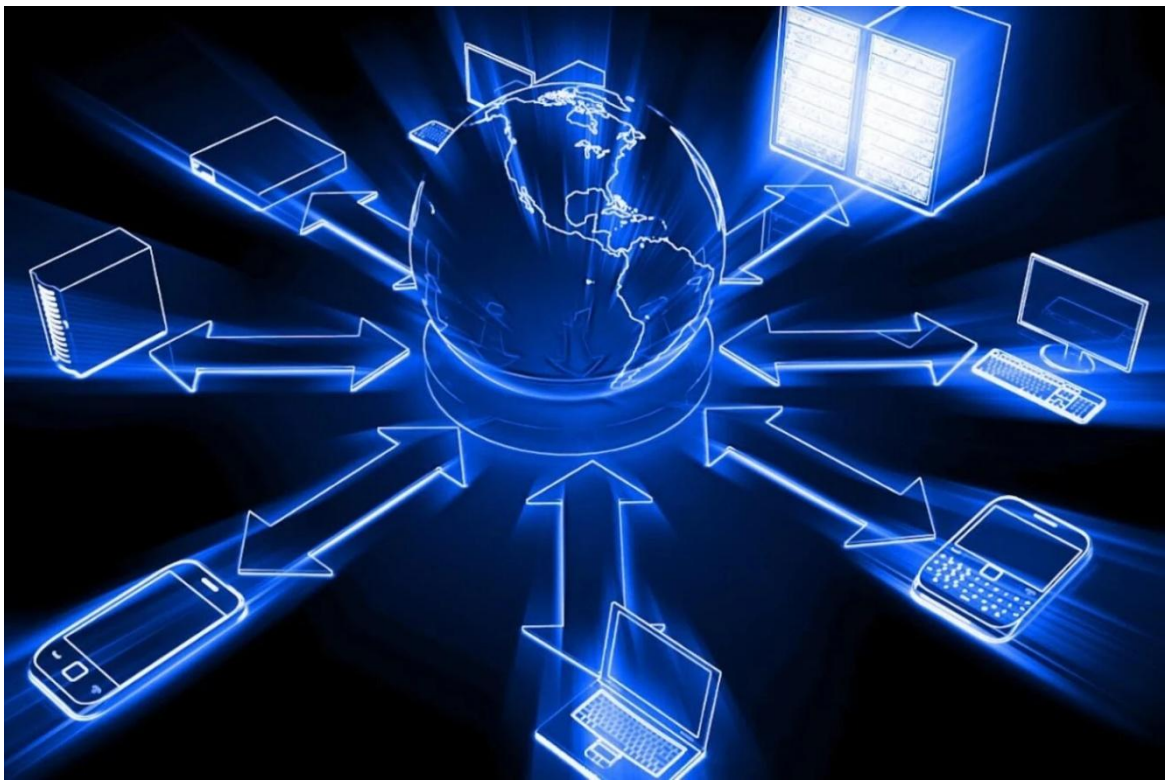
MEMORIA NANOFILES



ALUMNOS(G2.4):

Rafael Guillen García

1. INTRODUCCIÓN.

En este documento se detalla el diseño, la implementación y las mejoras del sistema **NanoFiles**, desarrollado como proyecto práctico para la asignatura de Redes de Comunicaciones (Grado en Ingeniería Informática, curso 2024/25). El objetivo principal del proyecto es diseñar y programar, en Java, los protocolos de comunicación necesarios para simular un sistema de compartición y transferencia de ficheros distribuido, compuesto por un **servidor de directorio** (implementado en Directory.java) y un conjunto de **peers** (implementados en NanoFiles.java).
El sistema NanoFiles se basa en dos modelos de comunicación:

- **Modelo cliente-servidor (UDP):** Utilizado para la comunicación entre cada peer y el servidor de directorio. El peer actúa como cliente, consultando los ficheros disponibles para descarga, publicando los ficheros que desea compartir y obteniendo la lista de servidores que comparten un determinado fichero.
- **Modelo peer-to-peer (TCP):** Permite la comunicación directa entre peers. Un peer puede actuar como cliente, solicitando información o descargando ficheros de otro peer que actúa como servidor de ficheros.

El usuario puede convertir cualquier peer en servidor de ficheros, eligiendo el puerto de escucha (por defecto 10000/tcp o uno efímero), de modo que otros peers puedan conectarse a él para obtener información o descargar ficheros.
El objetivo de la práctica es que el alumno sea capaz de diseñar y programar los protocolos de comunicación necesarios para que todas estas interacciones se realicen de forma fiable y eficiente, gestionando tanto la publicación y consulta de ficheros como la transferencia real de datos entre peers.
Las funcionalidades implementadas en este proyecto son las siguientes:

- **Comando ping.**
- **Comando filelist.**
- **Comando serve.**
- **Comando download.**
- **Comando upload.**
- **Comando quit.**
- **Comando myfiles.**
- **Comando help.**

Además, se han implementado varias mejoras opcionales, como el uso de puertos efímeros para el servidor, la ampliación del comando filelist para mostrar los servidores que comparten cada fichero y la actualización automática del directorio al salir.
A continuación, se describen en detalle los protocolos diseñados, los mensajes intercambiados, los autómatas de cada rol, las funcionalidades del proyecto y las mejoras implementadas, el manual de uso de cada comando y la conclusión.

2. PROTOCOLOS DISEÑADOS

- **Protocolo Cliente-Servidor**

Todos los mensajes de este protocolo son en formato ASCII campo:valor, con los campos separados por ":". El campo principal de cada mensaje es la operación, que indica la acción solicitada o la respuesta generada.

Tipos y descripción de los mensajes

Mensaje "**Solicitud de ping**"
- Operación: ping.
- Emisor: NanoFiles.
- Receptor: Directory.
- Campo(s) adicionales: identificador de protocolo del peer.
- Finalidad: pedir al directorio que compruebe si está activo y si el id de protocolo es compatible.
- Condiciones de envío: ninguna (se puede enviar en cualquier momento para comprobar compatibilidad).
- Acciones al recibirlo: el directorio compara el id de protocolo recibido con el tuyo, y responde con welcome si es igual, o denied si no lo es.
- Formato:

    operation: ping
    protocol: <protocolId>

Mensaje "**welcome**"
- Operación: welcome
- Emisor: Directory
- Receptor: NanoFiles
- Campos adicionales: ninguno
- Finalidad: informar de que el ping se ha producido correctamente y que el protocolId es el mismo que el del directorio.
- Condiciones de envío: se acaba de recibir un ping y es compatible.
- Acciones al recibirlo: el cliente muestra por consola que el directorio es compatible.
- Formato:

