciona el segundo constructor con el parámetro a *true* (modo implícito), la negociación SSL/TLS comienza justo después de que se haya establecido la conexión. En modo explícito (primer constructor), la negociación SSL/TLS se inicia cuando el usuario llama al método **execTLS()** y el servidor acepta el comando (como en **SMTPLSClient**). Aunque a veces no es necesario ejecutar el método **execTLS()**.

Por ejemplo, uso en modo implícito:

```
POP3SClient c = new POP3SClient(true);
c.connect(server, 995);
//resto de comandos aquí
```

Uso en modo explícito:

```
POP3SClient c = new POP3SClient();
c.connect(server, 110);
if (c.execTLS()) {
    //resto de comandos aquí
}
```

Los métodos de la clase **POP3SClient** son similares a los vistos para la clase **SMTPSClient**.

En el siguiente ejemplo nos conectamos al servidor POP3 local y visualizamos el número de mensajes del usuario *usu1*, utilizamos el constructor sin parámetros (modo explícito) pero no se usa el método **execTLS()** ya que no se necesita negociar el uso de TLS:

```
import java.io.IOException;
import org.apache.commons.net.pop3.POP3MessageInfo;
import org.apache.commons.net.pop3.POP3SClient;

public class Ejemplo1POP3 {
    public static void main(String[] args) {
        String server = "localhost", username = "usu1",
            password = "usu1";
        int puerto = 110;

        POP3SClient pop3 = new POP3SClient();
        try {
            //nos conectamos al servidor
            pop3.connect(server, puerto);
            System.out.println("Conexión realizada al servidor
                               POP3 " + server);

            //iniciamos sesión
            if (!pop3.login(username, password))
                System.err.println("Error al hacer login");
            else{
                //obtenemos todos los mensajes en un array
                POP3MessageInfo[] men = pop3.listMessages();

                if (messages == null)
                    System.out.println("Imposible recuperar mensajes.");
                else
                    System.out.println("Nº de mensajes: " + men.length);

                //finalizar sesión
                pop3.logout();
            }

            //nos desconectamos
            pop3.disconnect();

        } catch (IOException e) {
            System.err.println(e.getMessage());
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```



```

    }
    System.exit(0);
} // main

} // ..Ejemplo1POP3

```

La ejecución muestra la siguiente información:

```

Conexión realizada al servidor POP3 localhost
Nº de mensajes: 2

```

POP3MessageInfo se utiliza para devolver información acerca de los mensajes almacenados en el servidor POP3. Sus campos (*identifier*, *number* y *size*) se utilizan para referirse a cosas ligeramente diferentes dependiendo de la información que se devuelve:

- En respuesta a un comando de estado, *number* contiene el número de mensajes en el buzón de correo, *size* contiene el tamaño del buzón de correo en bytes, y el campo *identifier* es nulo.
- En respuesta a una lista de mensajes, *number* contiene el número de mensaje, *size* contiene el tamaño del mensaje en bytes, e *identifier* es nulo.
- En respuesta a una lista de un único mensaje, *number* contiene el número de mensaje, *size* no está definido, e *identifier* contiene el identificador único del mensaje.

En el siguiente método se recorre el array de mensajes y se visualiza información de los campos anteriores (*identifier*, *number* y *size*), se puede ver como varían sus valores al usar el array con la lista de mensajes y al usar el método **listUniqueIdentifier()**:

```

private static void Recuperarmensajes
    (POP3MessageInfo[] men, POP3SClient pop3) throws IOException {

    for (int i=0; i< men.length; i++) {
        System.out.println("Mensaje: " + (i+1));
        POP3MessageInfo msginfo = men[i]; //lista de mensajes
        System.out.println("IDentificador: " + msginfo.identifier +
            ", Number: " + msginfo.number + ", Tamaño: " + msginfo.size);

        System.out.println("Prueba de listUniqueIdentifier: ");
        POP3MessageInfo pmi = pop3.listUniqueIdentifier(i+1); //un mensaje
        System.out.println("\tIDentificador: " + pmi.identifier +
            ", Number: " + pmi.number + ", Tamaño: " + pmi.size);
    } //for

} //Recuperarmensajes

```

Se visualiza:

```

Conexión realizada al servidor POP3 localhost
Nº de mensajes: 2
Mensaje: 1
IDentificador: null, Number: 1, Tamaño: 35243
Prueba de listUniqueIdentifier:
    IDentificador: 7364971.CNM4C6D621B, Number: 1, Tamaño: -1
Mensaje: 2
IDentificador: null, Number: 2, Tamaño: 418
Prueba de listUniqueIdentifier:
    IDentificador: 5CLL90K.CNM4C6D625A, Number: 2, Tamaño: -1

```

En el siguiente método se muestran las cabeceras de los mensajes recogidos del servidor POP local, se realiza mediante el método **retrieveMessageTop(msginfo.number, 0)**, pasándole como primer parámetro el número de mensaje y como segundo el valor 0:

```
private static void Recuperarcabeceras
(POP3MessageInfo[] men, POP3SClient pop3) throws IOException {

    for (int i=0; i< men.length; i++) {
        System.out.println("Mensaje: " + (i+1));
        POP3MessageInfo msginfo = men[i]; //lista de mensajes

        //solo recupera cabecera
        System.out.println("Cabecera del mensaje:");
        BufferedReader reader = (BufferedReader)
            pop3.retrieveMessageTop(msginfo.number, 0);
        String linea;
        while ((linea = reader.readLine()) != null)
            System.out.println(linea.toString());
        reader.close();

    } //for

} //Recuperarcabeceras
```

Devuelve la siguiente salida:

```
Conexión realizada al servidor POP3 localhost
Nº de mensajes: 2
Mensaje: 1
Cabecera del mensaje:
Received: from spooler by persephone.pmail.gen.nz (Mercury/32 v4.62); 8
Aug 2008 15:26:19 +1200
X-Envelope-To: David.harris@pmail.gen.nz
From: "David Harris" <David.Harris@pmail.gen.nz>
Organization: Pegasus Mail, Dunedin, New Zealand
To: David.harris@pmail.gen.nz
Date: Thu, 27 Jan 2011 15:00:00 +1200
MIME-Version: 1.0
Subject: Welcome to Pegasus Mail!
Message-ID: <489BBDAC.24975.43E28F31@David.Harris.pmail.gen.nz>
Priority: normal
X-mailer: Pegasus Mail for Windows (4.61 wb1)
Content-type: Multipart/Related; boundary="Message-Boundary-20828"
X-PMFLAGS: 573075584 0 1 Y6FSIJKV.CNM

