Instituto Politécnico Nacional

Escuela Superior de Computo

Práctica 2
Hilos y Sockets

Unidad de aprendizaje: Sistemas Distribuidos

Profesor: M. en C. Carreto Arellano Chadwick

Alumno:

Peñarrieta Villa Jesus

Grupo: 7CM6

Introducción

En el entorno actual, donde la transferencia de archivos entre sistemas remotos es una necesidad constante ya sea en aplicaciones empresariales, educativas o personales, resulta fundamental comprender e implementar protocolos robustos de comunicación. Uno de los protocolos más antiguos y utilizados para esta finalidad es el FTP (File Transfer Protocol), el cual permite la transferencia de archivos entre un cliente y un servidor a través de una red TCP/IP.El protocolo FTP fue diseñado en los años 70 y sigue siendo ampliamente utilizado debido a su simplicidad y funcionalidad. Opera en un modelo cliente-servidor, donde el cliente solicita operaciones como subir (upload), descargar (download), borrar o listar archivos, y el servidor responde ejecutando estas acciones sobre un sistema de archivos determinado.

FTP utiliza tradicionalmente dos canales:

- **Canal de control**: mediante el cual el cliente y el servidor intercambian comandos y respuestas (por ejemplo, USER, PASS, LIST, RETR, STOR, CWD, etc.).
- **Canal de datos**: encargado de la transferencia efectiva de archivos o resultados (como el listado de directorios).

En este proyecto, se implementó un sistema propio denominado QuantumDrive, que simula un servidor FTP personalizado y un cliente gráfico (usando Java y Swing) capaz de comunicarse con él a través de sockets TCP bloqueantes. A diferencia de implementaciones modernas que usan protocolos más complejos o sockets no bloqueantes, esta versión inicial sirve como una introducción clara al funcionamiento básico del protocolo FTP y a la programación de redes.El uso de sockets bloqueantes implica que cada operación de lectura o escritura detiene el hilo hasta que se completa la transmisión de datos. Esta característica facilita la lógica de implementación y depuración, pero puede representar una limitación en entornos concurrentes de alta demanda. Aun así, esta aproximación es ideal para entender cómo fluye la comunicación entre cliente y servidor, cómo se estructuran los comandos y cómo se gestionan las rutas, directorios, archivos y sus metadatos.

A través de este proyecto, se busca no solo replicar la funcionalidad de un cliente FTP tradicional, sino también desarrollar una comprensión profunda del protocolo FTP, de la arquitectura cliente-servidor, y del manejo de archivos mediante programación en Java, utilizando herramientas como Streams, Threads, sockets y compresión ZIP, entre otras.

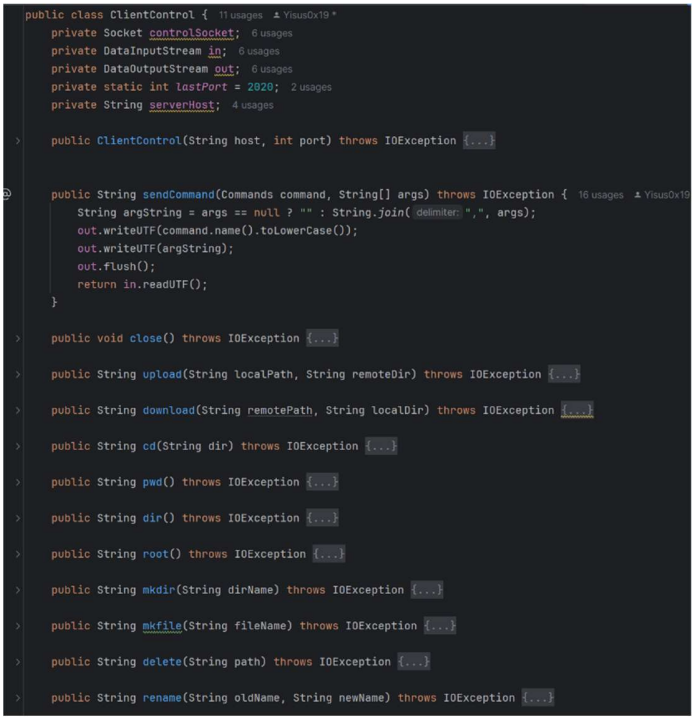
Objetivo

Diseñar e implementar un sistema de transferencia de archivos inspirado en el protocolo FTP, utilizando programación en Java y sockets TCP bloqueantes para simular la comunicación entre un cliente y un servidor. El sistema debe permitir la navegación por el sistema de archivos remoto, así como realizar operaciones típicas de un cliente FTP, como subir (upload), descargar (download), crear directorios y archivos, eliminar elementos y consultar metadatos.