    operation: welcome

Mensaje "**ping denied**"
- Operación: denied
- Emisor: Directory
- Receptor: NanoFiles
- Campos adicionales: ninguno
- Finalidad: informar de que se ha producido el ping correctamente, pero el protocolId no es el mismo que el del directorio
- Condiciones de envío: haber recibido un ping y que el protocolId no sea el del directorio.
- Acciones al recibirlo: el cliente muestra por consola que el directorio no es compatible
- Formato:

    operation: denied

Mensaje "**Solicitud de help**"
- Operación: help
- Emisor: NanoFiles
- Receptor: Directory
- Campos adicionales: ninguno
- Finalidad: facilitar información sobre el uso del programa
- Condiciones de envío: ninguna
- Acciones al recibirlo: el directorio imprime por consola información sobre las órdenes que soporta el programa
- Formato:

    operation: help

Mensaje "**Solicitud de ficheros**"
- Operación: get_files
- Emisor: NanoFiles
- Receptor: Directory
- Campos adicionales: ninguno
- Finalidad: solicitar al directorio una lista de sus ficheros publicados y los servidores que los comparten
- Condiciones de envío: haber recibido un welcome.
- Acciones al recibirlo: el directorio responderá con mensaje filelist que incluye la lista de archivos y de servidores que comparten cada archivo
- Formato:

    operation: get_files

Mensaje “**filelist**” (respuesta)
- Operación: filelist
- Emisor: Directory
- Receptor: NanoFiles
- Campos adicionales: lista con archivos publicados con su nombre, tamaño, hash y lista de servidores que los comparten
- Finalidad: devolver una lista de archivos al cliente junto con los servidores que los comparten
- Condiciones de envío: haber recibido una solicitud get_files válida
- Acciones al recibirlo: Imprimir por consola la lista de ficheros publicados al directorio, indicando su nombre, tamaño, hash y una lista de servidores que lo están compartiendo.
- Formato:

    operation: get_files
    files:
    archivo1.txt;1024;hash1;192.168.1.10:10000,archivo2.pdf;2048;hash2;192.168.1.11:10000

Mensaje “**register_files**” (solicitud)
- Operación: register_files
- Emisor: NanoFiles
- Receptor: Directory
- Campos adicionales: puerto TCP en el que el peer servirá los archivos (port) y la lista de archivos a registrar (files).
- Finalidad: registrar los archivos que el peer está compartiendo y el puerto TCP en el que los sirve.
- Condiciones de envío: el cliente ha realizado un ping exitoso (welcome) y ejecuta el comando serve
- Acciones al recibirlo: el directorio registra los archivos del peer y responde con un mensaje de confirmación
- Formato:

    operation: register_files
    port: 10000
    files: archivo1.txt;1024;hash1,archivo2.pdf;2048;hash2

Mensaje "**register_files**" (respuesta)
- Operación: register_files
- Emisor: Directory
- Receptor: NanoFiles
- Campos adicionales: status: ok (o error en caso de fallo)
- Finalidad: confirmar al cliente que el registro de archivos se ha realizado correctamente
- Condiciones de envío: el directorio ha procesado correctamente la solicitud de registro de archivo
- Acciones al recibirlo: el cliente informa al usuario del éxito o error en el registro
- Formato:

    operation: register_files
    status: ok

Mensaje "**Solicitud de download**"
- Operación: download
- Emisor: NanoFiles
- Receptor: Directory
- Campos adicionales: filename (nombre o subcadena del fichero a descargar)
- Finalidad: solicitar la descarga de un fichero.
- Condiciones de envío: haber recibido un welcome
- Acciones al recibirlo: buscar en la base de datos y responde con un mensaje server_list con los servidores que tienen el archivo
- Formato:

    operation: download
    filename: <nombre o subcadena>

Mensaje "**server_list**" (respuesta a download)
- Operación: server_list
- Emisor: Directory
- Receptor: NanoFiles
- Campos adicionales: servers (lista de servidores solicitada)
- Finalidad: devolver la lista de servidores pedida.
- Condiciones de envío: haber recibido una solicitud download válida.
- Acciones al recibirlo: el cliente intenta conectarse a los servidores listados para descargar el archivo.
- Formato:

    operation: server_list
    servers: 192.168.1.10:10000,192.168.1.11:10000

Mensaje “**Solicitud de baja**”
- Operación: unregister
- Emisor: NanoFiles
- Receptor: Directory
- Campos adicionales: port (puerto TCP del peer que se da de baja).
- Finalidad: solicitud para dar de baja el servidor
- Condiciones de envío: haber registrado previamente el servidor con éxito.
- Acciones al recibirlo: el directorio da de baja el servidor del peer
- Formato:
  operation: unregister
  port: 10000

Mensaje de **error**
- Operación: error o denied (según el caso)
- Emisor: Directory
- Receptor: NanoFiles
- Campos adicionales: message (descripción del error)
- Finalidad: informar al cliente de un error en la operación solicitada.
- Condiciones de envío: la operación solicitada ha fallado.
- Acciones al recibirlo: el cliente muestra el error por consola
- Formato:
  operation: error
  message: <descripción>

- **Protocolo Cliente-Cliente (P2P)**

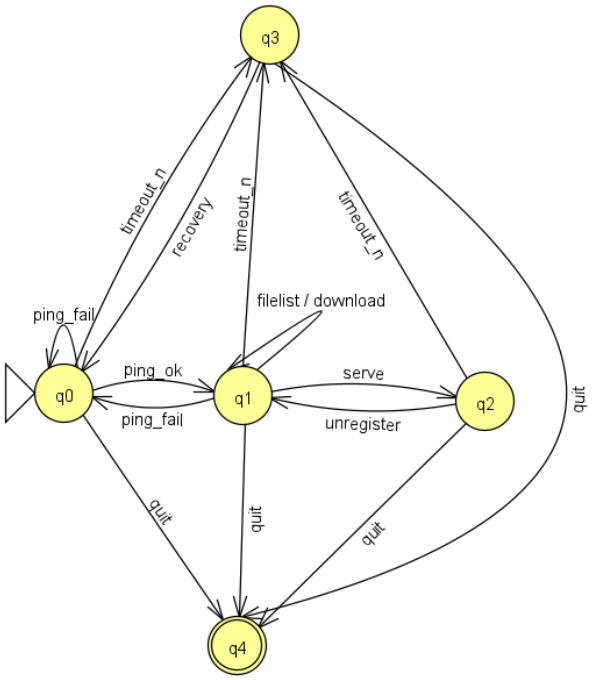
Este protocolo utiliza mensajes en formato binario, compuesto por un byte (opcode) que indica la operación.