Mensaje: 2
Cabecera del mensaje:
Received: from spooler by localhost (Mercury/32 v4.62); 13 Mar 2018
21:54:12 +0100
X-Envelope-To: <usul>
Return-path: yo@localhost.es
Received: from (127.0.0.1) by localhost (Mercury/32 v4.62) ID
MG000001;
    13 Mar 2018 21:53:18 +0100
Subject: Probando
X-UC-Weight: [#    ] 51
X-CC-Diagnostic: Not Header "Date" Exists (51)
```

Para recuperar todo el mensaje usamos el método `retrieveMessage(int messageId)` pasando el número de mensaje. En el ejemplo anterior cambiaríamos la definición del `reader`:

```
BufferedReader reader =  
    (BufferedReader) pop3.retrieveMessage(msginfo.number);
```

Para probarlo con una cuenta de Gmail necesitaríamos el segundo constructor (modo implícito). Los datos que nos proporciona son los siguientes:

- Servidor de correo entrante (POP3): **pop.gmail.com**,
- utilizar SSL: Sí,
- puerto: 995.

Para usar el servidor POP de Gmail escribo lo siguiente:

```
POP3SClient pop3 = new POP3SClient(true);  
pop3.connect("pop.gmail.com", 995);
```

ACTIVIDAD 4.5

Modifica los programas anteriores accediendo al servidor POP local añadiendo los mensajes que devuelve usando el método `getReplyString()`.

Accede a tu cuenta de correo externa ya sea de gmail o de otro proveedor y visualiza el número de mensajes que tienes y el cuerpo de los mensajes. Probablemente tendrás que configurar alguna opción en la cuenta para poder acceder a los mensajes siempre que quieras.

Realiza el Ejercicio 3.

4.5. PROGRAMACIÓN DE SERVIDORES CON JAVA

A continuación, vamos a construir un servidor de ficheros, su misión será proporcionar el acceso a ficheros y carpetas. Cuando se inicia el programa servidor se elige la carpeta o directorio de nuestro disco duro que se ofrecerá a los clientes que se conecten. Los clientes se conectarán al servidor y sólo podrán cargar y descargar ficheros, no podrán navegar a través de los directorios. La comunicación entre el cliente y el servidor se realizará mediante intercambio de objetos. El flujo de operaciones es el siguiente (Figura 4.31):

- Desde el programa servidor se elige la carpeta o directorio a la que los clientes podrán acceder.
- Se inicia el servidor en un puerto pactado, que el programa cliente debe conocer.
- Cuando un cliente se conecta se crea un objeto **EstructuraFicheros** con la información del directorio elegido.
- El servidor envía al cliente un objeto **EstructuraFicheros** nada más conectarse.
- Cuando el cliente solicita descargar un fichero realiza la petición mediante un objeto **PideFichero** con el nombre del fichero a descargar. El servidor recibe la petición y en respuesta le envía un objeto **ObtieneFichero** con los bytes y el tamaño del fichero solicitado.
- Cuando el cliente solicita cargar un fichero en el servidor, la petición se realiza mediante un objeto **EnviaFichero** que contiene el nombre del fichero, su contenido en bytes y su tamaño. El servidor acepta la petición, crea el nuevo fichero en su directorio y en respuesta le envía un objeto **EstructuraFicheros** con la estructura de ficheros actualizada para el directorio en el que se hizo la carga del fichero.

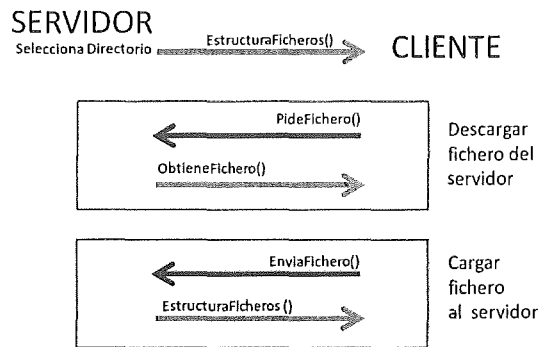


Figura 4.31. Comunicación entre el cliente y el servidor.

La clase **EstructuraFicheros** se utiliza para definir la estructura del directorio seleccionado con sus ficheros y directorios. Se definen los siguientes atributos:

- **name**: nombre del directorio,
- **path**: nombre completo,
- **isDir**: indica si es un directorio o no,
- **numeFich**: número de ficheros que tiene el directorio y
- **lista**: es un array de objetos **EstructuraFicheros** que contiene los ficheros y directorios del directorio seleccionado.

Se definen los métodos *get* para obtener el valor de los atributos anteriores y dos constructores. El primero se usa en el programa servidor para cargar la información del directorio seleccionado y el segundo se usa dentro de la clase, en el método *getListaFiles()* para ir cargando en el array la información de cada fichero o directorio del directorio seleccionado. El código es el siguiente:

```
import java.io.*;
public class EstructuraFicheros implements Serializable {
    private String name; // nombre del directorio
    private String path; // nombre completo (directorio+nombre)
    private boolean isDir; // es un directorio?
    private int numeFich; // número de fich del directorio
    private EstructuraFicheros[] lista; // lista de ficheros y carpetas

    //Primer constructor, se llama desde el programa servidor
    public EstructuraFicheros(String name) throws FileNotFoundException {
        File file = new File(name);
        this.name = file.getName();
        this.path = file.getPath();
        this.isDir = file.isDirectory();
        this.lista = getListaFiles(); //Se carga la lista de fic y direc

        if (file.isDirectory()) {
            File[] ficheros = file.listFiles();
            this.numeFich = 0;
            if (!(ficheros == null)) this.numeFich = ficheros.length;
        }
    }