Desarrollo

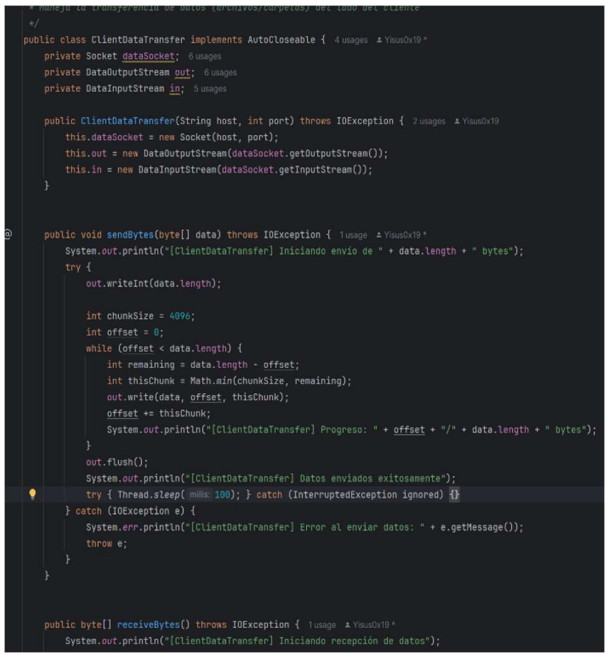
La práctica se compone de múltiples módulos clave:

1. Cliente (ClientControl.java):
    - Encargado de la conexión al servidor y el envío de comandos.
    - Gestiona comandos como `upload`, `download`, `cd`, `dir`, `stat`, `mkdir`, `mkfile`, `delete`, etc.
    - Utiliza flujos `DataInputStream` y `DataOutputStream` para la comunicación.

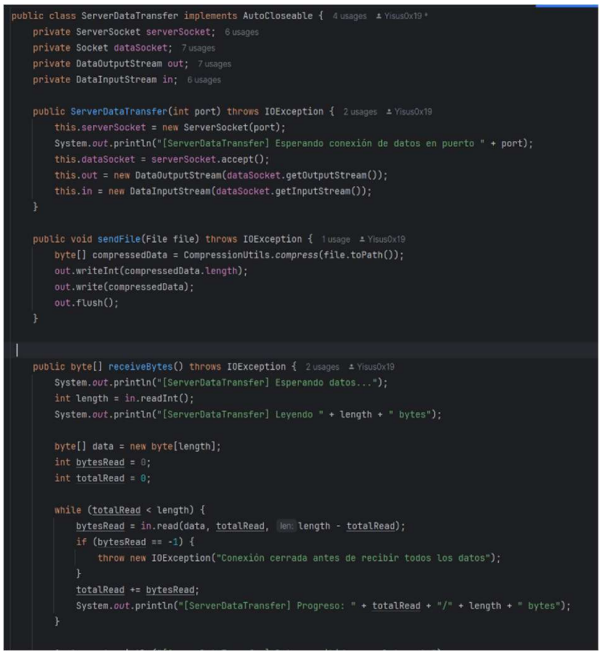


```
public class ClientControl {  11 usages  YisusOx19 *
    private Socket controlSocket;  6 usages
    private DataInputStream in;  6 usages
    private DataOutputStream out;  6 usages
    private static int lastPort = 2020;  2 usages
    private String serverHost;  4 usages

    public ClientControl(String host, int port) throws IOException {...}


    public String sendCommand(Commands command, String[] args) throws IOException {  16 usages  YisusOx19
        String argString = args == null ? "" : String.join( delimiter: ",", args);
        out.writeUTF(command.name().toLowerCase());
        out.writeUTF(argString);
        out.flush();
        return in.readUTF();
    }

    public void close() throws IOException {...}

    public String upload(String localPath, String remoteDir) throws IOException {...}

    public String download(String remotePath, String localDir) throws IOException {...}

    public String cd(String dir) throws IOException {...}

    public String pwd() throws IOException {...}

    public String dir() throws IOException {...}

    public String root() throws IOException {...}

    public String mkdir(String dirName) throws IOException {...}

    public String mkfile(String fileName) throws IOException {...}

    public String delete(String path) throws IOException {...}

    public String rename(String oldName, String newName) throws IOException {...}
```