NOMBRE DEL MENSAJE	OPCODE	EMISOR	RECEPTOR	CAMPOS ADICIONALES	FINALIDAD	CONDICIONES DE ENVÍO	ACCIONES AL RECIBIRLO
INVALIDAD_CODE	0x00	Ambos	Ambos	Ninguno	Indica mensaje inválido o desconocido	Error de codificación o recepción de mensaje no válido	Ignorar o cerrar conexión
FILEREQUEST	0x01	Cliente	Servidor	fileHash (20 Bytes)	Solicitar la descarga de un fichero identificado por su hash	El cliente quiere descargar un fichero	El servidor busca el fichero y responde con FILEREQUEST_ACCEPTED o FILE_NOT_FOUND
CHUNKREQUEST	0x02	Cliente	Servidor	fileHash (20 Bytes) offset (long) chunksize (int)	Solicitar un fragmento del fichero	Tras recibir FILEREQUEST_ACCEPTED	El servidor envía el chunk solicitado (CHUNK) o indica fuera de rango
STOP	0x03	Cliente	Servidor	Ninguno	Finalizar la transferencia de fichero o cancelar la operación	El cliente ha terminado la descarga o se cancela	El servidor cierra la conexión o termina la sesión
UPLOAD	0x04	Cliente	Servidor	fileHash (20 Bytes) fileSize (long)	Solicitar la subida de un fichero	El cliente quiere subir un fichero	El servidor acepta o rechaza la subida
FILENAME_TO_SAVE	0x05	Cliente	Servidor	fileName (TLV: longitud + nombre)	Enviar el nombre del fichero a guardar en el	Tras recibir UPLOAD	El servidor prepara la recepción del fichero

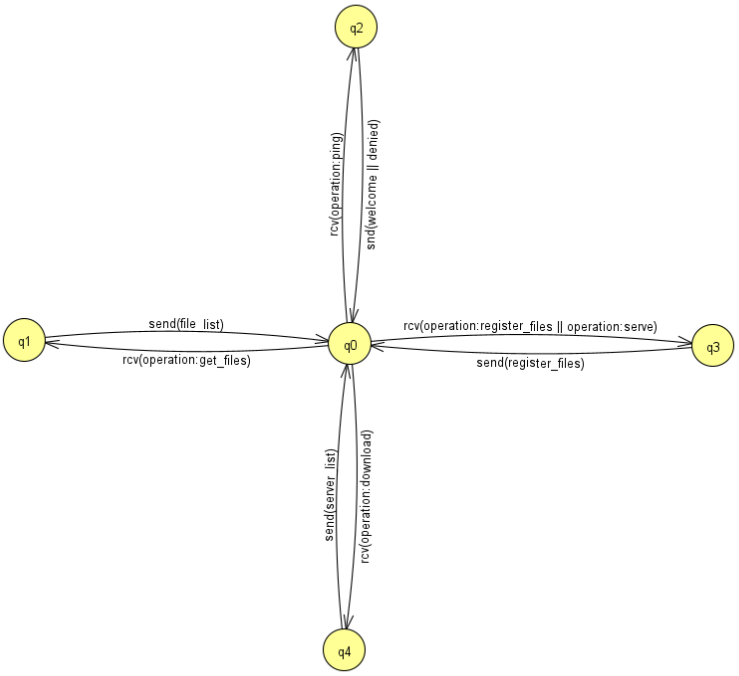
					servidor		
FILEREQUEST_ACCEPTED	0x11	Servidor	Cliente	fileSize (long) fileHash (20 Bytes)	Confirmar que el fichero está disponible y se puede descargar	Tras recibir FILEREQUEST y encontrar el fichero	El cliente puede solicitar chunks del fichero
FILE_NOT_FOUND	0x12	Servidor	Cliente	Ninguno	Informar de que el fichero solicitado no está disponible	Tras recibir FILEREQUEST y no encontrar el fichero	El cliente informa al usuario del error
CHUNK	0x13	Servidor	Cliente	Offset (long) chunkData (byte[]) chunkSize (int)	Enviar un fragmento del fichero	Tras recibir CHUNKREQUEST	El cliente escribe el chunk en el fichero local
CHUNKREQUEST_OUTOFRANGE	0x14	Servidor	Cliente	Ninguno	Indicar que el chunk solicitado está fuera del rango del fichero	Tras recibir CHUNKREQUEST fuera de rango	El cliente detiene la petición o solicita otro chunk
FILE_ALREADY_EXISTS	0x15	Servidor	Cliente	Ninguno	Informar de que el fichero ya existe en el servidor	Tras recibir UPLOAD y comprobar que ya existe el fichero	El cliente cancela la subida

