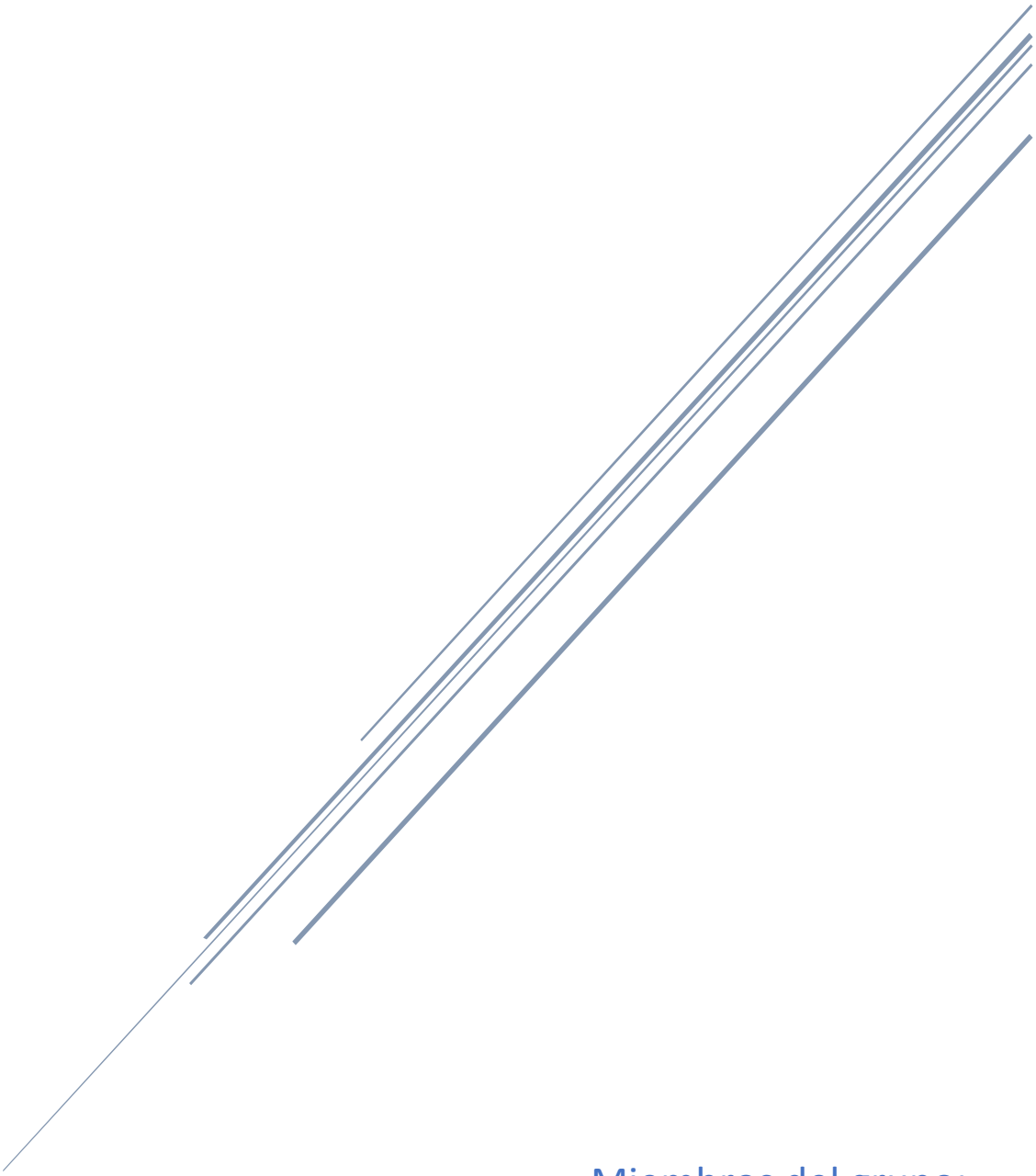
PROYECTO NANOFILES

Redes de Comunicación



Miembros del grupo:
Enrique Hernández Noguera- Subgrupo 2.2
Javier Galindo Garre – Subgrupo 2.2
Profesor de prácticas: Eduardo Salvador Iniesta Soto

Índice

Introducción

Este documento describe el diseño del protocolo de comunicación para un sistema de intercambio de archivos P2P con un directorio centralizado. El protocolo define los mensajes que los peers usan para comunicarse con el directorio y entre sí.