2. Transferencia de Datos (ClientDataTransfer.java y ServerDataTransfer.java):
    - Manejan los sockets dedicados para transferir datos binarios (archivos comprimidos).
    - Usan `ServerSocket` y `Socket` para establecer una comunicación secundaria en un puerto dinámico.



```java
/**
 * Maneja la transferencia de datos (archivos/carpetas) del lado del cliente
 */
public class ClientDataTransfer implements AutoCloseable {  4 usages  YisusOx19 *
    private Socket dataSocket;  6 usages
    private DataOutputStream out;  6 usages
    private DataInputStream in;  5 usages

    public ClientDataTransfer(String host, int port) throws IOException {  2 usages  YisusOx19
        this.dataSocket = new Socket(host, port);
        this.out = new DataOutputStream(dataSocket.getOutputStream());
        this.in = new DataInputStream(dataSocket.getInputStream());
    }


    public void sendBytes(byte[] data) throws IOException {  1 usage  YisusOx19 *
        System.out.println("[ClientDataTransfer] Iniciando envio de " + data.length + " bytes");
        try {
            out.writeInt(data.length);

            int chunkSize = 4096;
            int offset = 0;
            while (offset < data.length) {
                int remaining = data.length - offset;
                int thisChunk = Math.min(chunkSize, remaining);
                out.write(data, offset, thisChunk);
                offset += thisChunk;
                System.out.println("[ClientDataTransfer] Progreso: " + offset + "/" + data.length + " bytes");
            }
            out.flush();
            System.out.println("[ClientDataTransfer] Datos enviados exitosamente");
            try { Thread.sleep( millis: 100); } catch (InterruptedException ignored) {}
        } catch (IOException e) {
            System.err.println("[ClientDataTransfer] Error al enviar datos: " + e.getMessage());
            throw e;
        }
    }


    public byte[] receiveBytes() throws IOException {  1 usage  YisusOx19 *
        System.out.println("[ClientDataTransfer] Iniciando recepción de datos");
```



```java
public class ServerDataTransfer implements AutoCloseable {  4 usages  YisusOx19 *
    private ServerSocket serverSocket;  6 usages
    private Socket dataSocket;  7 usages
    private DataOutputStream out;  7 usages
    private DataInputStream in;  6 usages

    public ServerDataTransfer(int port) throws IOException {  2 usages  YisusOx19
        this.serverSocket = new ServerSocket(port);
        System.out.println("[ServerDataTransfer] Esperando conexión de datos en puerto " + port);
        this.dataSocket = serverSocket.accept();
        this.out = new DataOutputStream(dataSocket.getOutputStream());
        this.in = new DataInputStream(dataSocket.getInputStream());
    }

    public void sendFile(File file) throws IOException {  1 usage  YisusOx19
        byte[] compressedData = CompressionUtils.compress(file.toPath());
        out.writeInt(compressedData.length);
        out.write(compressedData);
        out.flush();
    }


    public byte[] receiveBytes() throws IOException {  2 usages  YisusOx19
        System.out.println("[ServerDataTransfer] Esperando datos...");
        int length = in.readInt();
        System.out.println("[ServerDataTransfer] Leyendo " + length + " bytes");

        byte[] data = new byte[length];
        int bytesRead = 0;
        int totalRead = 0;

        while (totalRead < length) {
            bytesRead = in.read(data, totalRead,  len: length - totalRead);
            if (bytesRead == -1) {
                throw new IOException("Conexión cerrada antes de recibir todos los datos");
            }
            totalRead += bytesRead;
            System.out.println("[ServerDataTransfer] Progreso: " + totalRead + "/" + length + " bytes");
        }
```

-

3. Servidor (ServerControl.java, ServerSessionHandler.java):
    - Recibe conexiones y comandos desde los clientes.
    - Ejecuta operaciones sobre el sistema de archivos del servidor.
    - Genera respuestas adecuadas a cada comando recibido.
    -

El servidor está diseñado para seguir una arquitectura modular basada en el protocolo FTP, en la que se separan las responsabilidades entre la gestión de comandos y la transferencia de datos. Las clases principales del servidor son ServerControl, ServerSessionHandler y ServerDataTransfer, las cuales colaboran para manejar múltiples clientes de forma simultánea utilizando sockets bloqueantes.

1. ServerControl
La clase ServerControl es la encargada de arrancar el servidor y aceptar nuevas conexiones entrantes. Para ello, crea un ServerSocket que escucha constantemente en un puerto de control (por ejemplo, el puerto 2121). Cuando un cliente se conecta, el servidor crea una nueva instancia de ServerSessionHandler, la cual es ejecutada en un hilo independiente. Esta separación permite que múltiples clientes puedan conectarse y trabajar de forma concurrente sin bloquearse entre sí.

2. ServerSessionHandler
Cada cliente es manejado por su propia instancia de ServerSessionHandler, lo que permite que las sesiones sean aisladas y concurrentes. Esta clase se encarga de interpretar los comandos enviados por el cliente (por ejemplo: UPLOAD, DOWNLOAD, CD, DIR, MKDIR, etc.) y de enviar las respuestas correspondientes. Utiliza flujos de entrada y salida (DataInputStream y DataOutputStream) para intercambiar información con el cliente a través del canal de control.

Cuando el cliente solicita subir o descargar archivos, el ServerSessionHandler delega la operación de transferencia de datos a la clase ServerDataTransfer, utilizando un puerto secundario temporal. Esto emula el comportamiento del protocolo FTP tradicional, donde las transferencias se realizan por un canal aparte.

3. ServerDataTransfer
La clase ServerDataTransfer se encarga exclusivamente de la transferencia binaria de archivos o carpetas. Para ello, crea un nuevo ServerSocket en un puerto específico (indicado por el cliente) y espera a que el cliente se conecte. Una vez establecida la conexión, esta clase puede:

- Recibir datos comprimidos desde el cliente y almacenarlos en el servidor (para uploads).

- Enviar archivos comprimidos al cliente (para downloads).

