ESTRUCTURA PROYECTO REDES 2024/25

PABLO MEROÑO HERRANZ
HUGO EGEA JARA

Grupo:2.4
Profesor: JOSÉ RUBÉN TITOS GIL

ÍNDICE:

1. Introducción.

En este documento se especifica el diseño de dos protocolos de comunicación de nivel de aplicación del sistema NanoFiles: uno para la comunicación entre cada peer y el directorio, que será un protocolo confiable semi-duplex basado en parada y espera, que operará sobre un protocolo de nivel de transporte no confiable como UDP; y otro para la comunicación entre dos peers, sabiendo que el protocolo de nivel de transporte es confiable (TCP).

Las mejoras que hemos considerado a la hora de diseñar el protocolo son las de upload, serve con puerto efímero, filelist ampliado con servidores y quit que actualiza ficheros y servidores.

2. Formato de los mensajes del protocolo de comunicación con el Directorio

Para definir el protocolo de comunicación con el *Directorio*, vamos a utilizar mensajes textuales con formato "campo:valor". El valor que tome el campo "operation" (código de operación) indicará el tipo de mensaje y por tanto su formato (qué campos vienen a continuación).

Tipos y descripción de los mensajes

Mensaje: **PING**
Sentido de la comunicación: Cliente→ Directorio

Descripción: Este mensaje lo envía el cliente de NanoFiles al Directorio para comprobar que el servidor de directorio está activo y usa un protocolo compatible con el del peer

operation:ping\n
protocol:<protocolID>\n
\n

Mensaje: **SUCCESSFUL**
Sentido de la comunicación: Directorio→ Cliente

Descripción: Este mensaje lo envía el Directorio al cliente de NanoFiles como mensaje de confirmación para comunicar que el ping ha sido exitoso y que el cliente se ha registrado como servidor y se han subido sus ficheros cuando ejecuta serve. También para indicar que un servidor de ficheros se ha dado de baja con éxito.

operation:successful\n
\n

Mensaje: **ERROR**
Sentido de la comunicación: Directorio→ Cliente

Descripción: Este mensaje lo envía el Directorio al cliente de NanoFiles como mensaje de error para comunicar que el ping ha fallado, que un cliente no puede descargar ficheros porque no ha subido antes los suyos, y para comunicar que no se ha podido registrar al cliente como servidor y por tanto no se han subido sus ficheros.

operation:Error\n
\n

Mensaje: **FILELIST**
Sentido de la comunicación: Cliente→ Directorio

Descripción: Este mensaje lo envía el cliente de NanoFiles al Directorio para obtener la lista de ficheros que los peers servidores han publicado al directorio.

operation:filelist\n
\n

Mensaje: **AVAILABLE_FILES**
Sentido de la comunicación: Directorio→ Cliente

Descripción: Este mensaje lo envía el Directorio al cliente de NanoFiles como respuesta al mensaje FILELIST. Envía la lista de ficheros que los peers servidores han publicado al directorio con información de cada fichero como su nombre, tamaño, hash y los peers servidores que lo tienen.

operation:availableFiles\n
filename:<filename> \n
size:<file size>\n
hash:<file hash> \n
servers:<list of servers>\n
…
\n

Mensaje: **SERVE**
Sentido de la comunicación: Cliente→ Directorio

Descripción: Este mensaje lo envía el cliente de NanoFiles al Directorio para publicar los ficheros que el peer sirve al resto. Para cada fichero se indica el nombre, tamaño y hash.

Ejemplo:
operation:serve\n
port:<port number>\n
filename:<filename>\n
size:<file size>\n
hash:<file_hash> \n
…
\n

Mensaje: **DOWNLOAD**
Sentido de la comunicación: Cliente→ Directorio

Descripción: Este mensaje lo envía el cliente de NanoFiles al Directorio para indicar que quiere descargar un fichero cuyo nombre contiene la subcadena indicada en el campo filename.

operation:download\n
filename:<subcadena>\n
\n

Mensaje: **SERVERS_SHARING_THIS_FILES**
Sentido de la comunicación: Directorio→ Cliente

Descripción: Este mensaje lo envía el Directorio al cliente de NanoFiles como respuesta al mensaje DOWNLOAD. Envía la lista de ficheros que los peers servidores han publicado al directorio cuyo nombre contiene la subcadena indicada en el mensaje download. Para cada fichero se indica su nombre y el servidor que lo tiene.

operation:serversSharingThisFile\n
filename:<filename>\n
server:<server>\n
…
\n

Mensaje: **UNREGISTER**
Sentido de la comunicación: Cliente→ Directorio

Descripción: Este mensaje lo envía el cliente de NanoFiles al Directorio para indicar que quiere darse de baja como servidor de ficheros en el puerto indicado.

operation:unregister\n
port:<port number>\n
/n

3. Formato de los mensajes del protocolo de transferencia de ficheros

Para definir el protocolo de comunicación con un servidor de ficheros, vamos a utilizar mensajes binarios multiformato. El valor que tome el campo "opcode" (código de operación) indicará el tipo de mensaje y por tanto cuál es su formato, es decir, qué campos vienen a continuación.

Tipos y descripción de los mensajes

Mensaje: **DownLoadFile (opcode = 1)**
Sentido de la comunicación: Cliente → Servidor de ficheros
Descripción: Este mensaje lo envía el par cliente al par servidor de ficheros (receptor) para indicar una subcadena del nombre del fichero que quiere descargar.
Ejemplo:

Opcode (1 byte)	Longitud (1 byte)	Valor (n bytes)
0x01	**0x18**	**ejerciciosDeProgramación**

Mensaje: **FileFound (opcode = 2)**
Sentido de la comunicación: Servidor de ficheros → Cliente
Descripción: Este mensaje lo envía el par servidor de ficheros al par cliente (receptor) para indicar que ha encontrado el fichero del mensaje de petición de descarga.
Ejemplo:

Opcode (1 byte)	Hash (SHA-1) (20 bytes)	FileSize (Long: 8 byte)
0x02	3c2727cbe32f267231fa1b2d04aa905a0e4c21d9	**0x00 0x00 0x00 0x00 0x00 0x20 0x01 0x00**

Mensaje: **FileNotFound (opcode = 3)**
Sentido de la comunicación: Servidor de ficheros → Cliente
Descripción: Este mensaje lo envía el par servidor de ficheros al par cliente (receptor) para indicar que no ha encontrado ningún fichero cuyo nombre contenga la subcadena del mensaje de petición de descarga.
Ejemplo:

Opcode (1 byte)
0x03

Mensaje: **AmbiguousName (opcode = 4)**
Sentido de la comunicación: Servidor de ficheros → Cliente
Descripción: Este mensaje lo envía el par servidor de ficheros al par cliente (receptor) para indicar que la subcadena del mesaje de petición de descarga es ambiguo pues coincide con el nombre de varios.
Ejemplo:

Opcode (1 byte)
0x04

Mensaje: **GetChunk (opcode = 5)**
Sentido de la comunicación: Cliente → Servidor de ficheros
Descripción: Este mensaje lo envía el par cliente al par servidor de ficheros (receptor) para indicar desde qué posición y que tamaño de porción quiere descargar de un fichero previamente acordado.