    //Segundo constructor
    public EstructuraFicheros(String name, String path, boolean isDir,
                              int numF) {
```

```

        this.name = name;
        this.path = path;
        this.isDir = isDir;
        this.numFich = numF;
    }
    //Métodos para obtener valores de los atributos
    public int getNumFich()          { return numFich; }
    public boolean isDir()           { return isDir; }
    public String getPath()          { return path; }
    public EstructuraFicheros[] getList() { return lista; }

    public String getName() {
        String name_dir = name;
        if (isDir) {
            // Si es un directorio, me quedo solo con su nombre
            int l = path.lastIndexOf(File.separator);
            name_dir = path.substring(l + 1, path.length());
        }
        return name_dir;
    }

    //Se sobrescribe para asociar a todo objeto un texto representativo
    //Si es un fichero se antepone al nombre del mismo la palabra DIR
    public String toString() {
        String nom = this.name;
        if (this.isDir) nom = "(DIR) " + name;
        return nom;
    }

    //Método interno para llenar el array con los ficheros y directorios
    //del directorio seleccionado. Cada elemento del array (un fichero o
    //un directorio) es de tipo EstructuraFicheros
    EstructuraFicheros[] getListFiles() {
        EstructuraFicheros[] lista = null;
        String sDirectorio = this.path;
        File f = new File(sDirectorio);
        File[] ficheros = f.listFiles(); //ficheros del directorio
        int longitud = ficheros.length; //nº de ficheros del directorio

        if (longitud > 0) { //si está vacío no se llena la lista
            lista = new EstructuraFicheros[longitud];
            //se recorre el array de ficheros para llenar la lista

            for (int x = 0; x < ficheros.length; x++) {
                EstructuraFicheros elemento;
                String nombre = ficheros[x].getName();
                String path = ficheros[x].getPath();
                boolean isDir = ficheros[x].isDirectory();
                int num = 0;
                if (isDir) {
                    //calculamos el número de ficheros
                    File[] fic = ficheros[x].listFiles();
                    if (!(fic == null)) num = fic.length;
                }
                elemento = new EstructuraFicheros(nombre, path, isDir, num);
                lista[x] = elemento; //se va llenando la lista
            } // for
        }
    }

```

```

    }//if

    return lista; //se devuelve la lista
} // getListaFiles

} // ..EstructuraFicheros

```

4.5.1. PROGRAMA SERVIDOR

El programa servidor, clase **Servidor**, es muy básico, se define el puerto, se elige el directorio con **JFileChooser**, se inicia el servidor y se entra en un proceso repetitivo donde se esperan las conexiones de los clientes, a cada cliente conectado se le envía el socket creado y un objeto **EstructuraFicheros(Directorio seleccionado)**.

La Figura 4.32 muestra un momento de la ejecución del programa servidor en el que se elige el directorio. En el ejemplo la ejecución del servidor se realiza en una máquina con sistema operativo Ubuntu y se elige el directorio *MisDocs*.

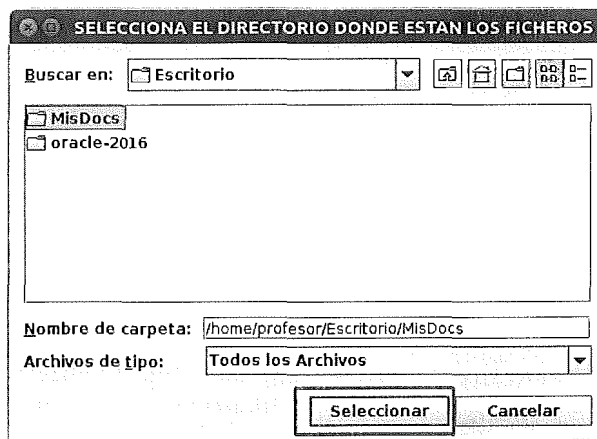


Figura 4.32. Ejecución del servidor.

```

import java.io.*;
import java.net.*;
import javax.swing.*;

public class Servidor {
    static Integer PUERTO = 44441;
    static public EstructuraFicheros NF;
    static ServerSocket servidor;

    public static void main(String[] args) throws IOException {
        String Directorio = "";
        JFileChooser f = new JFileChooser();
        f.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
        f.setDialogTitle("SELECCIONA EL DIRECTORIO DONDE  
ESTÁN LOS FICHEROS");
        int returnVal = f.showDialog(f, "Seleccionar");

        if (returnVal == JFileChooser.APPROVE_OPTION) {
            File file = f.getSelectedFile();
            Directorio = file.getAbsolutePath();
            System.out.println(Directorio);
        }

        //si no se selecciona nada salir

```

```

if(Directorio.equals("")){
    System.out.println("Debe seleccionar un directorio.");
    System.exit(1);
}

servidor = new ServerSocket(PUERTO);
System.out.println("Servidor Iniciado en Puerto: "+PUERTO);

while (true) {
    try {
        Socket cliente = servidor.accept();
        System.out.println("Bienvenido al cliente");
        NF = new EstructuraFicheros(Directorio);
        HiloServidor hilo = new HiloServidor(cliente, NF);
        hilo.start(); // Ejecutamos el hilo
    } catch (IOException e) {
        System.out.println(e.getMessage());
        System.exit(0);
    }
}
} //main
} // ..fin Servidor

```

Las operaciones por cada cliente conectado se realizan en la clase **HiloServidor**. En el constructor se definen los stream de entrada y de salida a partir del socket enviado por el servidor. El intercambio de objetos entre cliente y servidor se realizan en el método **run()** del hilo. El código es el siguiente:

```

import java.io.*;
import java.net.*;

class HiloServidor extends Thread {
    Socket socket;
    ObjectOutputStream outObjeto; //stream de salida
    ObjectInputStream inObjeto;    //stream de entrada
    EstructuraFicheros NF;