Esta operación se realiza utilizando flujos de bytes, lo cual permite una transferencia eficiente y directa de datos binarios.

```
public class ServerControl implements Runnable {  3 usages  Yisus0x19
    private ServerSocket serverSocket;  3 usages
    private boolean running = true;  2 usages

    public ServerControl(int port) throws IOException {  1 usage  Yisus0x19
        this.serverSocket = new ServerSocket(port);
    }

    @Override  Yisus0x19
    public void run() {
        while (running) {
            try {
                Socket clientSocket = serverSocket.accept();
                // Aquí se debe manejar la sesión del cliente en un hilo separado
                new Thread(new ServerSessionHandler(clientSocket)).start();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    public void stop() throws IOException {  no usages  Yisus0x19
        running = false;
        serverSocket.close();
    }
}
```

Flujo de Comunicación

1. ServerControl acepta una nueva conexión.

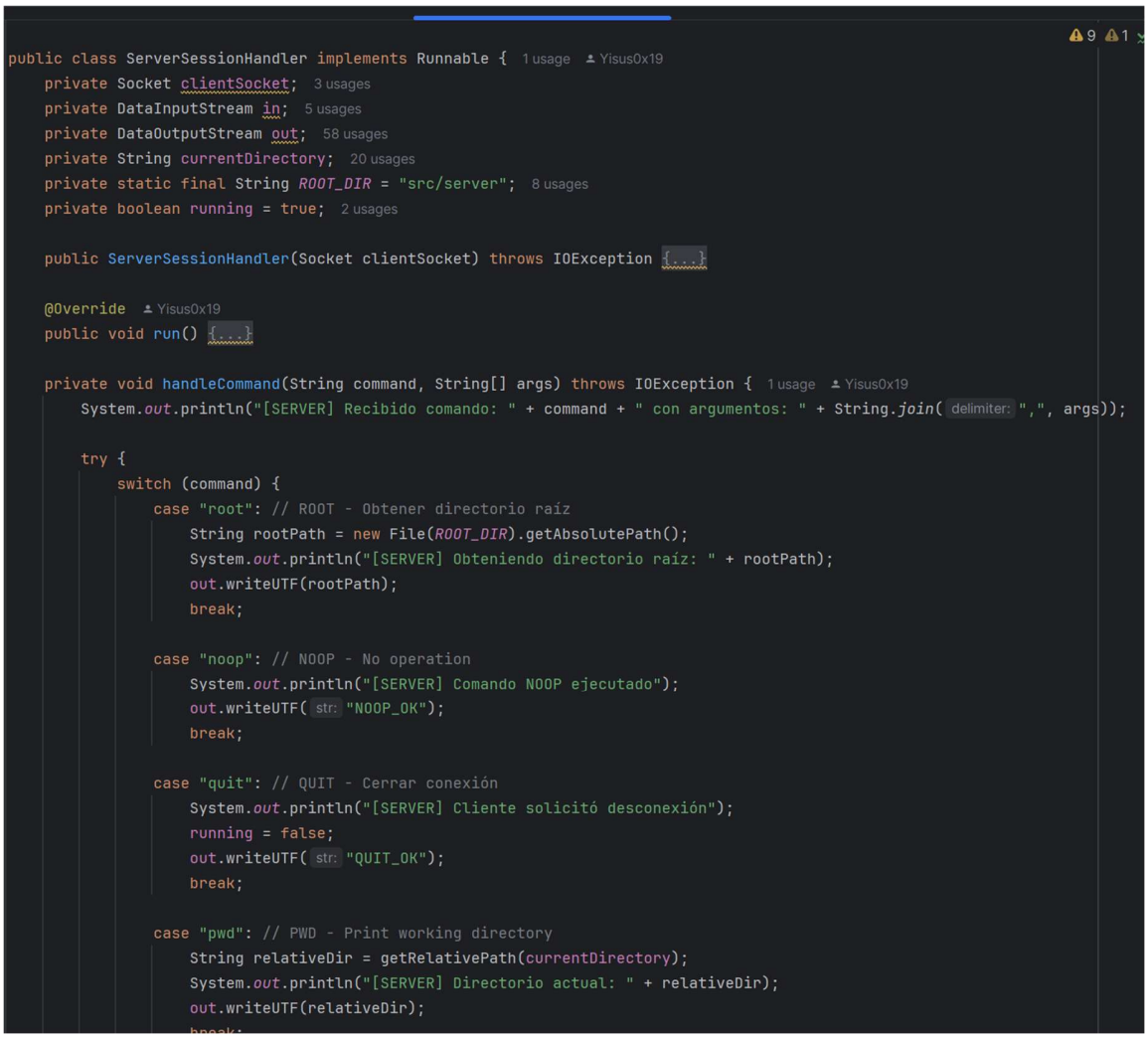
2. Crea un hilo con una instancia de ServerSessionHandler.