3. Autómatas de protocolo

Autómata rol cliente de directorio

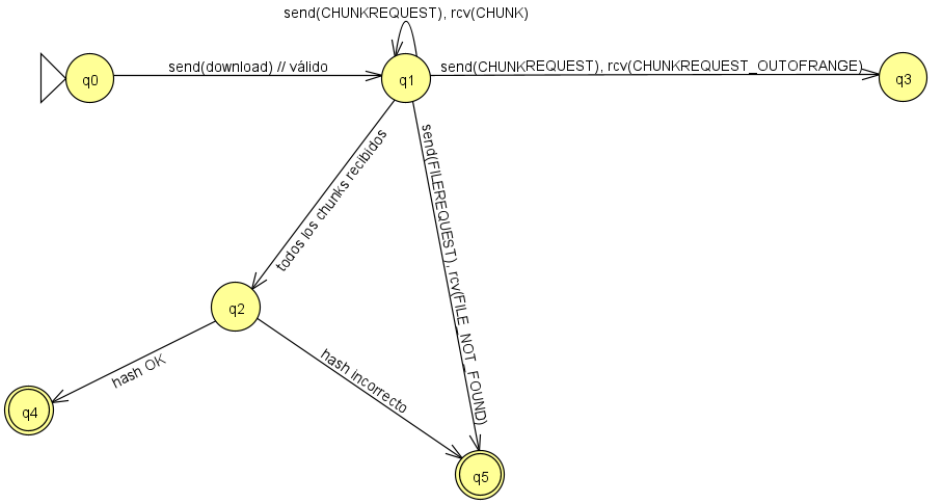


Autómata rol servidor de directorio

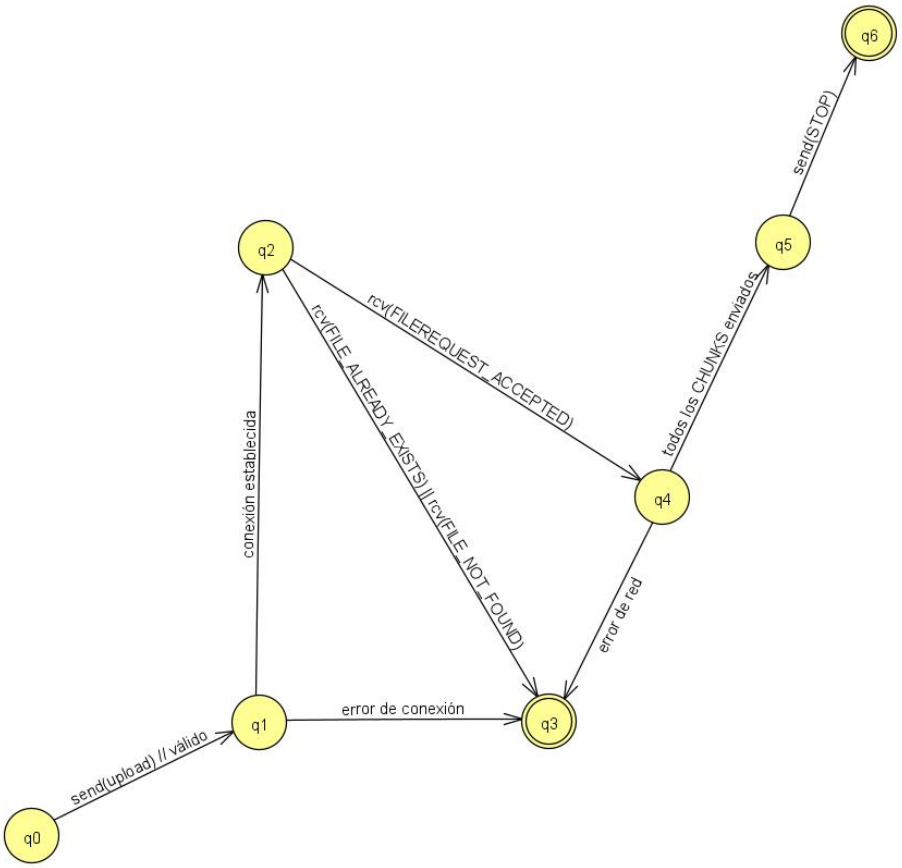


Autómata rol cliente de ficheros

(Comando Download)

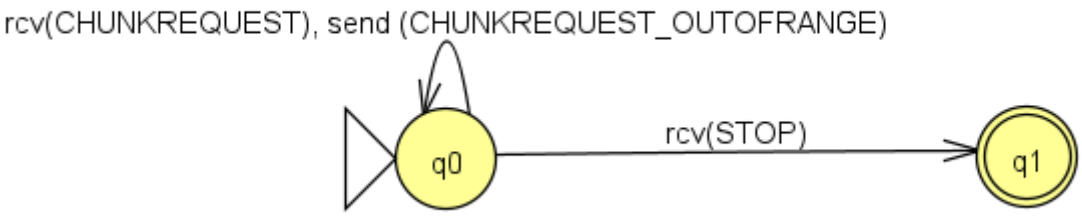


(Comando Upload)

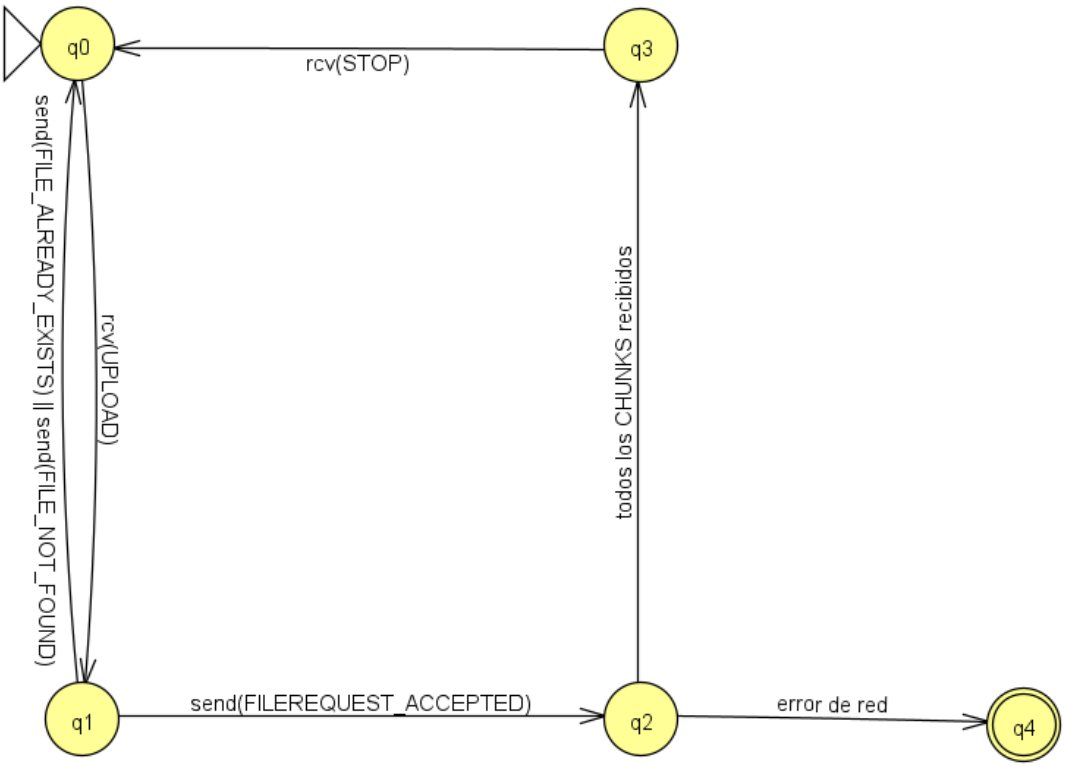


Autómata rol servidor de ficheros

(Comando Download)



(Comando Upload)



4. Ejemplo de intercambio de mensajes

- **Mensajes UDP (Cliente <-> Directorio)**

a) **Ping**

Cliente -> Directorio:
operation:ping
protocol:45536916

Directorio -> Cliente (éxito):
operation:welcome
status:ok

Directorio -> Cliente (fallo):
operation:denied
status:error
message:Incompatible protocol

b) **Solicitud de lista de ficheros**

Cliente -> Directorio:
operation:get_files

Directorio -> Cliente:
operation:file:list
status:ok
files:
documentodelservidor.txt;55;38d276c21563866401bf8360aba9e16d9af6a2bf;127.0.0.1:10000,estructurananofiles.png;72860;b37a6ce1636b49c6e0657926469ed7eedbb1e08d;127.0.0.1:10000ª

c) **Registro de archivos**

Cliente -> Directorio:

operation:register_files
port:10000
files:
documentodelservidor.txt;55;38d276c21563866401bf8360aba9e16d9af6a2bf,estructurananofiles.png;72860;b37a6ce1636b49c6e0657926469ed7eedbb1e08d