    //constructor
    public HiloServidor(Socket s, EstructuraFicheros nF)
        throws IOException {
        socket = s;
        NF = nF;
        inObjeto = new ObjectInputStream(socket.getInputStream());
        outObjeto = new ObjectOutputStream(socket.getOutputStream());
    }

    public void run() {
        try {
            //envio al cliente el objeto EstructuraFicheros
            outObjeto.writeObject(NF);

            while (true) {
                // obtengo lo que me pide cliente
                Object peticion = inObjeto.readObject();

                //compruebo que es lo que quiere
                if (peticion instanceof PideFichero) {
                    //el cliente pide un fichero al servidor

```

```

        PideFichero fichero = (PideFichero) peticion;
        EnviarFichero(fichero);
    }

    if (peticion instanceof EnviaFichero) {
        //el cliente envía un fichero para cargarlo
        //en el servidor
        EnviaFichero fic = (EnviaFichero) peticion;
        File d = new File(fic.getDirectorio());
        File fl = new File(d,fic.getNombre());

        //se crea el fichero en el directorio
        //con los bytes enviados en el objeto
        FileOutputStream fos = new FileOutputStream(fl);
        fos.write(fic.getContenidoFichero());
        fos.close();

        //se crea la nueva estructura de directorios
        EstructuraFicheros n = new
            EstructuraFicheros(fic.getDirectorio());

        outObjeto.writeObject(n); //se envia al cliente
    }
} //while

} catch (IOException e) {
    // cuando un cliente Cierra la conexion
    try {
        inObjeto.close();
        outObjeto.close();
        socket.close();
        System.out.println("Cerrando cliente");
    } catch (IOException ee){ ee.printStackTrace(); }
} catch (ClassNotFoundException e) { e.printStackTrace(); }

} //run()

```

En el método **run()** primero se envía al cliente el objeto **EstructuraFicheros** y después en un proceso repetitivo (*while (true)*) se analizan las peticiones del cliente. Las peticiones llegan como objetos, se comprobará si el objeto es una instancia de una clase determinada mediante el operador **instanceof**. El proceso repetitivo terminará cuando un cliente cierre la conexión al servidor, se producirá entonces una excepción **IOException**.

Si el objeto es **PideFichero**, *if (peticion instanceof PideFichero)*, es que el cliente solicita un fichero del servidor, la acción se lleva a cabo en el método **EnviarFichero(fichero)**. El método recibe un objeto **PideFichero** con el nombre del fichero solicitado por el cliente. Se van leyendo los bytes del fichero solicitado y se van almacenando en un array de bytes que después se enviará al cliente mediante el objeto **ObtieneFichero**. El código es el siguiente:

```

//este método envía al cliente el fichero solicitado
private void EnviarFichero(PideFichero fich) {
    //se obtiene fichero
    File fichero = new File(fich.getNombreFichero());
    FileInputStream filein = null;
    try {
        //se abre el fichero en el servidor
        filein = new FileInputStream(fichero);
    }
}

```



```

        long bytes = fichero.length(); //tamaño del fichero
        byte[] buff = new byte[(int) bytes];
        int i, j = 0;

        //se van leyendo los bytes del fichero y llenando el array
        while ((i = filein.read()) != -1) { // lectura de bytes
            buff[j] = (byte) i; //se almacenan en el array
            j++;
        }
        filein.close(); // cerrar fichero
        Object ff = new ObtieneFichero(buff);

        //envia objeto ObtieneFichero con los bytes del fichero
        outObjeto.writeObject(ff);
    } catch (FileNotFoundException e) { e.printStackTrace(); }
    catch (IOException e) { e.printStackTrace(); }

} //EnviarFichero

} // ..HiloServidor

```

La clase **PideFichero** define un atributo con el nombre del fichero que el cliente solicita y el método para obtenerlo, además del constructor:

```

import java.io.Serializable;

public class PideFichero implements Serializable {
    String nombreFichero;

    public PideFichero(String nombreFichero) {
        this.nombreFichero = nombreFichero;
    }

    public String getNombreFichero() {
        return nombreFichero;
    }
}

```

La clase **ObtieneFichero** define un atributo con el contenido en bytes del fichero y el método para obtenerlo, además del constructor:

```

import java.io.Serializable;

public class ObtieneFichero implements Serializable {
    byte[] contenidoFichero; //contenido en bytes del fichero

    public ObtieneFichero(byte[] contenidoFichero) {
        this.contenidoFichero = contenidoFichero;
    }

    public byte[] getContenidoFichero() {
        return contenidoFichero;
    }
}

```

Si el objeto es **EnviaFichero**, *if (peticion instanceof EnviaFichero)*, es que el cliente envía un fichero para cargarlo en el servidor. Entonces en el servidor se crea un objeto **File** en el directorio (en este caso es el directorio seleccionado en el servidor) y con el nombre y contenido incluido en los atributos de este objeto:

```

File d = new File(fic.getDirectorio());
File f1 = new File(d,fic.getNombre());
//se crea el fichero en el directorio
//con los bytes enviados en el objeto
FileOutputStream fos = new FileOutputStream(f1);
fos.write(fic.getContenidoFichero());
fos.close();

```

Una vez creado se envía al cliente la nueva estructura del directorio donde se incluye el fichero cargado:

```

//se crea la nueva estructura de directorios
EstructuraFicheros n = new EstructuraFicheros(fic.getDirectorio());
outObjeto.writeObject(n);//se envia al cliente

```

La clase **EnviaFichero** define los siguientes atributos y métodos para obtener el valor de los atributos, además del constructor:

```

import java.io.Serializable;

public class EnviaFichero implements Serializable {
    byte[] contenidoFichero; //contenido en bytes del fichero
    String nombre;           //nombre del fichero
    String directorio;       //directorio donde se cargará

    public EnviaFichero (byte[] contenidoFichero, String nombre,
                        String directorio) {
        this.contenidoFichero = contenidoFichero;
        this.nombre = nombre;
        this.directorio = directorio;
    }
    public String getNombre()      { return nombre; }
    public String getDirectorio() { return directorio; }

    public byte[] getContenidoFichero() {
        return contenidoFichero;
    }
}

```

4.5.2. PROGRAMA CLIENTE

El programa cliente, clase **clienteFicheros**, extiende **JFrame** e implementa **Runnable**, *public class clienteFicheros extends JFrame implements Runnable {}*. En el método *main()* se crea un socket al servidor (*localhost* si el servidor está en la misma máquina que el cliente o dirección IP si el servidor está localizado en una máquina diferente) en el puerto pactado, se crea una instancia de esta clase, se prepara la pantalla y se lanza el hilo:

```

public static void main(String[] args) throws IOException {

    int puerto = 44441;
    //localhost o IP
    Socket s = new Socket("localhost", puerto);
    clienteFicheros hiloC = new clienteFicheros(s);
    hiloC.setBounds(0, 0, 540, 500);
    hiloC.setVisible(true);
    new Thread(hiloC).start();

} // ..FIN main

```

Al ejecutarse el cliente se muestra una pantalla con un JList donde aparecerá el contenido de los ficheros y directorios del directorio seleccionado en el servidor. Se muestran los botones para subir y descargar un fichero y el botón para finalizar la ejecución. También se muestran diferentes campos donde se visualizarán mensajes, véase Figura 4.33. La pantalla es parecida a la diseñada para el cliente FTP.

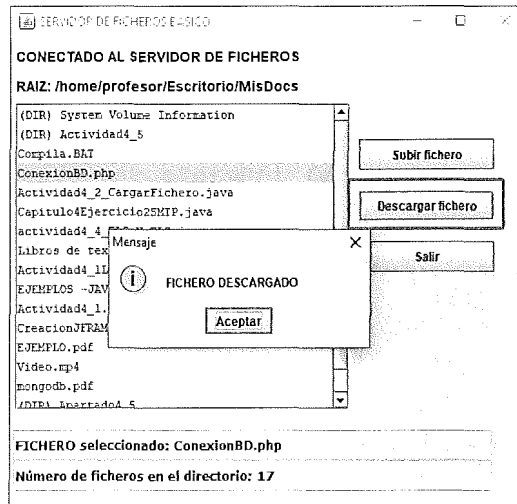


Figura 4.33. Ejecución del cliente, una vez descargado un fichero.

Algunas variables que se usarán en el ejemplo son las siguientes:

```
static Socket socket;
EstructuraFicheros nodo = null; //objeto EstructuraFicheros actual
ObjectInputStream inObjeto;      //stream de entrada
ObjectOutputStream outObjeto;    //stream de salida
EstructuraFicheros Raiz;         //objeto EstructuraFicheros Raiz

//lista para los datos del directorio
static JList listaDirec = new JList();

//para saber directorio y fichero seleccionado
static String direcSelec = "";    //nombre del directorio actual
static String ficheroSelec = "";  //fichero seleccionado
static String ficheroCompleto = ""; //directorio + fichero
```

En el constructor se obtienen los flujos de entrada y de salida, se definen los campos de pantalla, las acciones de los botones y al hacer clic en un elemento de JList:

```
public clienteFicheros(Socket s) throws IOException {
    super("SERVIDOR DE FICHeros BÁSICO");
    socket = s;
    try {
        outObjeto = new ObjectOutputStream(socket.getOutputStream());
        inObjeto = new ObjectInputStream(socket.getInputStream());
    } catch (IOException e) {
        e.printStackTrace();
        System.exit(0);
    }
    //definición de los campos de la pantalla
    //ACCIONES DE LOS BOTONES
    listaDirec.addListSelectionListener(new ListSelectionListener()... )
    botonSalir.addActionListener(new ActionListener() ...)
```

```

    botonDescargar.addActionListener(new ActionListener()...)
    botonCargar.addActionListener(new ActionListener()...)
} //constructor

```

En el método *run()* se lee el objeto **EstructuraFicheros** que manda el servidor, se carga en el array *nodos* la lista de ficheros y se manda al método *llenarLista()* para llenar el JList y mostrarlo en pantalla:

```

public void run() {
    try {
        cab.setText("Conectando con el servidor .....");
        // OBTENER DIRECTORIO RAIZ
        Raiz = (EstructuraFicheros) inObjeto.readObject();
        EstructuraFicheros[] nodos = Raiz.getList(); //lista de ficheros
        direcSelec = Raiz.getPath(); //directorio actual
        llenarLista(nodos, Raiz.getNumFich());
        cab3.setText("RAIZ: " + direcSelec);
        cab.setText("CONECTADO AL SERVIDOR DE FICHEROS");
        campo2.setText("Número de ficheros en el directorio: " +
                       Raiz.getNumFich());
    } catch (IOException e1) {e1.printStackTrace();System.exit(1);}
    catch (ClassNotFoundException e1)
        {e1.printStackTrace();System.exit(1);}
} //fin run

```

El método *llenarLista()*, recibe un array de elementos **EstructuraFicheros** y el número de elementos del array, llena el JList. Cada elemento del JList es un objeto **EstructuraFicheros**:

```

private static void llenarLista(EstructuraFicheros[] files, int numero)
{
    if (numero == 0) return;
    DefaultListModel modeloLista = new DefaultListModel();
    listaDirec.setForeground(Color.blue);
    Font fuente = new Font("Courier", Font.PLAIN, 12);
    listaDirec.setFont(fuente);
    listaDirec.removeAll();

    for (int i = 0; i < files.length; i++)
        modeloLista.addElement(files[i]);

    try {
        listaDirec.setModel(modeloLista);
    } catch (NullPointerException n) {}
} // Fin llenarLista

```

El código asociado al botón *Salir* es cerrar el socket y finalizar la ejecución:

```

botonSalir.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        try {
            socket.close(); //cerrar socket
            System.exit(0);
        } catch (IOException ex) {ex.printStackTrace();}
    }
});

```

El siguiente código se ejecuta al hacer clic en un elemento de la lista, se obtiene el objeto **EstructuraFicheros** sobre el que hemos hecho clic, se comprueba si es un directorio, en este

caso se visualiza un mensaje (por simplificar el ejemplo no se permite la navegación por los directorios); si no lo es se trata de un fichero, se guarda el nombre y el nombre completo en las variables:

```
listaDirec.addListSelectionListener(new ListSelectionListener() {
    public void valueChanged(ListSelectionEvent lse) {
        if (lse.getValueIsAdjusting()) {
            ficheroSelec = "";
            ficheroCompleto = "";