3. El cliente envía comandos por el canal de control.

4. ServerSessionHandler procesa los comandos.

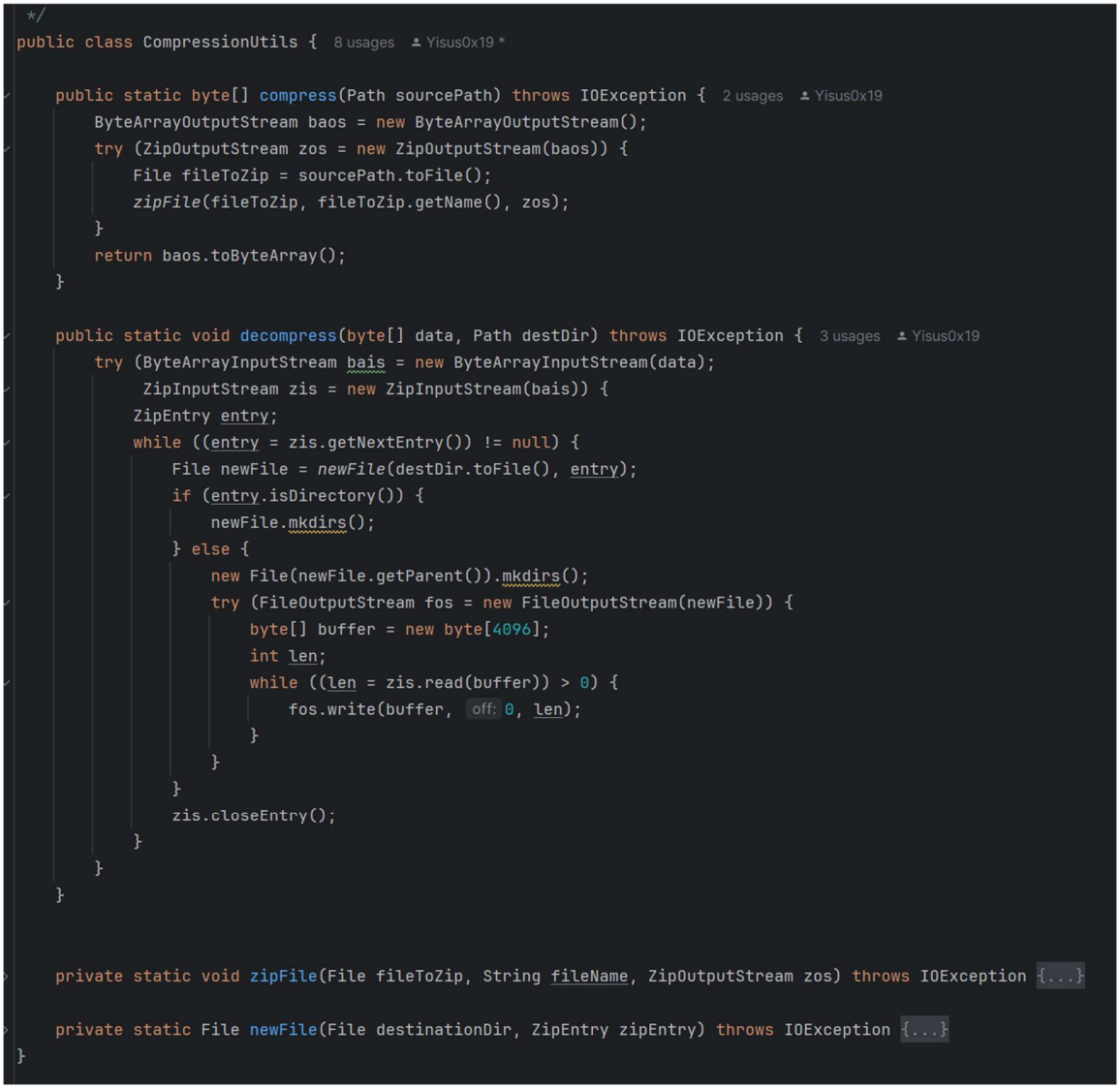
5. Si se requiere transferencia de archivos, se activa ServerDataTransfer en un puerto dinámico.

Esto permite que el canal de control permanezca limpio y enfocado en los comandos, mientras que los datos se transfieren por un canal dedicado.



```java
public class ServerSessionHandler implements Runnable {  1 usage  Yisus0x19
    private Socket clientSocket;  3 usages
    private DataInputStream in;  5 usages
    private DataOutputStream out;  58 usages
    private String currentDirectory;  20 usages
    private static final String ROOT_DIR = "src/server";  8 usages
    private boolean running = true;  2 usages

    public ServerSessionHandler(Socket clientSocket) throws IOException {...}

    @Override  Yisus0x19
    public void run() {...}

    private void handleCommand(String command, String[] args) throws IOException {  1 usage  Yisus0x19
        System.out.println("[SERVER] Recibido comando: " + command + " con argumentos: " + String.join( delimiter: ",", args));

        try {
            switch (command) {
                case "root": // ROOT - Obtener directorio raíz
                    String rootPath = new File(ROOT_DIR).getAbsolutePath();
                    System.out.println("[SERVER] Obteniendo directorio raíz: " + rootPath);
                    out.writeUTF(rootPath);
                    break;

                case "noop": // NOOP - No operation
                    System.out.println("[SERVER] Comando NOOP ejecutado");
                    out.writeUTF( str: "NOOP_OK");
                    break;

                case "quit": // QUIT - Cerrar conexión
                    System.out.println("[SERVER] Cliente solicitó desconexión");
                    running = false;
                    out.writeUTF( str: "QUIT_OK");
                    break;

                case "pwd": // PWD - Print working directory
                    String relativeDir = getRelativePath(currentDirectory);
                    System.out.println("[SERVER] Directorio actual: " + relativeDir);
                    out.writeUTF(relativeDir);
                    break;
```