Directorio -> Cliente:

operation:register_files
status:ok

d) **Solicitud de Descarga**

Cliente -> Directorio:

operation:download
filename: documento.txt

Directorio -> Cliente:

operation:server_list
status:ok
servers:127.0.0.1:10000

e) **Solicitud de Baja**

Cliente -> Directorio

operation:register_files
port:10000

f) **Mensaje de Error:**

Directorio -> Cliente:

operation:denied
status:error
message: El archivo solicitado no existe

- **Mensajes TCP Binarios (Peer <-> Peer)**

a) **Solicitud de fichero**
   Cliente -> Servidor:
   [OPCODE_FILEREQUEST][fileHash]

b) **Fichero aceptado:**
   Servidor -> Cliente:
   [OPCODE_FILEREQUEST_ACCEPTED]

c) **Fichero no encontrado:**
   Servidor -> Cliente:
   [OPCODE_FILE_NOT_FOUND]

d) **Solicitud de chunk:**
   Cliente -> Servidor:
   [OPCODE_CHUNKREQUEST][fileHash][offset][chunkSize]

e) **Envío de chunk:**
   Servidor -> Cliente:
   [OPCODE_CHUNK][offset][chunkSize][chunkData]

f) **Chunk fuera de rango:**
   Servidor -> Cliente:
   [OPCODE_CHUNKREQUEST_OUTOFRANGE]

g) **Solicitud de subida:**
   Cliente -> Servidor:
   [OPCODE_UPLOAD][fileHash]

h) **Nombre de fichero para guardar:**
   Cliente -> Servidor:
   [OPCODE_FILENAME_TO_SAVE][fileNameLength][fileName]

i) **Fichero ya existe:**
   Servidor -> Cliente:
   [OPCODE_FILE_ALREADY_EXISTS]

j) **Parada de la transferencia:**
   Cliente -> Servidor:
   [OPCODE_STOP]

- **Ejemplo de secuencia completa de download**

1. **Cliente -> Servidor:**
   [FILEREQUEST][fileHash]

2. **Servidor -> Cliente:**
   [FILEREQUEST_ACCEPTED]

3. **Cliente -> Servidor:**
   [CHUNKREQUEST][fileHash][offset][chunkSize]

4. **Servidor -> Cliente:**
   [CHUNK][offset][chunkSize][chunkData]

5. **Repetir 3-4 hasta terminar.**
6. **Cliente -> Servidor:**
   [STOP]

- **Ejemplo de secuencia completa de upload**

1. **Cliente -> Servidor:**
   [UPLOAD][fileHash]

2. **Servidor -> Cliente:**
   [FILEREQUEST ACCEPTED] o [FILE_ALREADY_EXISTS]

3. **Cliente -> Servidor:**
   [FILENAME_TO_SAVE][fileNameLength][fileName]

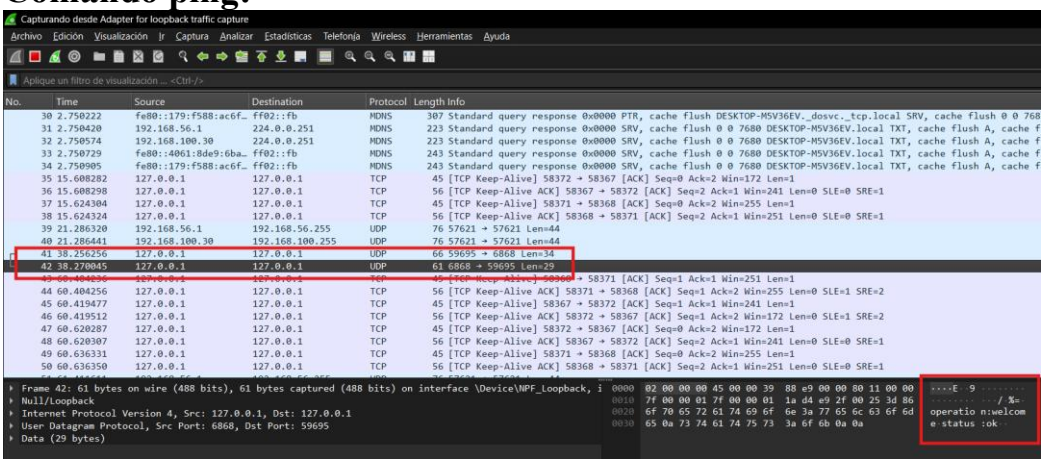
4. **Servidor -> Cliente:**
   [FILEREQUEST_ACCEPTED]

5. **Cliente -> Servidor:**
   [CHUNK][offset][chunkSize][chunkData] (repetir por cada chunk)

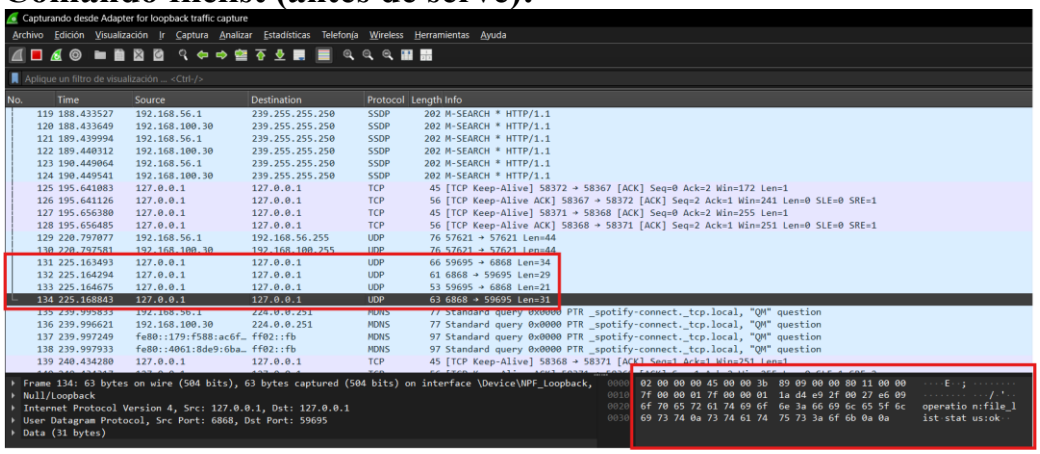
6. **Cliente -> Servidor:**
   [STOP]