Al diseñarlo, hemos considerado varias mejoras:

- Estructura clara: Se han definido distintos tipos de mensajes para organizar mejor la comunicación.
- Transferencias eficientes: Los archivos se descargan por fragmentos desde varios peers para acelerar la descarga.
- Gestión automática: Los peers pueden registrarse y compartir archivos fácilmente con el directorio.
- Compatibilidad y seguridad: Se verifica la compatibilidad del protocolo y se usan hashes para asegurar la integridad de los archivos.
- Manejo de errores: Se han previsto respuestas específicas para distintos problemas, como archivos no encontrados o errores en el registro

Protocolos Diseñados

Directorio

Formato de los mensajes y ejemplos de los mismos

Mensajes asociados al comando “Ping”

Mensaje "Solicitud de Ping"

- Operación: ping.

- Emisor: NanoFiles.

- Receptor: Directory.

- Campo/s adicionales: identificador de protocolo del peer.

- Finalidad: pedirle al directorio que compruebe si está activo y si el protocol_id es compatible.

- Condiciones de envío: ninguna.

- Acciones al recibirlo: el directorio comprueba si el protocol_id recibido es igual al suyo y responde según se cumpla.

- Formato:

operation: ping\n

protocol: <protocolId>\n

\n

- Ejemplo

operation: ping\n

protocol: <77777777>\n

\n

NOTA: en realidad lo que se envía es...

operation:ping\n protocol:7777777\n\n

Mensaje "Respuesta de Ping Correcto"

- Operación: pingOK.

- Emisor: Directory.

- Receptor: NanoFiles.

- Campo/s adicionales: ninguno.

- Finalidad: informar al peer de que el directorio está a la escucha y el protocol_id es el mismo (compatible).

- Condiciones de envío: que se haya recibido un mensaje ping (solicitud de ping) por parte de un peer y que se cumpla que el protocol_id es compatible con el de dicho peer.

- Acciones al recibirlo: el peer informa por la consola de que el directorio está activo y es compatible.

- Formato:

operation: pingOK\n

\n

Mensaje "Respuesta de Ping Incorrecto"

- Operación: pingNOK.

- Emisor: Directory.

- Receptor: NanoFiles.

- Campo/s adicionales: ninguno.

- Finalidad: informar al peer de que el directorio está a la escucha pero que el protocol_id no es el mismo (incompatible).

- Condiciones de envío: que se haya recibido un mensaje ping (solicitud de ping) por parte de un peer y que el protocol_id sea incompatible con el de dicho peer.

- Acciones al recibirlo: el peer informa por la consola de que el directorio está activo pero que no es compatible.

- Formato:

operation: pingNOK\n

\n

Mensajes asociados al comando “Filelist”

Mensaje "Solicitud de fileList"

- Operación: filelist.

- Emisor: NanoFiles.

- Receptor: Directory.

- Campo/s adicionales: ninguno.

- Finalidad: pedirle al directorio una lista con todos los ficheros (lista que incluirá los nombres, tamaño en bytes y código hash de cada fichero) que el/los clientes Nanofile/s hayan subido al mismo.

- Condiciones de envío: se haya establecido conexión con el directorio y se haya asegurado de que este es compatible mediante el comando ping.

- Acciones al recibirlo: el directorio envía la información descrita en el apartado "Finalidad" de todos los ficheros que tenga y, en caso de que no tenga ninguno, no enviará información alguna (datagrama vacío).

- Formato:

Receiving… operation: filelist\n

\n

Mensaje "Respuesta con lista de files"

- Operación: responseFiles.

- Emisor: Directory.

- Receptor: NanoFiles.

- Campo/s adicionales: tantos campos "file" como ficheros quiera comunicar el directorio.

- Finalidad: dar al peer que lo pidió la lista con todos los ficheros que haya subidos en el directorio, dicha lista contendrá el nombre, tamaño y hash de cada fichero subido.

- Condiciones de envío: que se haya recibido un mensaje filelist de un peer.

- Acciones al recibirlo: el directorio envía un mensaje al peer que lo pidió con la información definida en el apartado "Finalidad".

- Formato:

operation: responseFiles \n

filehash: ...

filename: …

filesize: …

ipservidor: …

tcpportservidor: …

\n

- Ejemplo:

operation:responseFiles

filehash: 474116ce3a63b6a8bec94999aa4754474a82dc50

filename: lorem.txt

filesize: 4000

ipservidor: /127.0.0.1

tcpportservidor: 59314

ipservidor: /127.0.0.1

tcpportservidor: 58990

filehash: 523dc15ec387fc3d9e46b0cc04a73bc3a188c8de

filename: ejemplo.txt

filesize: 18

ipservidor: /127.0.0.1

tcpportservidor: 58990

// Básicamente, utilizaremos un formato de tipo field:value en el que field para cada fichero enviado será "file" mientras

// que value contendrá los valores nombre, tamaño y hash de dicho fichero enviado, separados entre si por el símbolo '&'.