4. Compresión (CompressionUtils.java):
    - Se encarga de comprimir archivos y carpetas antes de transferirlos.
    - También descomprime los archivos al llegar al destino.



```
 */
public class CompressionUtils {  8 usages  Yisus0x19 *

    public static byte[] compress(Path sourcePath) throws IOException {  2 usages  Yisus0x19
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        try (ZipOutputStream zos = new ZipOutputStream(baos)) {
            File fileToZip = sourcePath.toFile();
            zipFile(fileToZip, fileToZip.getName(), zos);
        }
        return baos.toByteArray();
    }

    public static void decompress(byte[] data, Path destDir) throws IOException {  3 usages  Yisus0x19
        try (ByteArrayInputStream bais = new ByteArrayInputStream(data);
             ZipInputStream zis = new ZipInputStream(bais)) {
            ZipEntry entry;
            while ((entry = zis.getNextEntry()) != null) {
                File newFile = newFile(destDir.toFile(), entry);
                if (entry.isDirectory()) {
                    newFile.mkdirs();
                } else {
                    new File(newFile.getParent()).mkdirs();
                    try (FileOutputStream fos = new FileOutputStream(newFile)) {
                        byte[] buffer = new byte[4096];
                        int len;
                        while ((len = zis.read(buffer)) > 0) {
                            fos.write(buffer,  off: 0, len);
                        }
                    }
                }
                zis.closeEntry();
            }
        }
    }


    private static void zipFile(File fileToZip, String fileName, ZipOutputStream zos) throws IOException {...}

    private static File newFile(File destinationDir, ZipEntry zipEntry) throws IOException {...}
}
```

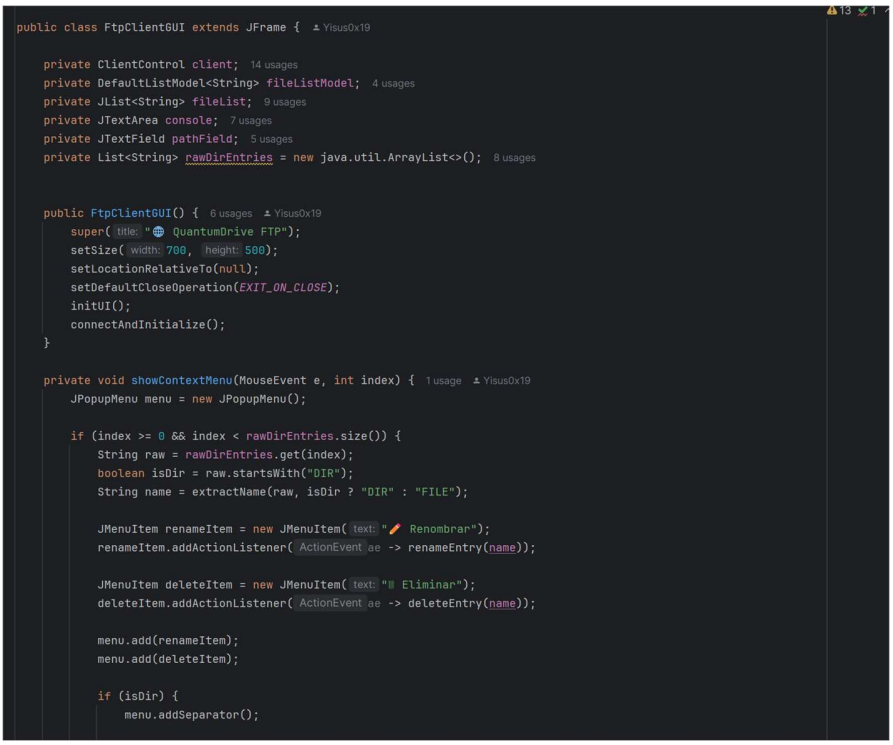
Una parte fundamental del sistema QuantumDrive es la capacidad de transferir archivos y carpetas de forma eficiente, especialmente cuando se trata de múltiples archivos o estructuras complejas de directorios. Para lograr esto, se incorporó una clase utilitaria llamada CompressionUtils, la cual se encarga de comprimir archivos y descomprimirlos al enviarlos o recibirlos, respectivamente.Esta clase encapsula toda la lógica relacionada con la compresión ZIP y opera sobre flujos de bytes (byte[]), lo que permite integrarla fácilmente con las clases ClientDataTransfer y ServerDataTransfer.