5. Capturas Wireshark

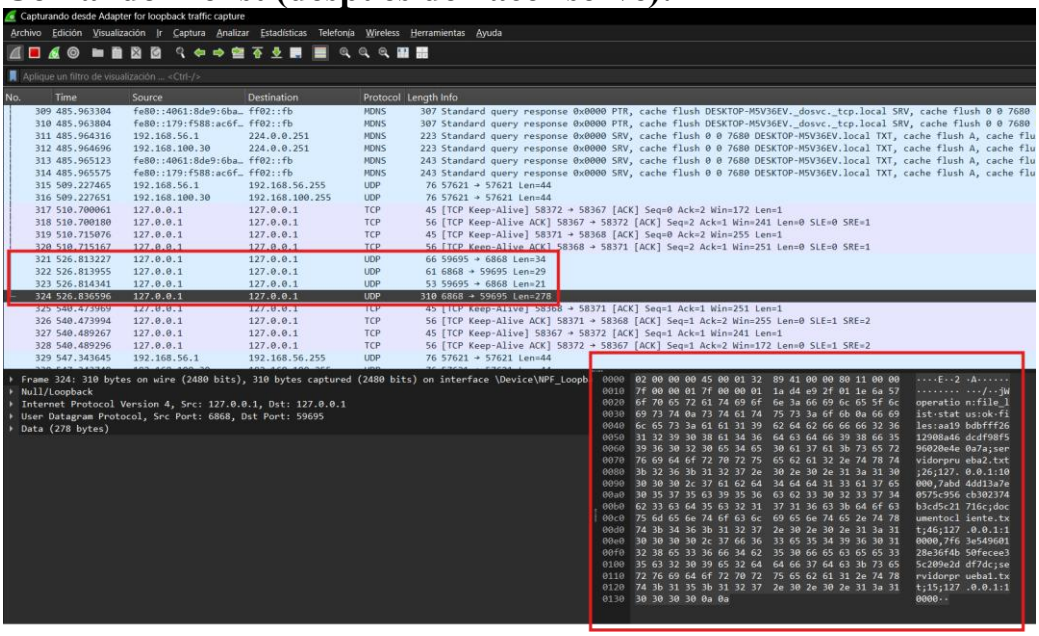
- **Comando ping:**



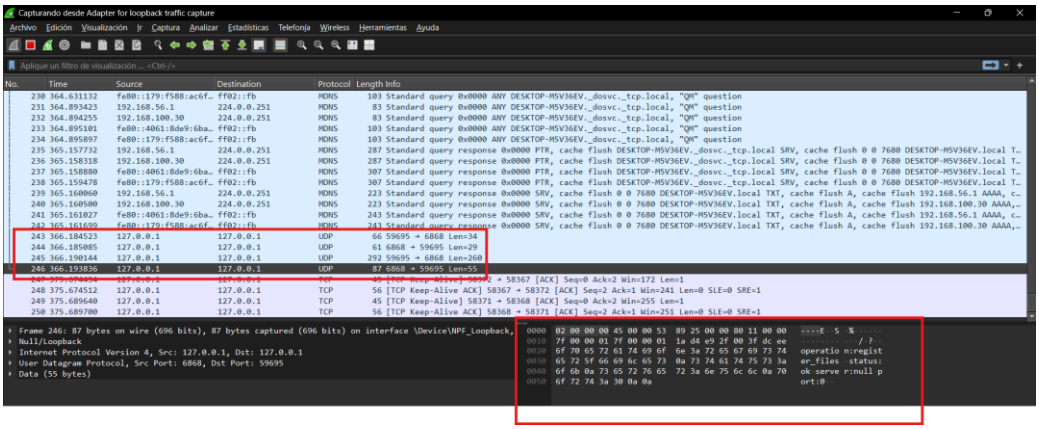
- **Comando filelist (antes de serve):**



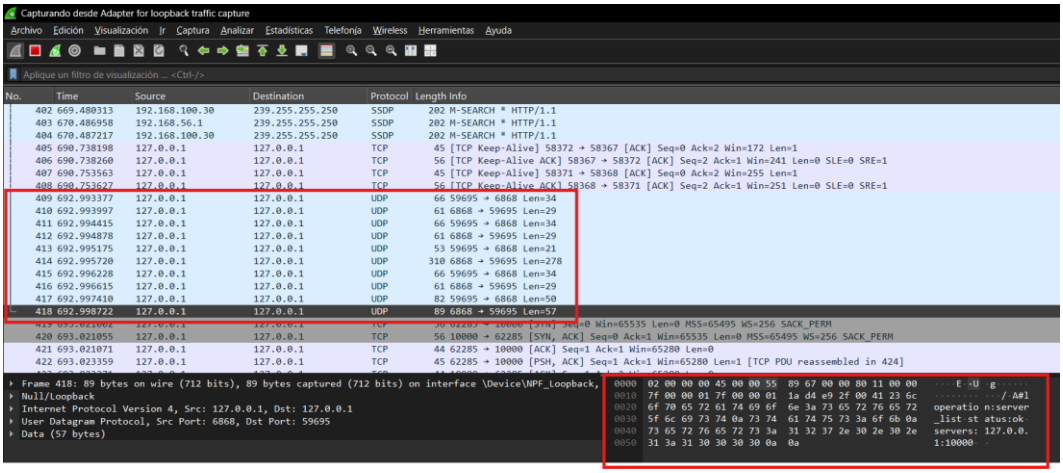
- **Comando filelist (después de hacer serve):**