            //se obtiene el elemento seleccionado de la lista
            nodo = (EstructuraFicheros) listaDirec.getSelectedValue();
            if (nodo.isDir()) {
                //ES UN DIRECTORIO
                campo.setText("FUNCIÓN NO IMPLEMENTADA..... ");
            } else {
                //SE TRATA DE UN FICHERO
                ficheroSelec = nodo.getName();
                ficheroCompleto = nodo.getPath();
                campo.setText("FICHERO seleccionado: " + ficheroSelec);
            } // fin else
        } // fin valueChanged
    }
}); //fin lista
```

Al pulsar en el botón *Subir fichero* se abre una ventana (**JFileChooser**) desde la que el cliente elige el fichero a subir al servidor. Una vez elegido se leen los bytes del mismo y se van almacenando en un array de bytes para mandárselo al servidor en un objeto **EnviaFichero**. Se envía al servidor y el servidor responde con un objeto **EstructuraFicheros** que contiene la lista de ficheros actualizada. Se llama al método *llenarLista()* para que se muestre de nuevo la lista de ficheros incluyendo el fichero que se acaba de subir:

```
botonCargar.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        JFileChooser = new JFileChooser();
        f.setFileSelectionMode(JFileChooser.FILES_ONLY);
        f.setDialogTitle("Selecciona el Fichero a SUBIR AL SERVIDOR DE FICHEROS");
        int returnVal = f.showDialog(f, "Cargar");
        if (returnVal == JFileChooser.APPROVE_OPTION) {
            File file = f.getSelectedFile();
            String archivo = file.getAbsolutePath();
            String nombreArchivo = file.getName();
            BufferedInputStream in;

            //leer fichero y almacenarlo en array de bytes
            try {
                in = new BufferedInputStream
                    (new FileInputStream(archivo));
                long bytes = file.length();
                byte[] buff = new byte[(int) bytes];
                int i, j = 0;
                //lectura del fichero
                while ((i = in.read()) != -1) {
                    buff[j] = (byte) i; //carga datos en el array
                    j++;
                }
            }
        }
    }
});
```

```

        in.close(); // cerrar stream de entrada

        //Crear objeto EnviaFichero con los bytes del fichero,
        //el nombre y el directorio donde se cargará
        Object ff = new EnviaFichero(buff,
                                     nombreArchivo, direcSelec);
        //se envia al servidor
        outObjeto.writeObject(ff);
        JOptionPane.showMessageDialog(null, "FICHERO CARGADO");

        //obtengo de nuevo la lista de ficheros
        nodo = (EstructuraFicheros) inObjeto.readObject();
        EstructuraFicheros[] lista = nodo.getList();
        direcSelec = nodo.getPath();
        llenarLista(lista, nodo.getNumFich());
        campo2.setText("Número de ficheros en el directorio:
                       " + lista.length);

    } catch (FileNotFoundException e1) {e1.printStackTrace();}
    catch (IOException ee) {ee.printStackTrace(); }
    catch (ClassNotFoundException e2) {e2.printStackTrace();}
}
}); //Fin boton cargar

```

Al pulsar en el botón *Descargar fichero* se hace una petición al servidor mediante el objeto **PideFichero** donde se solicita el fichero seleccionado. Antes hay que comprobar si se ha hecho clic sobre algún fichero del **JList**, eso se hace preguntando por el valor de la variable *ficheroCompleto*. A continuación se abre un **FileOutputStream** con el nombre del fichero seleccionado para copiar lo que se reciba del **ObjectInputStream** del socket. Se lee el objeto recibido del socket, **ObtieneFichero**, y se pasan los bytes al fichero abierto mediante **FileOutputStream**. El fichero se creará en nuestro directorio de trabajo actual y el nombre será el mismo que aparece en el **JList**:

```

botonDescargar.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if (ficheroCompleto.equals("")) return; //no se ha seleccionado
        //PIDO ESTE FICHERO ficheroCompleto
        PideFichero pido = new PideFichero(ficheroCompleto);
        try {
            //pido fichero al servidor
            outObjeto.writeObject(pido);

            //se crea fichero con el nombre del seleccionado
            //en el directorio actual
            FileOutputStream fos = new FileOutputStream(ficheroSelec);

            //recibo el fichero del servidor
            Object obtengo = inObjeto.readObject();

            if (obtengo instanceof ObtieneFichero) {
                ObtieneFichero fic = (ObtieneFichero) obtengo;
                fos.write(fic.getContenidoFichero()); //escribo bytes
                fos.close();
                JOptionPane.showMessageDialog(null, "FICHERO DESCARGADO");
            }
        } catch (IOException e1) {e1.printStackTrace();}
    }
}

```