Ejemplo:

Opcode (1 byte)	**File offset (Long: 8 byte)**	**Chunk size (Int: 4 bytes)**
0x05	**0x00 0x00 0x00 0x00 0x00 0x20 0x01 0x00**	**0x01 0x00 0x00 0x00**

Mensaje: **GiveChunk (opcode = 6)**
Sentido de la comunicación: Servidor de ficheros → Cliente y Cliente → Servidor de ficheros
Descripción: Este mensaje lo envía el par servidor de ficheros al par cliente (receptor) para enviarle trozos con el tamaño indicado del fichero acordado en el mensaje de descarga. También lo puede enviar el cliente al servidor para subir los trozos de un fichero acordado en el mensaje de subida.
Ejemplo:

Opcode (1 byte)	**Longitud (Int: 4 bytes)**	**Valor (n bytes)**
0x06	**0x00 0x10 0x00 0x00**	**…**

Mensaje: **UploadFile (opcode = 7)**
Sentido de la comunicación: Cliente → Servidor de ficheros
Descripción: Este mensaje lo envía el par cliente al par servidor de ficheros (receptor) para indicar una subcadena del nombre del fichero que quiere subir.
Ejemplo:

Opcode (1 byte)	**Longitud (1 byte)**	**Valor (n bytes)**
0x07	**0x18**	**ejerciciosDeProgramación**

Mensaje: **CloseConnection (opcode = 8)**
Sentido de la comunicación: Servidor de ficheros → Cliente
Descripción: Este mensaje lo envía el par cliente al par servidor de ficheros (receptor) cuando ya ha recibido todos los chunks necesarios y así cortar la conexión. También se usa cuando un par cliente está subiendo un fichero a otro cliente y ya ha terminado de enviar todos los chunks.
Ejemplo:

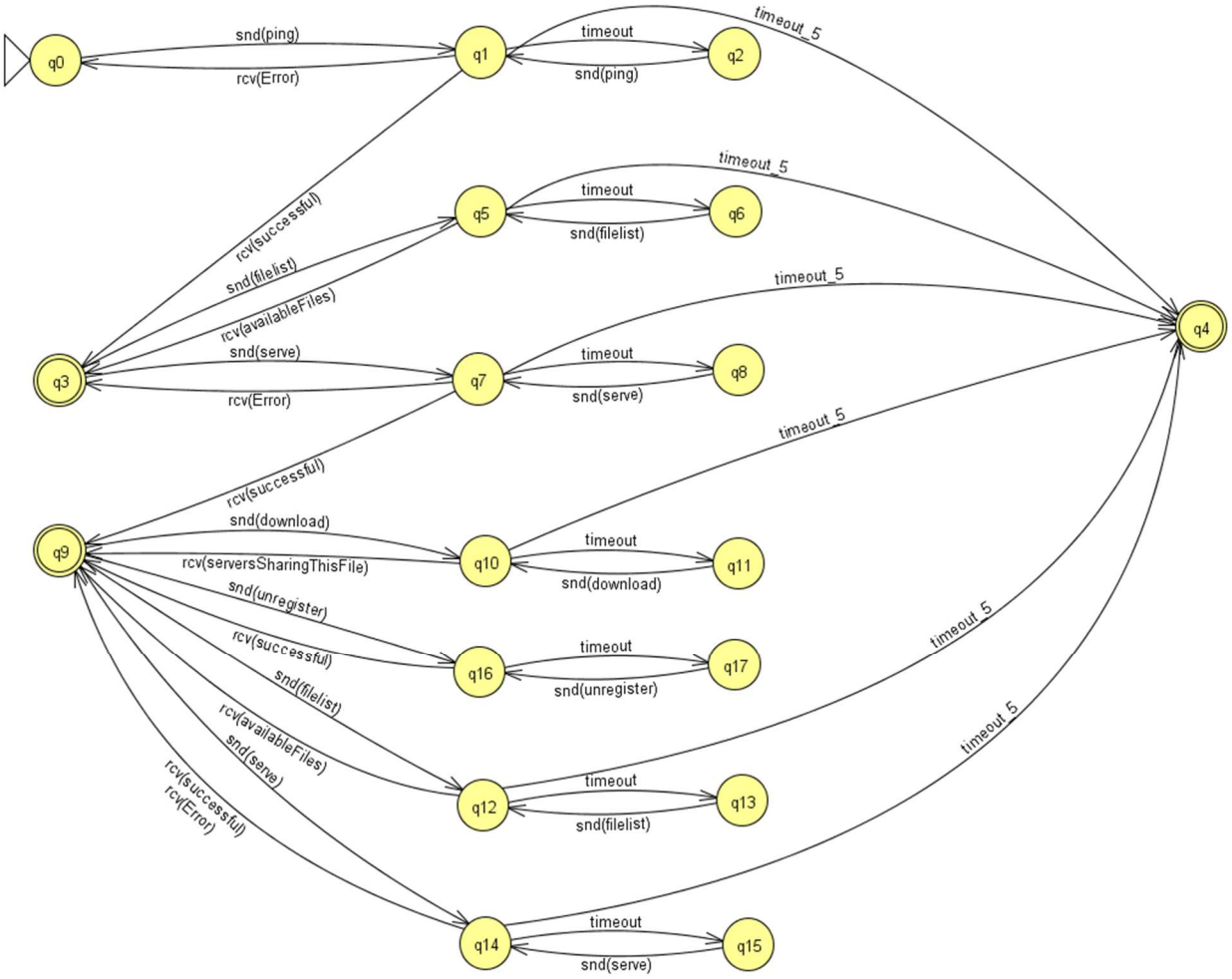
Opcode (1 byte)
0x08

4. Autómatas de protocolo

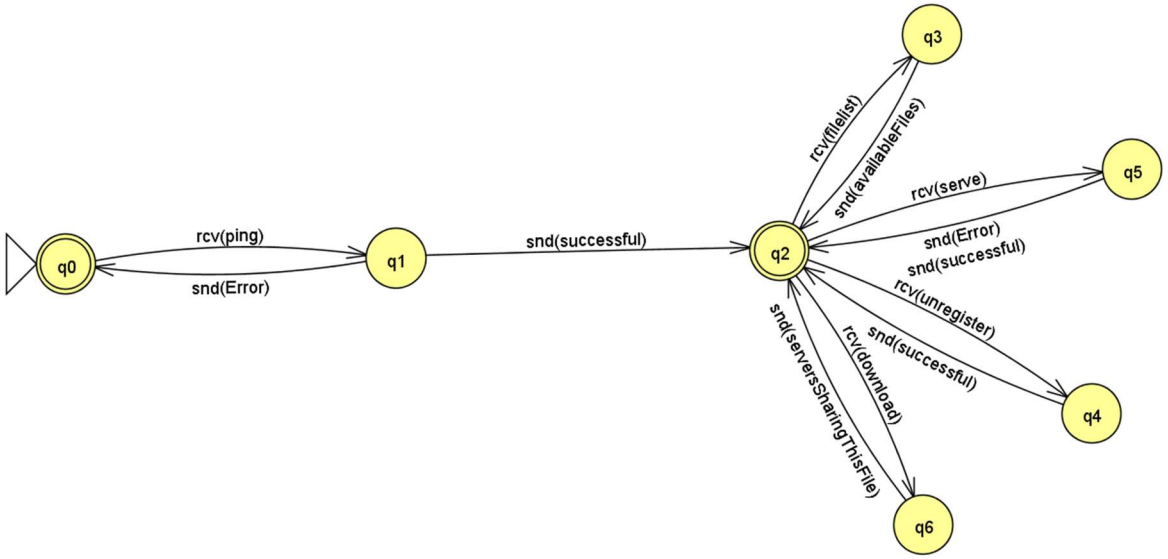
Con respecto a los autómatas, hemos considerado las siguientes restricciones:

- Un cliente del directorio no puede realizar ninguna operacion relacionada con los ficheros si antes no ha hecho un ping exitoso con el Directorio.
- Un cliente de ficheros no puede descargar ningún fichero proporcionado por otro peer servidor si antes no ha publicado sus ficheros.
- Un cliente de ficheros no puede subir ningún fichero a otro cliente si este último no está sirviendo ya sus ficheros.
- Para cualquier mensaje que el cliente envía al Directorio, si se producen 5 timeouts se llega a un estado final.

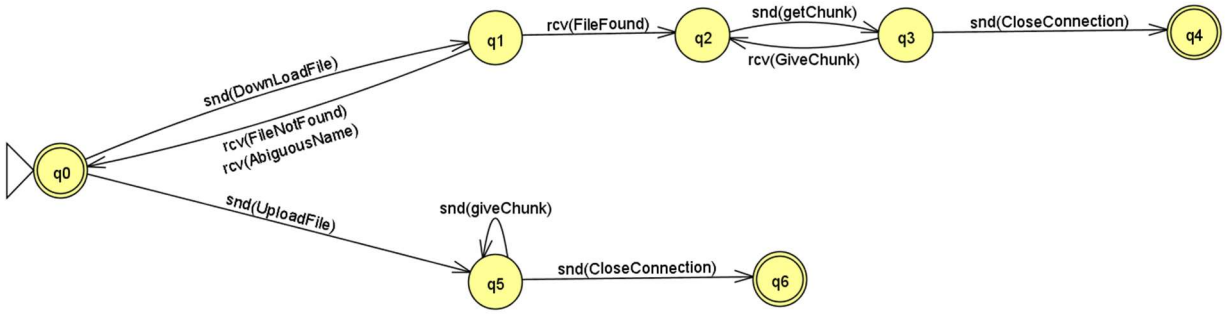
Autómata rol cliente de directorio



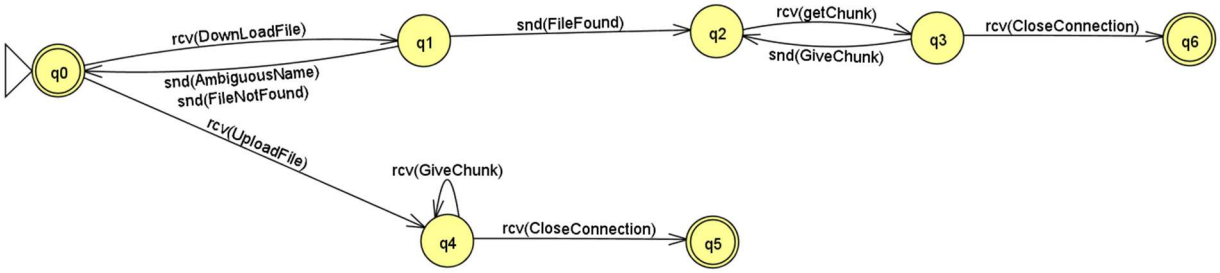
Autómata rol servidor de directorio



Autómata rol cliente de ficheros



Autómata rol servidor de ficheros



5. Mejoras implementadas y breve descripción sobre su programación

Mejora: upload

Descripción:

Se ha implementado la funcionalidad que permite a un peer enviar (subir) archivos a otro peer que actúa como servidor. Esta mejora amplía el protocolo TCP del sistema NanoFilesP2P para incluir la subida de ficheros.

Funcionamiento:

- **Lado cliente (emisor):**
  - A través del comando upload, el cliente localiza el fichero a subir desde su base de datos local (NanoFiles.db) utilizando una subcadena del nombre.
  - Si se encuentra un único fichero coincidente, se establece una conexión TCP con el servidor destino y se envía un mensaje OPCODE_UPLOAD_FILE indicando el nombre del fichero.
  - A continuación, se transmite el contenido del fichero en bloques binarios (chunks) mediante mensajes OPCODE_GIVE_CHUNK, y finalmente se cierra la conexión con OPCODE_CLOSE_CONNECTION.
- **Lado servidor (receptor):**
  - Al recibir el mensaje OPCODE_UPLOAD_FILE, el servidor crea un fichero local en su carpeta compartida y abre un FileOutputStream.
  - A medida que recibe los mensajes OPCODE_GIVE_CHUNK, escribe los datos en el fichero.
  - Al cerrarse la conexión, el servidor calcula el hash, tamaño y ruta del fichero recibido, y lo registra en su base de datos (NanoFiles.db).

Clases modificadas:

- NFController: interpreta el comando upload y lanza el proceso.
- NFControllerLogicP2P: implementa la lógica de subida en uploadFileToServer.
- NFServer: extiende el método serveFilesToClient para manejar la recepción de archivos y su integración en la base de datos.
- FileDataBase: se ha añadido el método `public void addFile(FileInfo fileInfo)` para añadir el nuevo fichero a la base de datos.

Mejora: Serve con puerto efímero

Descripción:

Se ha modificado el sistema para permitir que un peer sirva archivos usando un puerto efímero TCP, es decir, uno asignado dinámicamente por el sistema operativo.

Funcionamiento:

- **Inicio del servidor**:
  - En la clase NFControllerLogicP2P, el método startFileServer() instancia un objeto NFServer pasando el puerto 0.
  - Dentro del constructor de NFServer, se usa serverSocket.bind(new InetSocketAddress(port)). Si el puerto es 0, el sistema escoge uno libre automáticamente.
  - Luego, se actualiza this.port con serverSocket.getLocalPort() para registrar el puerto realmente asignado.