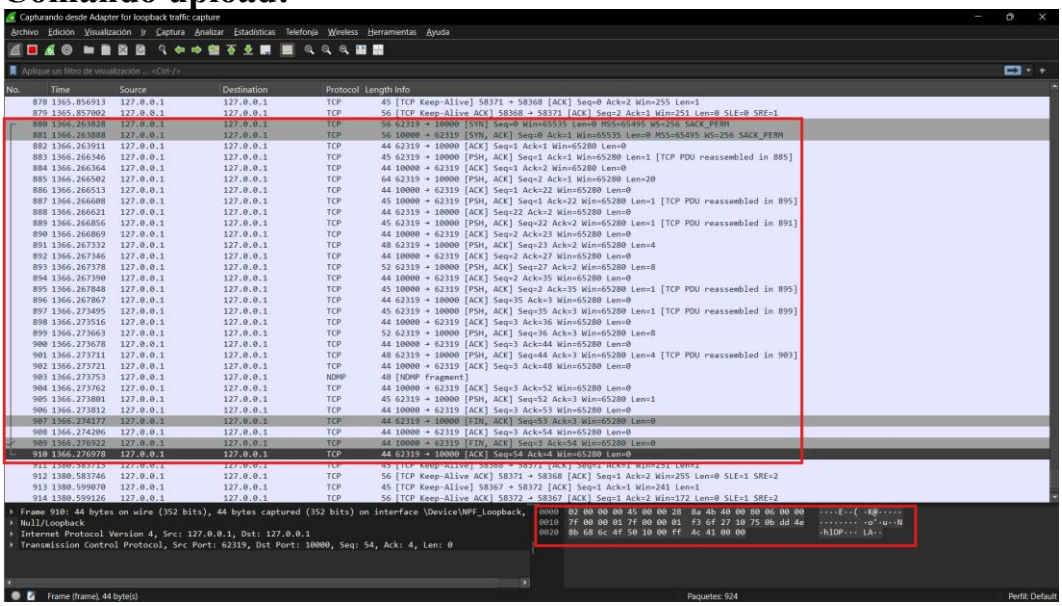
- **Comando serve:**



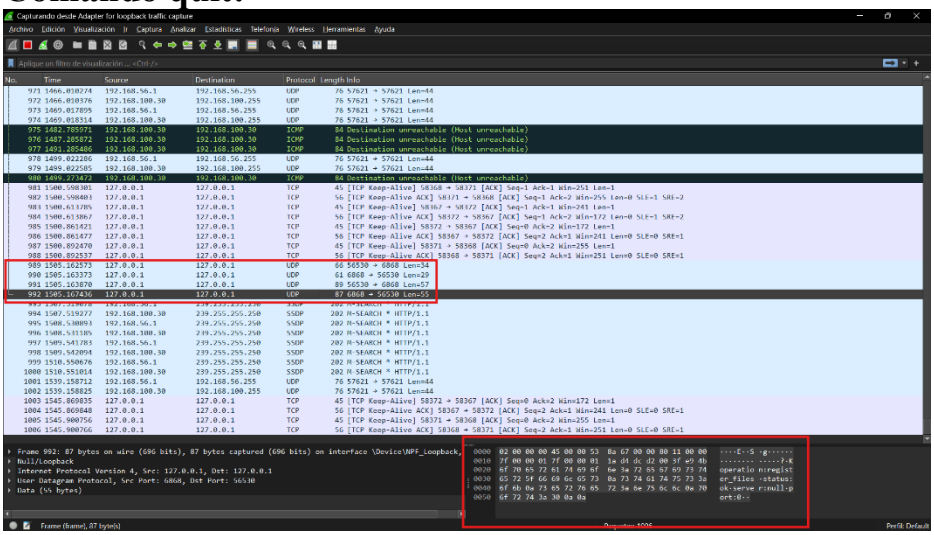
- **Comando download:**



- **Comando upload:**



- **Comando quit:**



6. FUNCIONALIDADES IMPLEMENTADAS

- Funcionalidades Básicas

1. **Comando ping.**
   Permite a un peer comprobar que el servidor de directorio está activo y que utiliza un protocolo compatible. El peer envía mensaje ping y espera la respuesta welcome o denied del directorio.
   Uso: verifica la disponibilidad y compatibilidad antes de cualquier operación

   1º Peer: Envia protocolo UDP -> operation:ping
   2º Directory: Comprueba el protocolId y responde con operation:welcome u operation:denied
   3º Peer: muestra el resultado por la consola

2. **Comando filelist**
   El peer solicita al directorio la lista de ficheros publicados y disponibles para descarga. El directorio responde con la lista de archivos y sus metadatos.
   Uso: permite al usuario conocer qué archivos están disponibles en la red.

   1º Peer: Envia protocolo UDP -> operation:get_files
   2º Directory: Responde con operation:file_list y la lista de archivos publicados, que incluye el nombre, tamaño, hash y lista de servidores (en el formato IP:Puerto) que comparten cada fichero
   3º Peer: imprime la lista por la consola

3. **Comando serve.**
   Lanza un servidor de ficheros en el peer, que escucha en un puerto TCP (por defecto 10000). El peer registra automáticamente sus archivos en el directorio.
   Uso: permite que otros peers puedan conectarse y descargar archivos de este peer.

   1º Peer: lanza el servidor TCP en un hilo y envía el protocolo UDP -> operation:register_files
   2º Directory: añade el peer y sus archivos a la base de datos y responde con status:ok

4. **Comando download.**

   Permite descargar un fichero identificado por una subcadena de su nombre. El peer consulta al directorio qué servidores tienen el archivo y descarga fragmentos (chunks) de ellos.
   Uso: facilita la transferencia de archivos entre peers.

   1º Peer: envía el protocolo UDP -> operation:download con el campo filename (subcadena o nombre completo)
   2º Directory: busca los archivos que coincidan y responde con operation:server_list, status:ok y la lista de IP:Puerto de los Peers que tienen el archivo.
   3º Peer:
   - Elige uno de los servidores de la lista.
   - Establece la conexión TCP con el peer que ha elegido.
   - Envía [FILEREQUEST][fileHash]
   - El servidor responde con PeerMessageOps.FILEREQUEST_ACCEPTED (si tiene el archivo) o FILE_NOT_FOUND.
   - Si es accepted, el cliente solicita los fragmentos con [CHUNKREQUEST][fileHash][offset][chunkSize]
   - El servidor responde con CHUNK[offset][chunkSize][chunkData]
   - El cliente escribe cada chunk en el archivo local.
   - Repite hasta completar el archivo.
   - Al terminar envía [STOP] y cierra la conexión.
   - Finalmente, el cliente calcula el hash del archivo descargado y lo compara con el hash original para verificar la integridad.

5. **Comando quit.**

   Sale del programa de forma ordenada, asegurando que el peer se da de baja correctamente en el directorio si estaba sirviendo archivos.

   1º Peer: si había hecho serve, envía el protocolo UDP -> operation:register_files al directorio con su puerto TCP y el campo files vacío
   2º Directory elimina el peer y sus archivos de la base de datos, responde con operation:register_files y status:ok
   3º Peer: cierra sus hilos y termina el proceso

6. **Comando myfiles.**
   Muestra los archivos que el peer local está compartiendo, es decir, los archivos que se encuentran en la carpeta configurada al iniciar NanoFiles (nf-shared por defecto)
   Uso: obtener una lista con el nombre, tamaño y hash de cada archivo local disponible para compartir.

   Peer: lee su carpeta y muestra los archivos.

7. **Comando help.**
   Muestra la lista de comandos disponibles en el Shell de NanoFiles, junto con una breve descripción de cada uno.
   Uso: para recordar cómo funciona cada comando.

   Peer: muestra la lista de comandos y su descripción.