```

        catch (ClassNotFoundException e1) {e1.printStackTrace();}
    }
}); // Fin boton descargar

```

En este ejemplo solo se permite cargar y descargar ficheros del directorio seleccionado en el servidor. Si el directorio seleccionado tiene a su vez otros directorios, estos se mostrarán en pantalla con la palabra (DIR) delante pero la navegación a través de ellos no se podrá realizar. Se mostrará un mensaje indicando que la función no está implementada: *FUNCIÓN NO IMPLEMENTADA*, véase Figura 34.

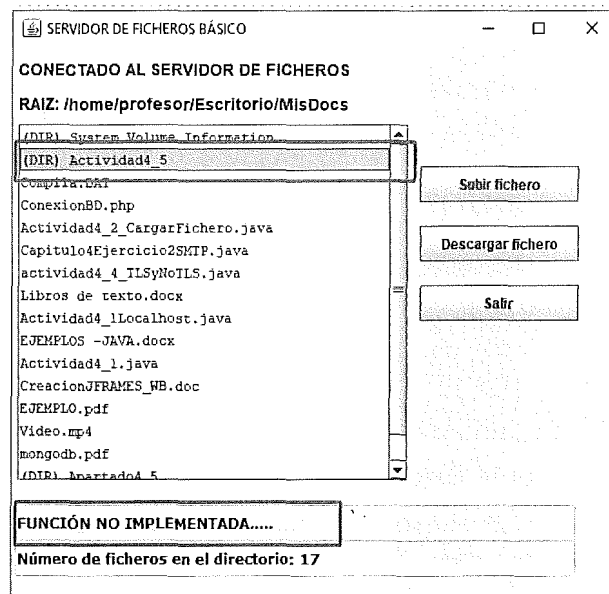


Figura 4.34. Ejecución del cliente, función no implementada.

Tal como está diseñado el ejemplo si hay varios clientes conectados en un momento determinado y uno de los clientes sube varios ficheros al servidor, estos no serán visibles para el resto de los clientes conectados hasta que no reciban por parte del servidor un objeto **EstructuraFicheros** con la lista actual de los ficheros del directorio, y esto se produce cuando el cliente carga un fichero al servidor. Se puede añadir un botón para que el cliente solicite en cualquier momento la lista actual de los ficheros en el servidor.

Para ejecutar el ejemplo las clases *EnviaFichero*, *EstructuraFicheros*, *ObtieneFichero* y *PideFichero* tienen que estar tanto en la máquina cliente como en el servidor. Además, en la máquina cliente se necesita la clase *clienteFicheros* y en el servidor las clases *Servidor* e *HiloServidor*.

Para hacer el ejemplo más sencillo se ha optado por enviar el fichero en un solo objeto, es decir, se lee todo el fichero, se almacena en un array de bytes y se envía. Si el fichero es muy grande pueden ocurrir errores de memoria como *java.lang.OutOfMemoryError: Java heap space*. Para evitarlo se puede enviar el fichero por trozos, por ejemplo, enviarlo en trozos de 4000 bytes.

COMPRUEBA TU APRENDIZAJE

1º) Realiza los cambios necesarios en el cliente FTP para que se pueda conectar a cualquier servidor FTP. Haz que el usuario tenga que introducir el nombre del servidor FTP, nombre de usuario y su clave, ya sea desde la línea de comandos o utilizando pantalla gráfica.

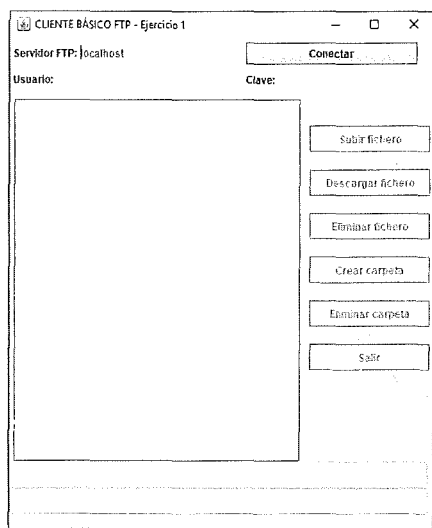


Figura 4.35. Pantalla inicial del Ejercicio 1.

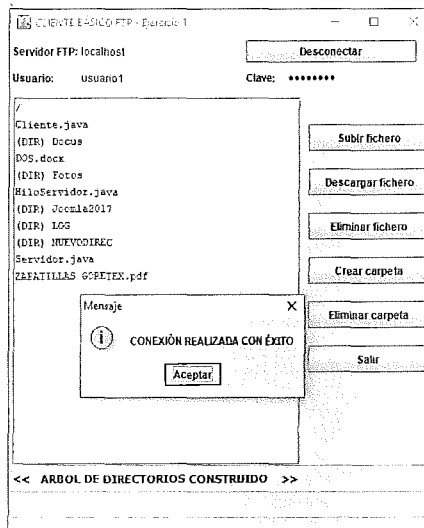


Figura 4.36. Ejercicio 1 con conexión al servidor.

Por ejemplo, la Figura 4.35 muestra una pantalla desde la que se introducen los datos. Inicialmente todos los botones están deshabilitados, al pulsar en el botón *Conectar*, y si la conexión ha tenido éxito, se habilitan para realizar las operaciones, véase Figura 4.36 Visualiza los mensajes que puedan surgir usando *JOptionPane.showMessageDialog*. Para introducir la contraseña utiliza la clase **JPasswordField** (*JPasswordField clave = new JPasswordField();*)

2º) Realiza un programa Java que cree un cliente SMTP. Diseña una pantalla (véase Figura 4.37) desde la que introduciremos los datos necesarios para la conexión: el nombre del servidor SMTP, el puerto, el nombre de usuario y su clave; los datos para enviar el correo: la dirección de correo del remitente y del destinatario, el asunto y el cuerpo del mensaje. Los botones de opción definen la negociación SSL/TLS: *Sin TLS*, no se necesita negociación y *Con TLS* se necesita la negociación (hay que ejecutar el método **execTLS()**).



Figura 4.37. Pantalla inicial Ejercicio 2.



Figura 4.38. Ejercicio 2 con mensaje enviado.

Al pulsar en el botón *Conectar* se debe realizar la conexión con el servidor SMTP y se debe activar el botón *Enviar mensaje*. También debe aparecer en el lugar del botón *Conectar* el botón *Desconectar* para realizar la desconexión del servidor. Visualiza los mensajes que puedan surgir, por ejemplo: “Conexión realizada”, “Mensaje enviado”, “Usuario autenticado”, “No se puede realizar la conexión”, etc... usando *JOptionPane.showMessageDialog*, véase Figura 4.38.