- **Comunicación con el directorio**:
    - Tras iniciar el servidor, el método registerFileServer en NFControllerLogicDir obtiene el puerto real mediante controllerPeer.getServerPort() y lo incluye en el mensaje DirMessage al directorio.
    - El mensaje serve enviado al directorio incluye ese puerto dinámico, lo que permite a otros peers conectarse correctamente.

Clases modificadas:
- NFControllerLogicP2P -> método startFileServer(): inicia NFServer con puerto efímero (port = 0).
- NFServer -> constructor actualiza this.port tras el bind.
- NFControllerLogicDir -> usa el puerto dinámico para registrar al peer servidor.

Mejora: filelist ampliado con servidores

Descripción:
Se ha extendido el funcionamiento del comando filelist para que no solo devuelva la lista de ficheros disponibles en el sistema, sino también las direcciones IP y puertos de los servidores (peers) que los comparten. Esto permite a los usuarios saber de antemano quién tiene qué archivo antes de descargarlo.

Funcionamiento:
- **Respuesta del directorio:**
    - En NFDIrectoryServer se utiliza una estructura para registrar los archivos que comparte cada peer:
      public static Map<InetSocketAddress, ArrayList<FileInfo>> *addrMapFiles.*
    - Cuando un cliente ejecuta el comando serve, se le registra en este mapa con su dirección y la lista de ficheros que sirve.
    - Al recibir una solicitud con OPERATION_FILELIST, se recorre este mapa para generar un nuevo mapa invertido:
      Map<FileInfo, List<InetSocketAddress>> availableFiles;
    - Este nuevo mapa asocia cada fichero con la lista de peers que lo tienen.
    - Este mapa se envía dentro de un objeto DirMessage como respuesta al cliente.

- **Cliente: mostrar información ampliada:**
    - El cliente, en el método getAndPrintFileList() de NFContollerLogicDir, recibe este mapa y lo imprime con un nuevo método añadido a la clase FileInfo:
      FileInfo.*printToSysoutWithServers*(trackedFiles);
    - Este método muestra en consola una tabla con columnas:
        - Nombre del fichero
        - Tamaño
        - Hash
        - Lista de servidores que lo tienen
    - Se decidió añadir este método en vez de modificar el método
      public static void printToSysout(FileInfo[] files),
      por el simple hecho de que este último se usa en otros casos y su modificación los complicaría.

Mejora: Quit que actualiza ficheros y servidores

Descripción:

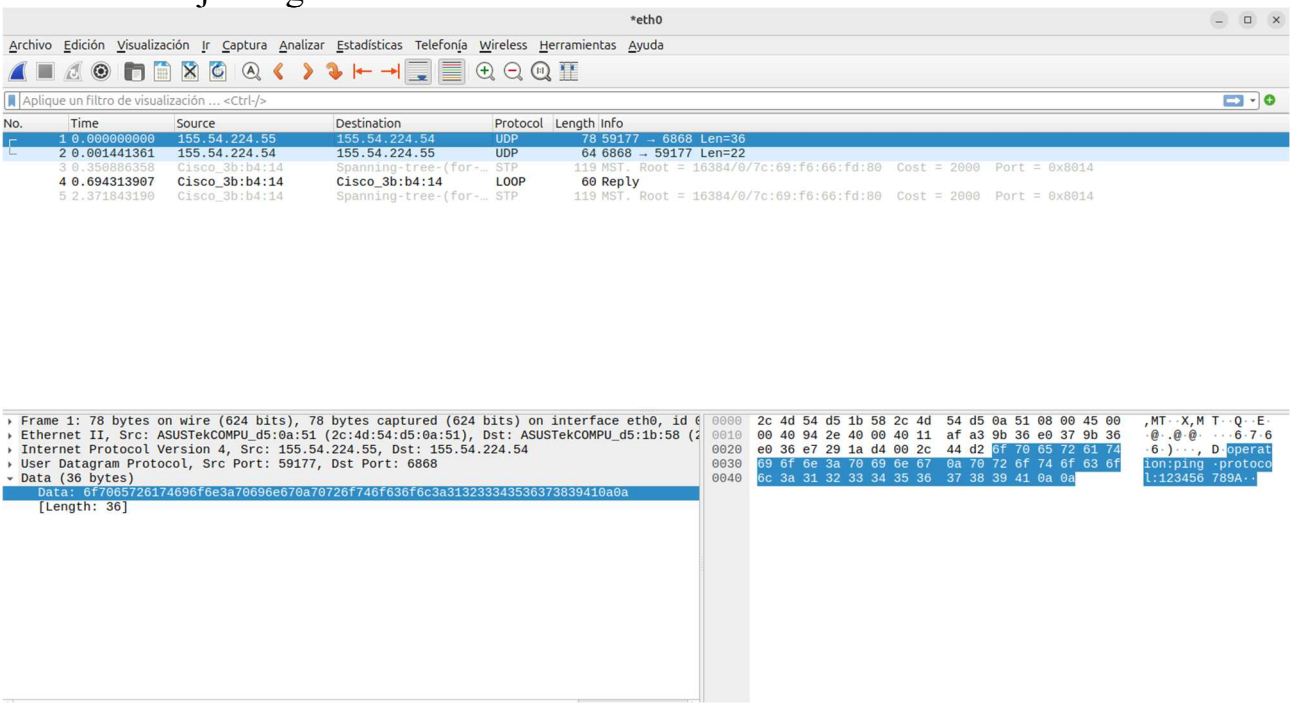
Se ha implementado una mejora en el comando quit para que, al salir de la aplicación, el peer se dé de baja correctamente como servidor de ficheros en el directorio. Esto evita que otros peers intenten conectarse a un servidor inexistente, manteniendo la integridad del sistema.

Funcionamiento:

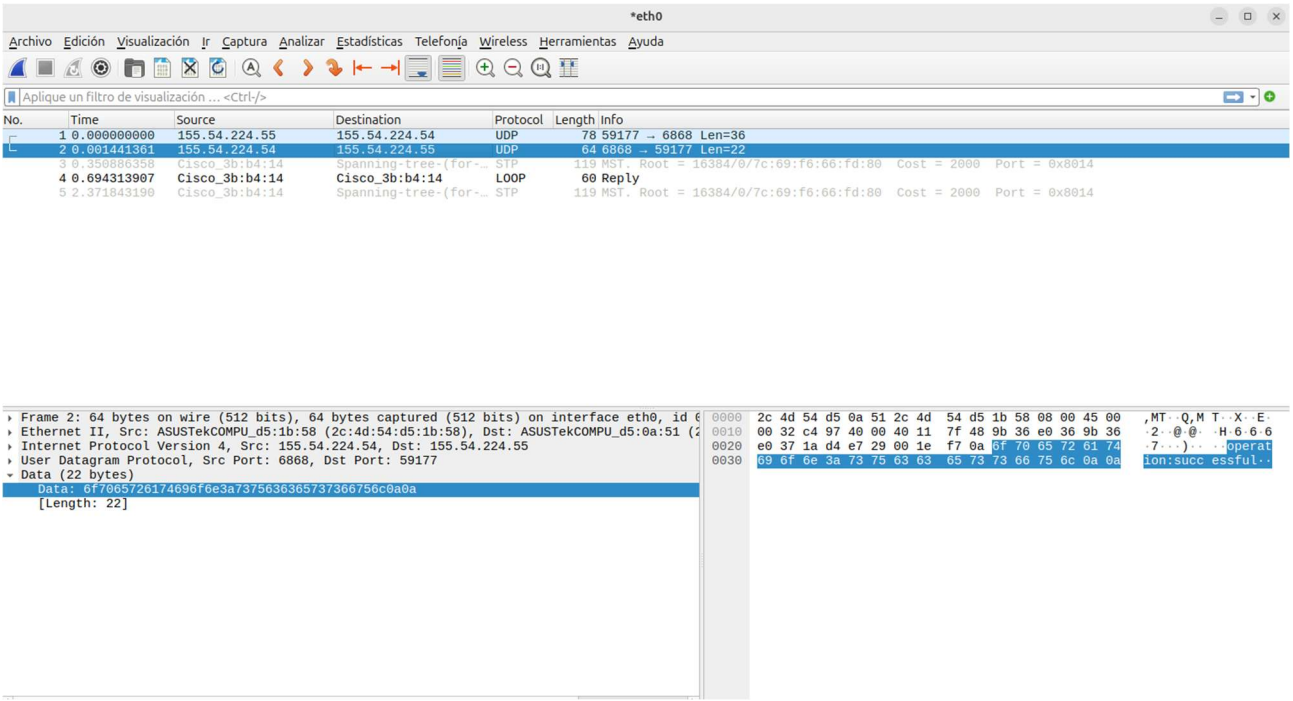
- **Al ejecutar el comando quit:**
  - En NFController, al procesar el comando COM_QUIT, se comprueba si el servidor TCP está activo (controllerPeer.serving()).
  - Si es así, se detiene el servidor (controllerPeer.stopFileServer()).
  - A continuación, se llama a controllerDir.unregisterFileServer() para comunicar al directorio que este peer ya no está disponible.
- **Desregistro en el directorio:**
  - En NFControllerLogicDir, el método unregisterFileServer() crea un mensaje DirMessage de tipo OPERATION_UNREGISTER, incluyendo el puerto del servidor (obtenido mediante controllerPeer.getServerPort()).
  - En el NFDirectoryServer, al recibir este mensaje, elimina del mapa addrMapFiles la entrada correspondiente a esa dirección.