- ## Mejoras Implementadas

1. Comando serve con puerto efímero.

El comando serve permite ahora especificar el puerto como argumento. Si no se indica puerto, se utiliza el valor por defecto (10000). Si el usuario indica el valor 0, se asigna automáticamente un puerto efímero.
La clase NFShell se encarga de leer el comando serve y sus argumentos introducidos por el usuario, incluyendo el puerto (o 0 para efímero). Esta información se pasa a la lógica P2P (NFControllerLogicP2P), que lanza el servidor de ficheros (NFServer) en el puerto indicado. El constructor de NFServer acepta el puerto como parámetro. Si se le pasa 0, el servidor se inicializa en un puerto efímero asignado por el sistema operativo. Para conocer el puerto real en el que está escuchando el servidor, se utiliza el método getPort() de NFServer, que devuelve el puerto finalmente asignado. A continuación, la lógica P2P (NFControllerLogicP2P) comunica este puerto real a la lógica de directorio (NFControllerLogicDir), que lo registra en el directorio mediante el método registerFileServer, junto con la lista de archivos compartidos.

2. Filelist ampliado: mostrar servidores por fichero.

El comando filelist ha sido ampliado para mostrar, además de la lista de archivos publicados en el directorio, qué peers (servidores) están compartiendo cada archivo.
La clase NFShell se encarga de leer el comando filelist introducido por el usuario, que se pasa a la lógica de directorio (NFControllerLogicDir), que es la responsable de solicitar la lista de archivos al directorio mediante el método getAndPrintFileList(), que a su vez utiliza DirectoryConnector, que envía una petición UDP al directorio solicitando la lista de archivos.
El directorio (NFDirectoryServer) responde con un mensaje que inclute, para cada archivo, su hash, nombre, tamaño y lista de servidores (IP:Puerto) que lo comparten.
Al recibir la respuesta, DirectoryConnector construye un array de

objetos FileInfo, donde cada uno contiene los metadatos (nombre, tamaño, hash y servidores que lo comparten) del archivo. La lista de servidores se almacena en el atributo serversList de FileInfo. Finalmente, el método getAndPrintFileList() de NFControllerLogicDir llama a FileInfo.printToSysout(), que imprime por pantalla una tabla con la información de cada archivo, incluyendo la columna de servidores.

3. **Comando quit actualiza ficheros y servidores.**

Se ha implementado la actualización automática del directorio, eliminando la información del servidor de ficheros y los archivos que el peer compartía, al ejecutar el comando quit.
Cuando el usuario introduce el comando quit, el Shell (NFShell) traduce este texto al código de comando NFCommands.COM_QUIT.
En la clase NFController, el método processCommand() detecta que el comando actual es quit e invoca la lógica necesaria para actualizar el directorio y detener el servidor de ficheros si está activo.
El controlador utiliza la lógica de Directorio (NFControllerLogicDir) para comunicar al directorio que el peer ya no va a servir archivos. Esto se realiza mediante el método unregisterFileServer, que envía el mensaje unregister al directorio mediante el protocoloUDP. Este mensaje incluye la dirección y puerto del servidor y los archivos que deben eliminarse del registro global.
El directorio, al recibir este mensaje, elimina el servidor y los archivos asociados de sus estructuras internas.
Si el peer estaba actuando como servidor de ficheros, la lógica P2P (NFControllerLogicP2P) se encarga de detener el servidor (NFServer) y liberar el puerto ocupado, mediante el método terminate() de la clase NFSserver, que cierra el socket y detiene la aceptación de nuevas conexiones.
Una vez ha realizado todo lo anterior, el método shouldQuit() de NFController devuelve true, provocando la salida del bucle principal y la impresión del mensaje de despedida (“Bye.”).

4. Comando upload: subida de archivos entre peers.

Se ha implementado el comando upload, que permite a un peer enviar un fichero local a otro peer servidor, siempre que este no posea ya una copia del fichero.
Cuando el usuario introduce el comando upload seguido del nombre de un archivo, el Shell (NFShell) interpreta este texto y lo traduce a NFCommands.COM_UPLOAD, junto con el nombre del archivo indicado como argumento.
En la clase NFController, el método processCommand() detecta que el comando actual es upload e invoca la lógica necesaria para compartir el archivo. Este método verifica que el peer esté actuando como servidor de ficheros.
El controlador utiliza la lógica de Directorio (NFControllerLogicDir) para registrar el nuevo archivo en el directorio global. Para ello, llama al método registerFile(), que se encagar de calcular la información relevante del archivo (metadatos) y construir y enviar un mensaje de register al directorio mediante el protocolo UDP, incluyendo la dirección y puerto del servidor, así como los metadatos del archivo.
El directorio, al recibir este mensaje, añade el archivo y la referencia al servidor y a sus estructuras internas, permitiendo que otros peers vean y descarguen el archivo recién compartido.
Localmente, la lógica P2P (NFControllerLogicP2P) y el servidor de ficheros (NFServer) actualizan su base de datos compartidos, asegurando que el archivo esté disponible para futuras peticiones de descarga.
Finalmente, el método processCommand() informa del resultado de la operación (éxito o error).

7. MANUAL DE USO

Cómo utilizar cada comando: los argumentos entre “[“ y “]” significa que son opcionales, los argumentos enter “<” y “>” son obligatorios.

- **comando help:** help
  Ejemplo: help

- **comando ping:** ping
  Ejemplo: ping

- **comando filelist:** filelist
  Ejemplo: filelist

- **comando serve:** serve [puerto]
  Ejemplos: serve || serve 12345 || serve 0

- **comando download:** download <subcadena_nombre> <nombre_para_guardar>
  Ejemplo: download docum documentodescargado.txt

- **comando upload:** upload <subcadena_nombre_local> <ip:puerto_destino> (ip puede ser localhost)
  Ejemplo: upload documento.txt 127.0.0.1:10000

- **comando myfiles:** myfiles
  Ejemplo: myfiles

- **comando quit:** quit
  Ejemplo: quit

8. CONCLUSIÓN

Sinceramente, nos ha parecido un proyecto de una dificultad bastante alta. Pero a su vez es bastante útil e interesante a la hora de nuestro desarrollo como informáticos y programadores. Además, nos ha permitido aprender más a fondo los conceptos dados en clase.
Nos hemos peleado bastante con Java y con los protocolos UDP y TCP, pero gracias a eso ahora entendemos mucho mejor cómo funciona la comunicación entre programas en red y también hemos visto lo importante que es diseñar bien los mensajes y controlar los posibles errores.
Por último, queremos recalcar que hemos hecho uso de Inteligencia Artificial como apoyo puntual para resolver dudas, aclarar conceptos y obtener ayuda sobre el funcionamiento de algunas partes de código, pero no hemos utilizado la IA para la generación de código, puesto que pensamos que es una herramienta muy útil si se usa con el fin de aprender y de mejorar como informáticos.