Funcionalidad principal
La clase contiene dos métodos públicos:

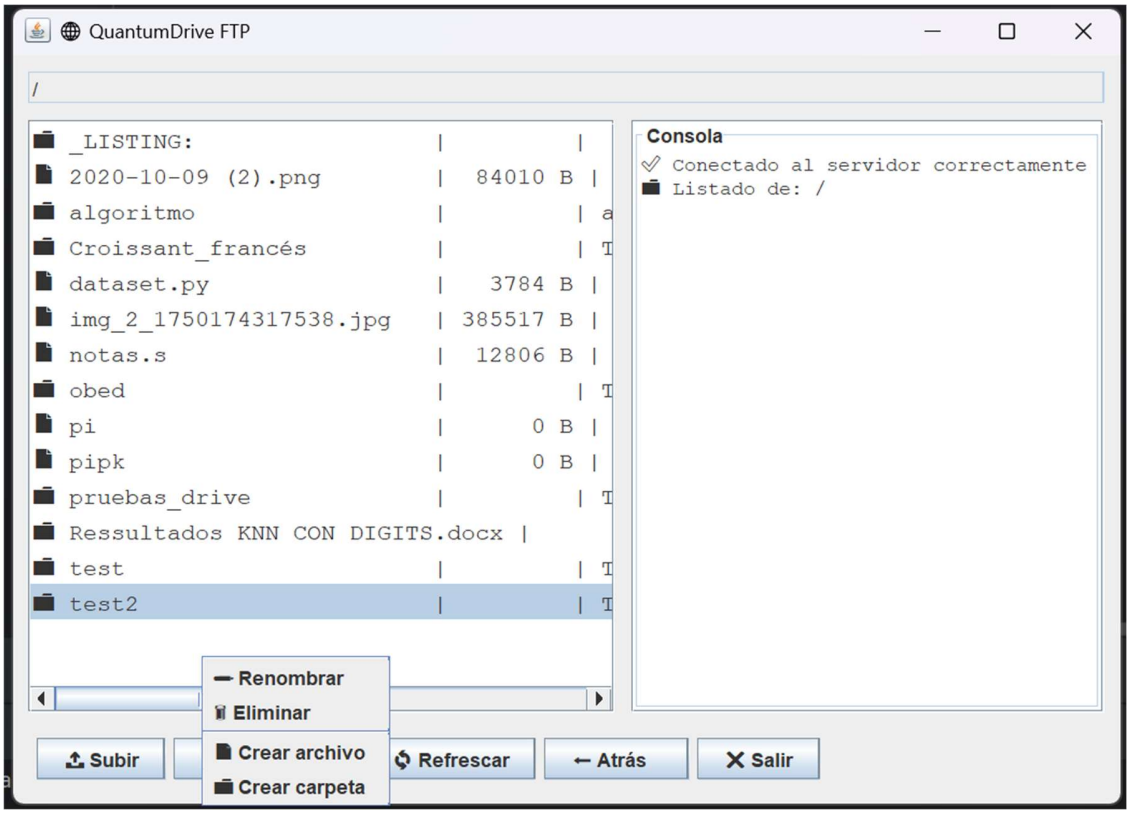
- compress(Path path): Recibe una ruta de archivo o carpeta y devuelve un arreglo de bytes (byte[]) que representa la versión comprimida en formato ZIP.
- decompress(byte[] data, Path destination): Toma un arreglo de bytes comprimido y lo descomprime en una ubicación destino del sistema de archivos.

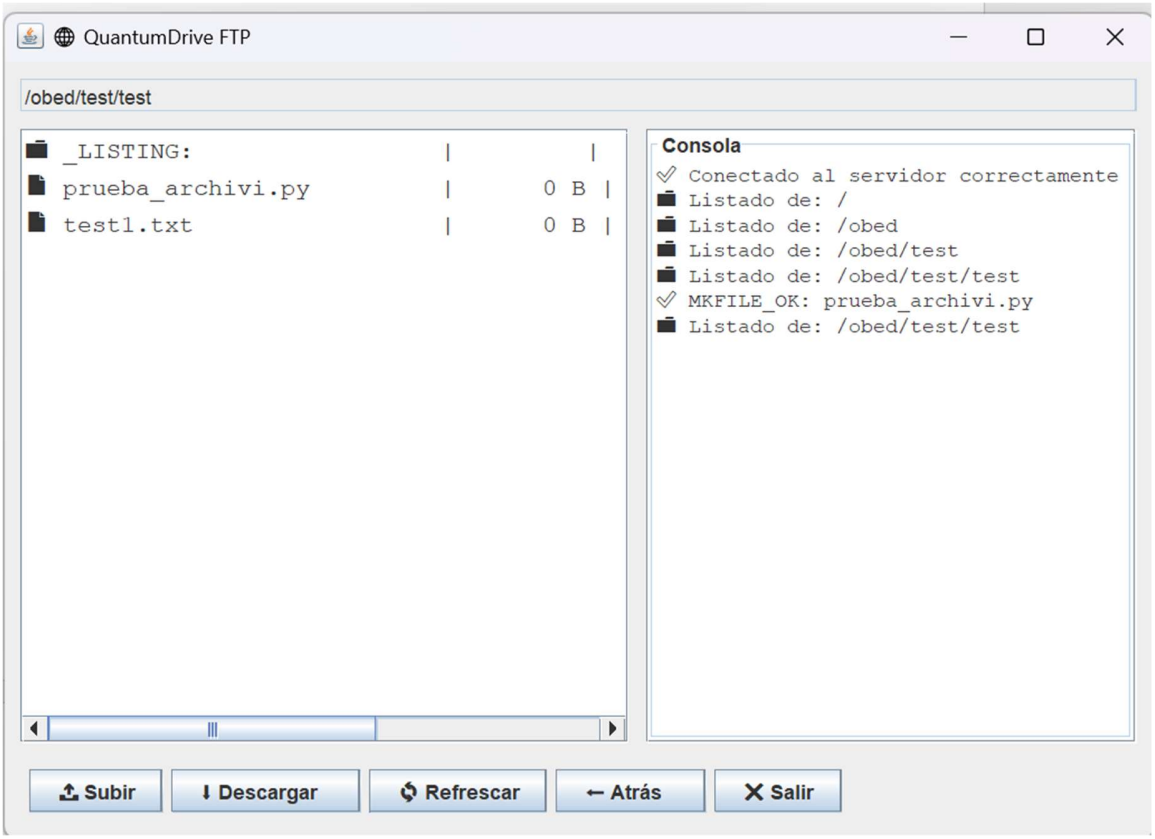
5. Interfaz Gráfica (FtpClientGUI.java):