3º) Crea un cliente de correo POP3 desde el que introduciremos el nombre del servidor POP (por defecto que aparezca **localhost**), el puerto (por defecto **110**), el nombre de usuario, la clave y el modo en el que se realiza la negociación (implícito o no implícito). Inicialmente el botón *Conectar* está desactivado y no aparece ningún botón de opción seleccionado. Al seleccionar una de las opciones se activará el botón *Conectar*. Debe aparecer un *JList* en la que aparecerá la lista con los mensajes a recuperar y un *JTextArea* no editable para que muestre el mensaje seleccionado del *JList*, véase Figura 4.39.

Al pulsar el botón *Conectar* se debe activar el botón *Recuperar mensajes del servidor* y el botón *Conectar* aparecerá como *Desconectar*. Si se pulsa el botón *Recuperar mensajes del servidor* en el *JList* aparecen los mensajes por leer. Al pulsar en el mensaje en el *JTextArea* se visualizará su contenido (Figura 4.40). Al pulsar en el botón *Desconectar* se cierra la sesión del usuario y la conexión al servidor, se limpia la pantalla, el botón *Desconectar* cambia a *Conectar* y se desactiva el botón *Recuperar mensajes del servidor*. Visualiza los mensajes que puedan surgir usando *JOptionPane.showMessageDialog*.

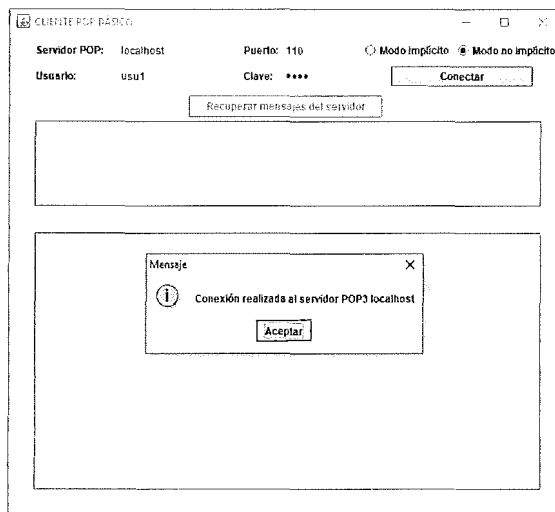


Figura 4.39. Pantalla inicial Ejercicio 3.

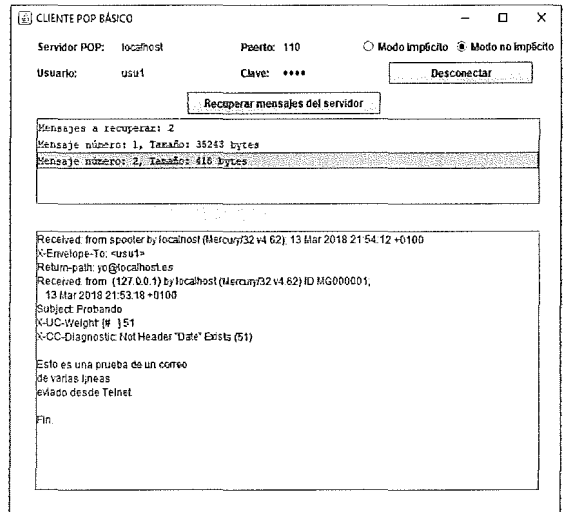


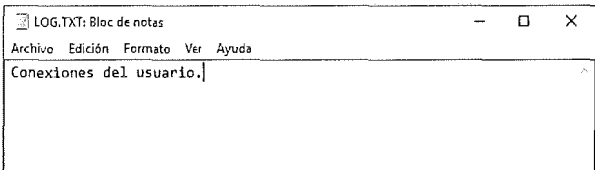
Figura 4.40. Pantalla Ejercicio 3 con mensajes

4º) Añade el botón *Actualizar* al programa cliente **clienteFicheros**. Este botón le muestra al cliente el contenido actual del directorio seleccionado en el servidor. Para solicitar al servidor esta información realiza una petición mediante un objeto, por ejemplo, **PideDirectorio**, en el que solicites al servidor la información del directorio; el servidor enviará al cliente un nuevo objeto **EstructuraFicheros** con la información actual y el cliente la mostrará en pantalla.

ACTIVIDADES DE AMPLIACIÓN

1º) A partir del servidor de ficheros del epígrafe 4.5 haz las modificaciones necesarias para que se pueda navegar a través de los directorios que hay dentro del directorio seleccionado en el servidor, como se hizo con el cliente FTP.

2º) Uso del servidor FTP localhost, preparación del entorno. Crea 3 usuarios (si ya están creados no hace falta crearlos de nuevo), crea una carpeta para cada usuario, y dentro de esa carpeta de usuario crear una nueva carpeta con nombre LOG, y dentro de esta carpeta crear un fichero de texto llamado LOG.TXT. El contenido inicial del fichero es el siguiente:



A continuación, se muestran los nombres de usuarios, contraseñas, carpetas y subcarpetas a crear, Sobre esta carpeta el usuario tendrá control total:

Nombre de usuario	contraseña	Carpetas
usuario1	usu1	usuario1 usuario1/LOG
usuario2	usu2	usuario2 usuario2/LOG
usuario3	usu3	usuario3 usuario3/LOG

Realiza un programa Java que pida por teclado un nombre de usuario y su contraseña en un proceso repetitivo que finalizará cuando el nombre de usuario sea *. Una vez introducido el nombre y la contraseña, conectar el usuario al servidor FTP *localhost*. Cada vez que se conecte, hay que acceder a su directorio */LOG*, en este directorio hay un fichero llamado **LOG.TXT** en el que debes registrar la información sobre la fecha y hora de la conexión del usuario. Ejemplo de contenido del fichero para un usuario puede ser lo siguiente:

```
Conexiones del usuario.  
Hora de conexión: Mon Mar 06 11:00:09 CET 2018
```

Si el usuario vuelve a conectarse hay acceder de nuevo al directorio y modificar el fichero LOG.TXT, hay que añadir una nueva línea. Y así sucesivamente. Ejemplo, el usuario ha realizado 3 conexiones:

```
Conexiones del usuario.  
Hora de conexión: Mon Mar 06 11:00:09 CET 2018  
Hora de conexión: Mon Mar 06 11:04:59 CET 2018  
Hora de conexión: Tue Mar 07 08:55:12 CET 2018
```

Para comprobar si se van añadiendo correctamente los datos, se consultará el fichero LOG.TXT para ver si refleja las conexiones realizadas por el usuario.

Al finalizar el proceso de lectura deberás enviar un mensaje de correo electrónico a alguna de tus cuentas, indicando el número de usuarios correctos que se han conectado al servidor FTP hasta introducir como nombre de usuario el *. Debes añadir un asunto al correo. Utiliza la clase **AuthenticatingSMTPClient** para crear el cliente SMTP.