6. Capturas de Whireshark

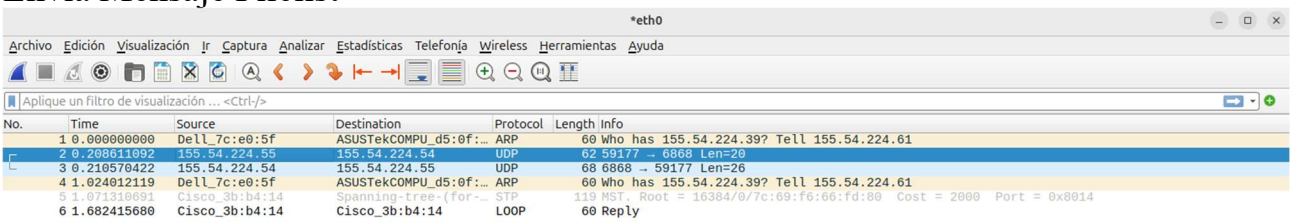
Envía Mensaje Ping

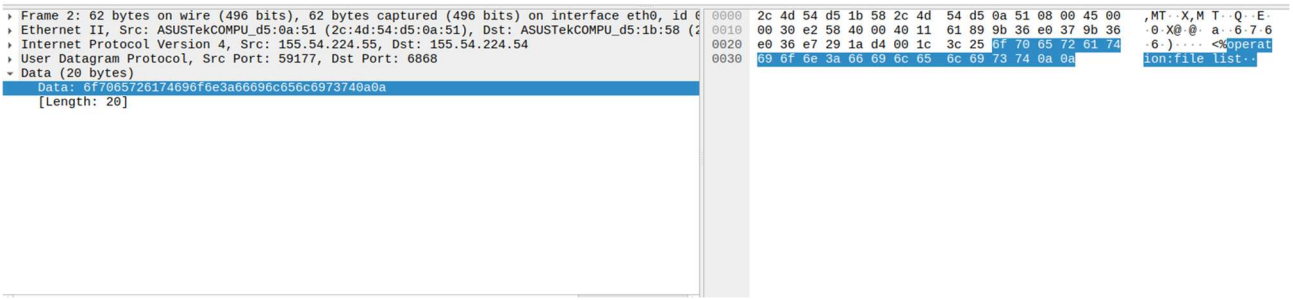


Recibe Successful

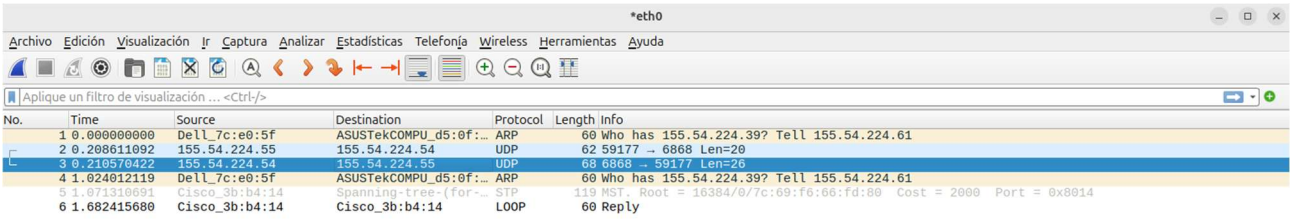


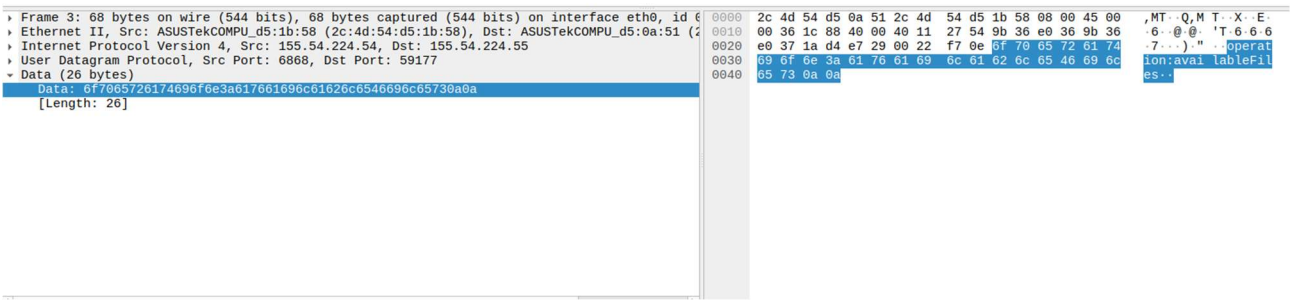
Envía Mensaje Filelist



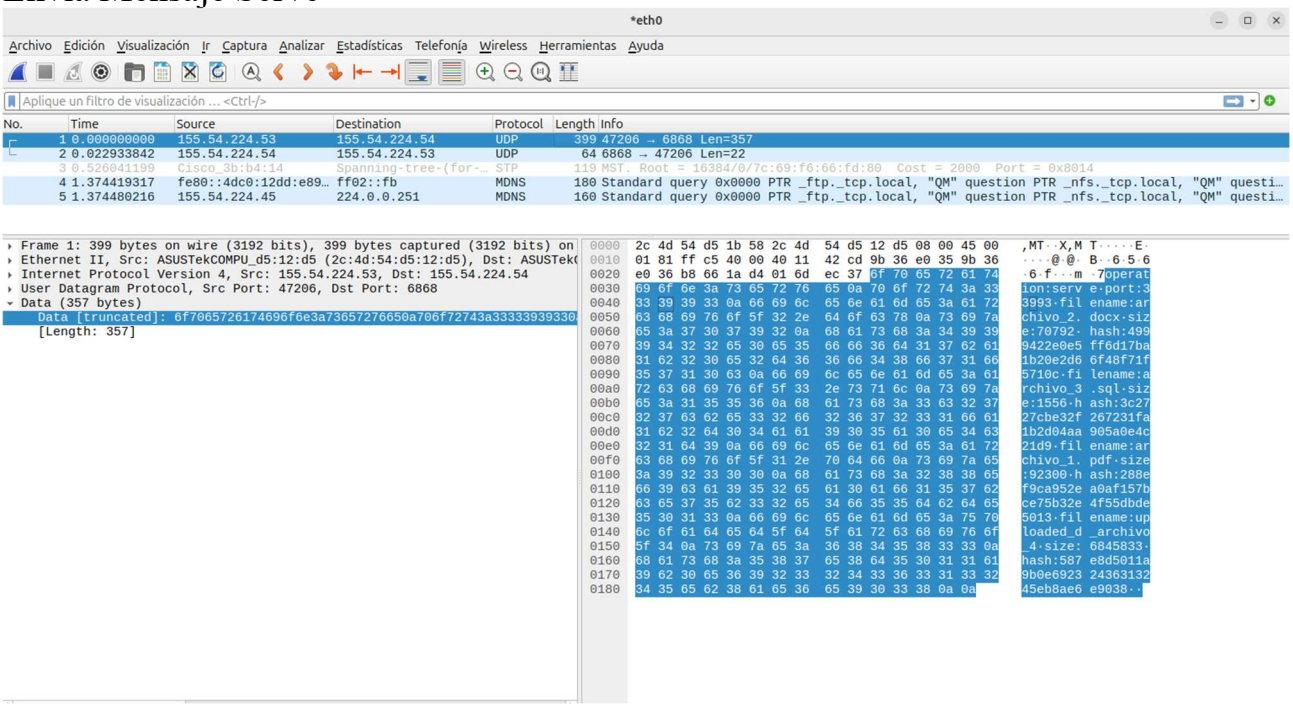


Recibe AvailableFiles

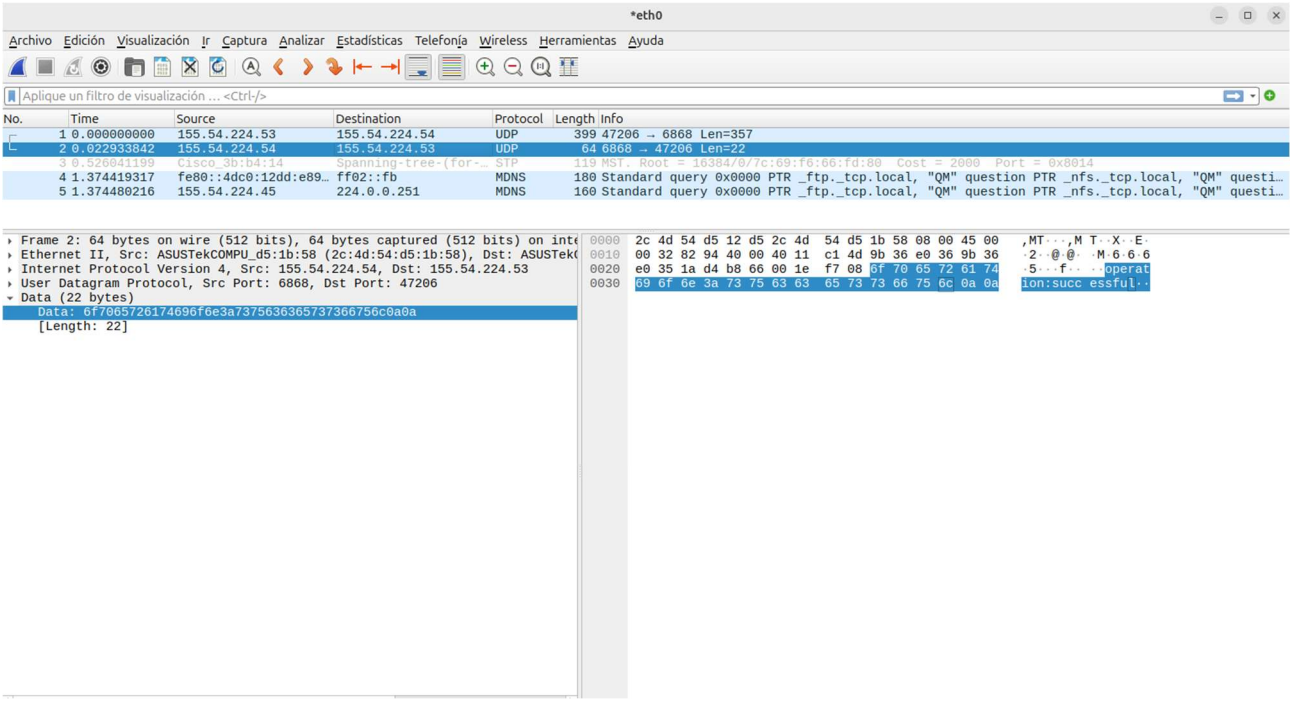




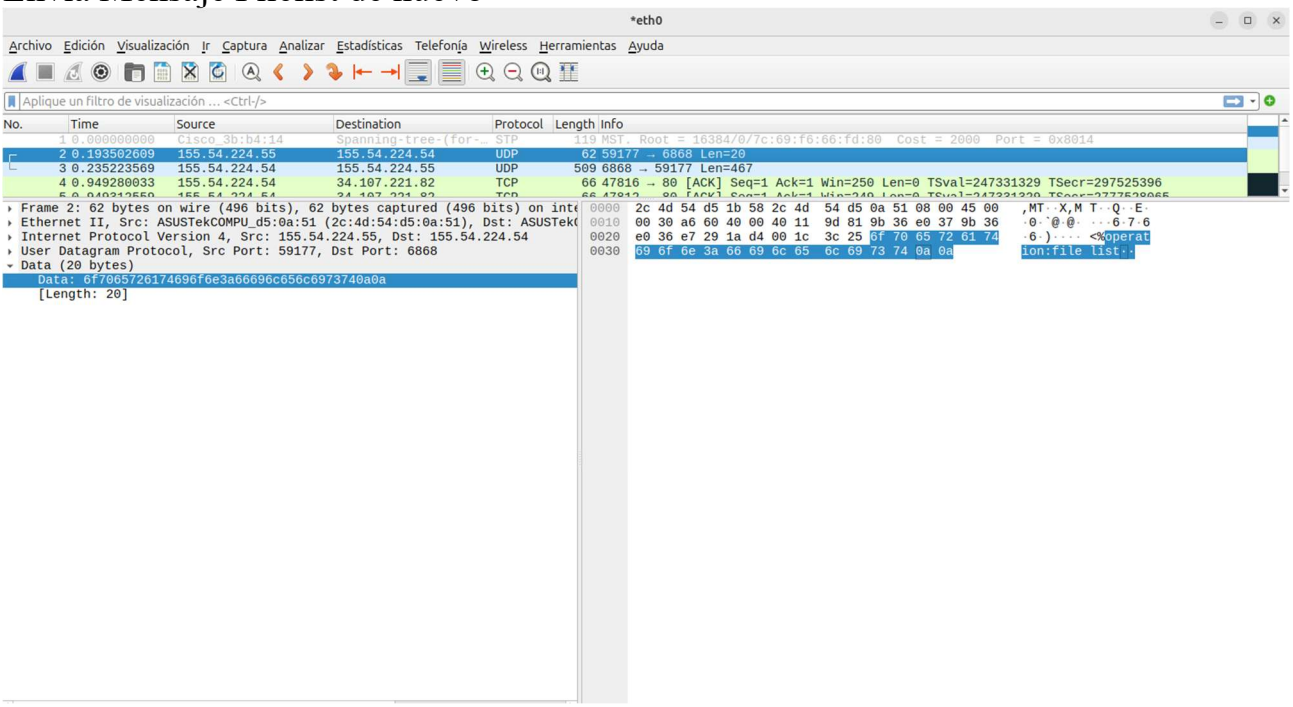
Envía Mensaje Serve



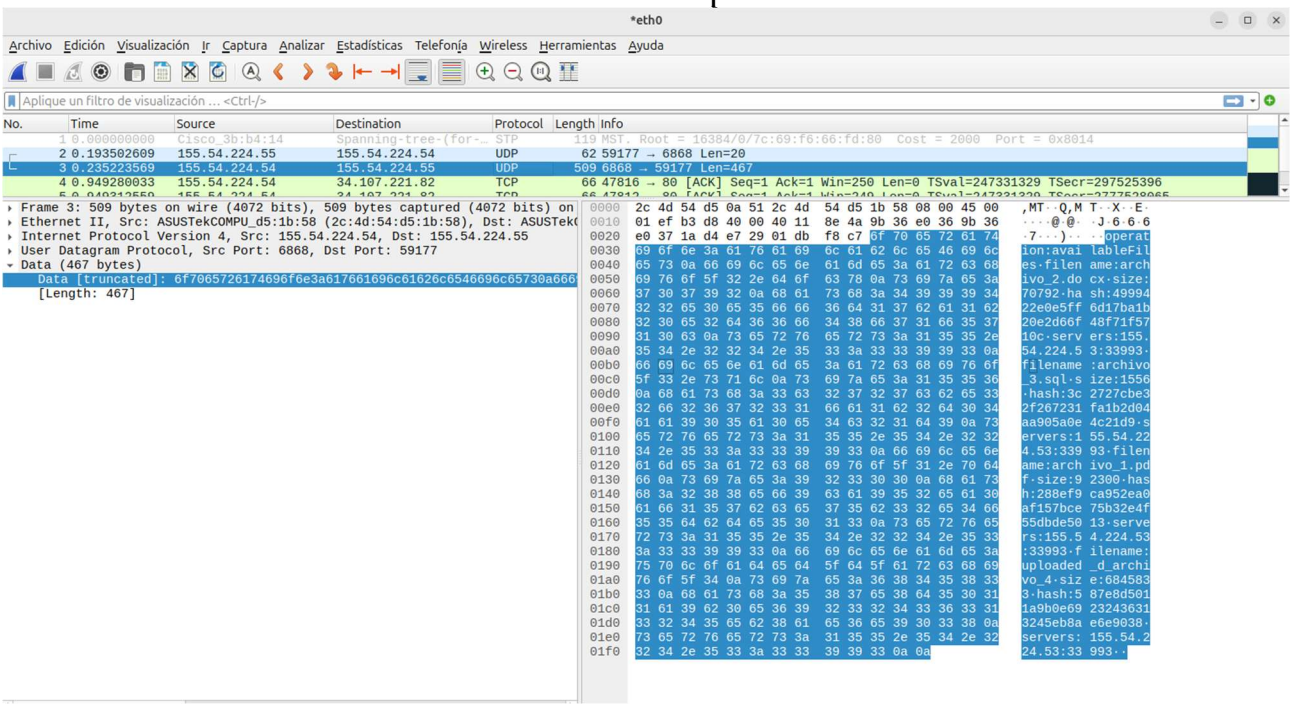
Recibe Successful



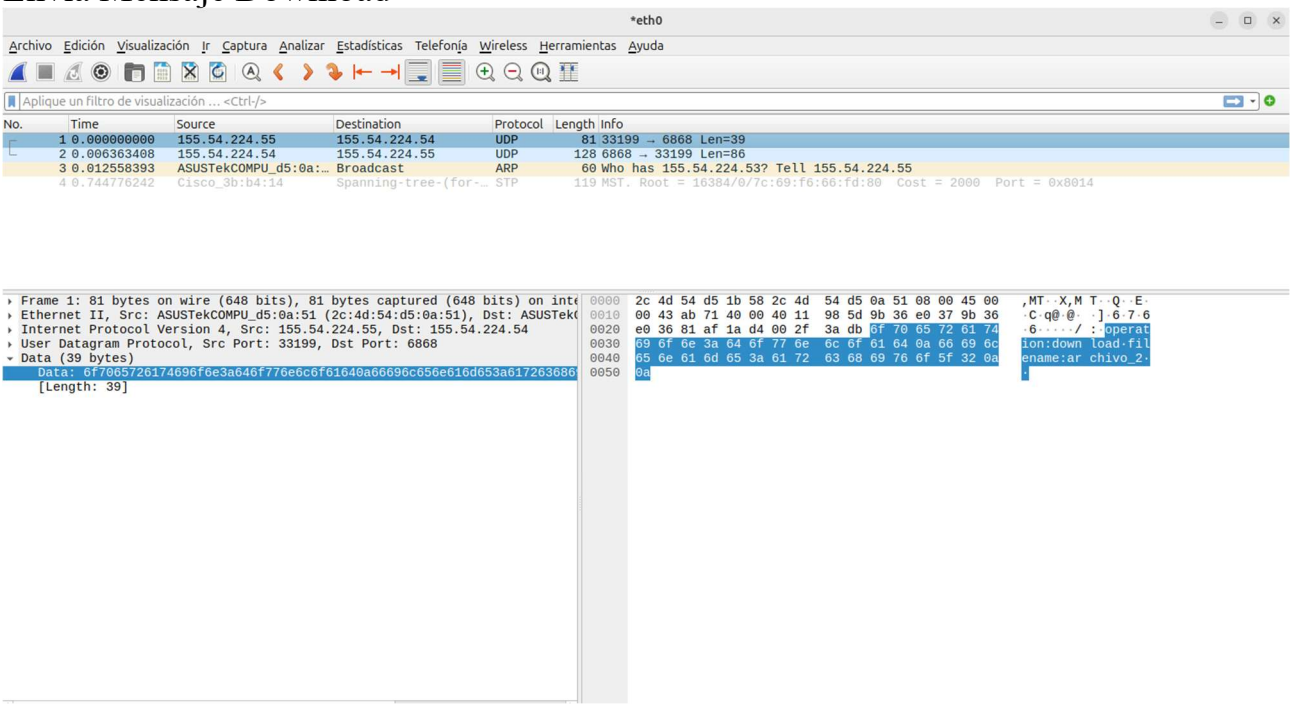
Envía Mensaje Filelist de nuevo



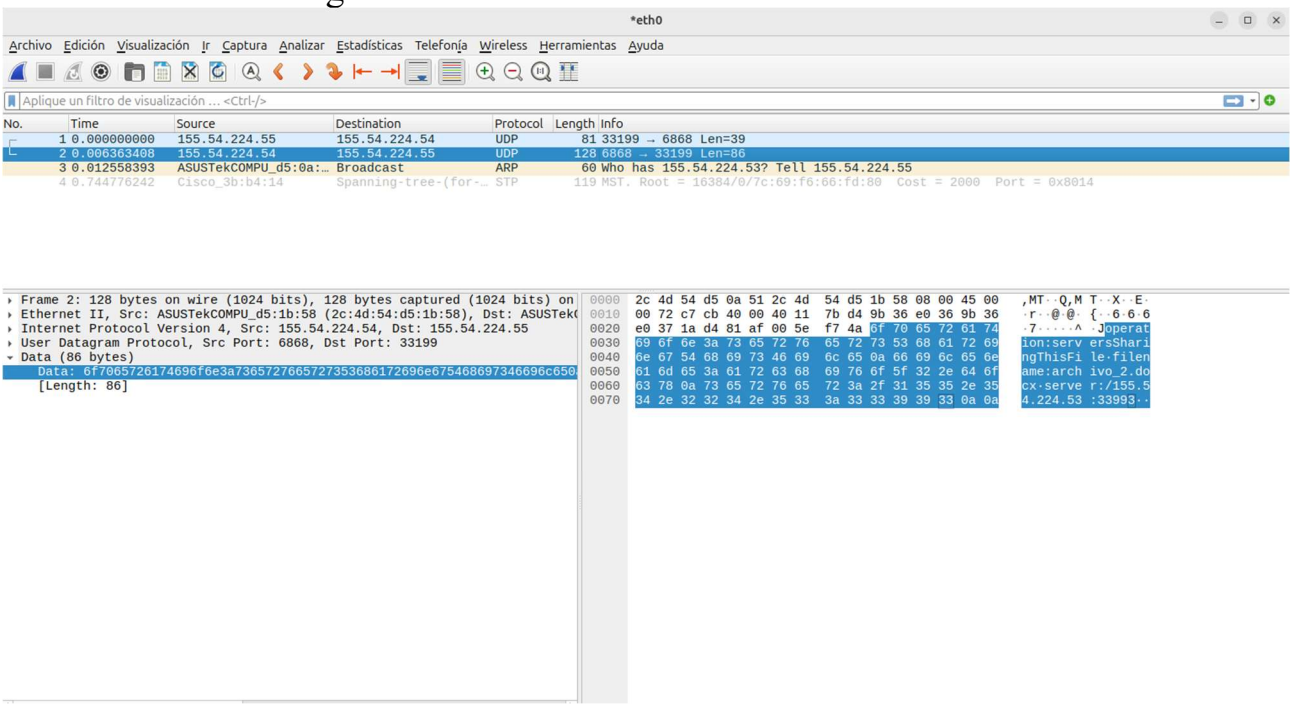
Recibe AvailableFiles con los nuevos ficheros que está sirviendo



Envía Mensaje Download



Recibe ServersSharingThisFile



7. Conclusiones

El desarrollo del sistema nanoFilesP2P ha permitido comprender e implementar conceptos fundamentales de redes de computadores, como la