- Proporciona una interfaz Swing para facilitar la interacción.
- Muestra el contenido de los directorios remotos y permite acciones con botones y clics.
- Permite subir o descargar archivos y carpetas de forma visual.



```
public class FtpClientGUI extends JFrame {  Yisus0x19

    private ClientControl client;  14 usages
    private DefaultListModel<String> fileListModel;  4 usages
    private JList<String> fileList;  9 usages
    private JTextArea console;  7 usages
    private JTextField pathField;  5 usages
    private List<String> rawDirEntries = new java.util.ArrayList<>();  8 usages


    public FtpClientGUI() {  6 usages  Yisus0x19
        super( title: "🌐 QuantumDrive FTP");
        setSize( width: 700,  height: 500);
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        initUI();
        connectAndInitialize();
    }

    private void showContextMenu(MouseEvent e, int index) {  1 usage  Yisus0x19
        JPopupMenu menu = new JPopupMenu();

        if (index >= 0 && index < rawDirEntries.size()) {
            String raw = rawDirEntries.get(index);
            boolean isDir = raw.startsWith("DIR");
            String name = extractName(raw, isDir ? "DIR" : "FILE");

            JMenuItem renameItem = new JMenuItem( text: "✏️ Renombrar");
            renameItem.addActionListener( ActionEvent ae -> renameEntry(name));

            JMenuItem deleteItem = new JMenuItem( text: "🗑 Eliminar");
            deleteItem.addActionListener( ActionEvent ae -> deleteEntry(name));

            menu.add(renameItem);
            menu.add(deleteItem);

            if (isDir) {
                menu.addSeparator();
```


QuantumDrive FTP
/
_LISTING: |
2020-10-09 (2).png | 84010 B |
algoritmo |
Croissant_francés |
dataset.py | 3784 B |
img_2_1750174317538.jpg | 385517 B |
notas.s | 12806 B |
obed |
pi | 0 B |
pipk | 0 B |
pruebas_drive |
Ressultados KNN CON DIGITS.docx |
test |
test2 |
Consola
Conectado al servidor correctamente
Listado de: /
Renombrar
Eliminar
Crear archivo
Crear carpeta
Subir
Refrescar
Atrás
Salir


QuantumDrive FTP
/obed/test/test
_LISTING: |
prueba_archivi.py | 0 B |
test1.txt | 0 B |
Consola
Conectado al servidor correctamente
Listado de: /
Listado de: /obed
Listado de: /obed/test
Listado de: /obed/test/test
MKFILE_OK: prueba_archivi.py
Listado de: /obed/test/test
Subir
Descargar
Refrescar
Atrás
Salir

Cada componente interactúa con los demás mediante protocolos preestablecidos sobre sockets TCP. Se manejan errores comunes como pérdida de conexión, archivos no encontrados y permisos restringidos, mejorando la robustez del sistema.

Conclusión

El desarrollo de QuantumDrive permitió comprender y aplicar conceptos fundamentales de la comunicación en redes, como el modelo cliente-servidor, los sockets TCP, la transferencia binaria y la manipulación del sistema de archivos remoto. A través del uso de sockets bloqueantes, se logró implementar un sistema funcional, aunque con ciertas limitaciones respecto a la concurrencia. Este trabajo también enfatizó la importancia del diseño modular y la interacción fluida entre componentes. La interfaz gráfica demostró ser una herramienta valiosa para mejorar la experiencia del usuario. En conjunto, esta práctica representa una base sólida para futuras implementaciones con tecnologías más avanzadas, como sockets no bloqueantes o asincronía mediante NIO.