comunicación mediante sockets UDP y TCP, el diseño de protocolos de mensajería ASCII, y la coordinación entre múltiples peers en una arquitectura distribuida tipo P2P.

Durante el proyecto, se ha logrado construir una infraestructura funcional que permite a distintos usuarios compartir, descargar y subir archivos de forma eficiente y descentralizada, con la ayuda de un servidor directorio centralizado que actúa como punto de coordinación.

A lo largo del proceso, se han abordado múltiples retos, entre ellos: el diseño de protocolos personalizados, para lo cual fue necesario definir y parsear mensajes ASCII bien estructurados para intercambiar información entre clientes, servidores y el directorio; y la gestión de sockets y concurrencia, para lo que se trabajó con sockets TCP con el objetivo de establecer múltiples conexiones en paralelo.

Además, se han implementado mejoras relevantes al diseño original que amplían su funcionalidad y robustez. Gracias a estas mejoras, el sistema es más dinámico, realista y útil en un entorno simulado de compartición de ficheros. Además, el trabajo ha fomentado el pensamiento modular, la reutilización de código y el uso de estructuras como mapas para gestionar asociaciones clave entre direcciones y ficheros.

En resumen, *nanoFilesP2P* ha sido una experiencia muy útil para comprender cómo funciona un sistema donde varios usuarios pueden compartir archivos entre sí. A lo largo del proyecto se han aplicado ideas clave sobre cómo se comunican los programas entre diferentes ordenadores, y se ha aprendido a organizar bien el código para que todo funcione de forma coordinada y fiable. Además, las mejoras añadidas han permitido construir un sistema más completo, flexible y cercano a cómo funcionan realmente este tipo de aplicaciones.