Mensajes asociados al comando “Serve”

Mensaje "Servir files"

- Operación: serve.

- Emisor: NanoFiles.

- Receptor: Directory.

- Campo/s adicionales: número de puerto en el que escucha el peer servidor de ficheros y lista con la información (nombre, tamaño y hash) de todos los ficheros que el peer desea subir al directorio (un campo file por cada fichero a servir).

- Finalidad: informar de que has compartido tus ficheros al directorio, es decir, ahora son accesibles a través del mismo.

- Condiciones de envío: se haya establecido conexión con el directorio y se haya asegurado de que este es compatible mediante el comando ping.

- Acciones al recibirlo: el directorio registra la información de todos los ficheros enviados por el peer.

- Formato:

operation: serve\n

serverport: …

filehash: …

filename: …

filesize: …

\n

- Ejemplo

operation:serve

serverport:59314

filehash:474116ce3a63b6a8bec94999aa4754474a82dc50

filename:lorem.txt

filesize:4000

// Utilizaremos la misma manera de transmitior información (metadata) sobre un fichero que en el mensaje responseFiles previamente descrito.

Mensaje "Respuesta de confirmación de registro"

- Operación: serveSuccess.

- Emisor: Directory.

- Receptor: NanoFiles.

- Campo/s adicionales: ninguno.

- Finalidad: informar al peer que pidió registrar una lista de files en el directorio de que esta ha sido registrada con éxito.

- Condiciones de envío : que se haya recibido previamente el comando "serve" de algún peer.

- Formato:

operation: serveSuccess\n

\n

Mensajes asociados al comando “Download”

Mensaje "Descargar file"

- Operación: search.

- Emisor: NanoFiles.

- Receptor: Directory.

- Campo/s adicionales: nombre completo o subcadena del file a descargar.

- Finalidad: pedirle al directorio si tiene disponible un determinado fichero dado el nombre/subcadena del mismo y en caso de que disponga del mismo, que este nos envíe una lista con las direcciones y puerto (InetSocketAdress) de los peers que han publicado dicho fichero.

- Condiciones de envío: se haya establecido conexión con el directorio y se haya asegurado de que ha sido exitosa mediante el comando ping.

- Acciones al recibirlo: el directorio debe comprobar que el fichero identificado por la cadena enviada en el mensaje está disponible. Tras esto, puede ser que el fichero no esté disponible, que el identificador de fichero que el peer compartió sea ambiguo (casos de error sobre los que tendremos que informar al peer) y, finalmente, que el

fichero se encuentre disponible, tras lo que el directorio pasará a transmitirle el mismo al peer.

- Formato:

operation: search\n

target: <name>\n

\n

- Ejemplo:

operation:search

target:ejemplo.txt

Mensaje "Respuesta de confirmación de file disponible"

- Operación: searchSuccess.

- Emisor: Directory.

- Receptor: NanoFiles.

- Campo/s adicionales: dir

- Finalidad: informar al peer que pidió descargar un file dado su nombre o una subcadena del mismo de que el file está disponible y se le dará una lista de los peers que han compartido dicho fichero (lista que contendrá la dirección y puerto de escucha de dicho peer).

- Condiciones de envío: que se haya recibido previamente el comando "search" del peer al que se está respondiendo y que se identifique exitosamente el fichero identificado por la cadena del campo "target" de dicho mensaje search.

- Formato:

operation: searchSuccess \n

cadidateip: …

candidatetcpport: …

\n

- Ejemplo

operation:searchSuccess

candidateip:127.0.0.1

candidatetcpport:58990

Mensaje "Respuesta de file no disponible o ambiguo"

- Operación: searchUnSuccess.

- Emisor: Directory.

- Receptor: NanoFiles.

- Campo/s adicionales: ninguno.

- Finalidad: informar al peer que pidió descargar un file dado su nombre o una subcadena del mismo de que este no se encuentra disponible o de que dicho nombre/subcadena es amgiguo.

- Condiciones de envío: que se haya recibido previamente un mensaje de tipo "search" del peer al que se está respondiendo y que el directorio no disponga del fichero que debería ser identificado por la cadena del campo "target" de dicho mensaje search o que el directorio disponga de varios ficheros cuyo nombre case con la subcadena dada.

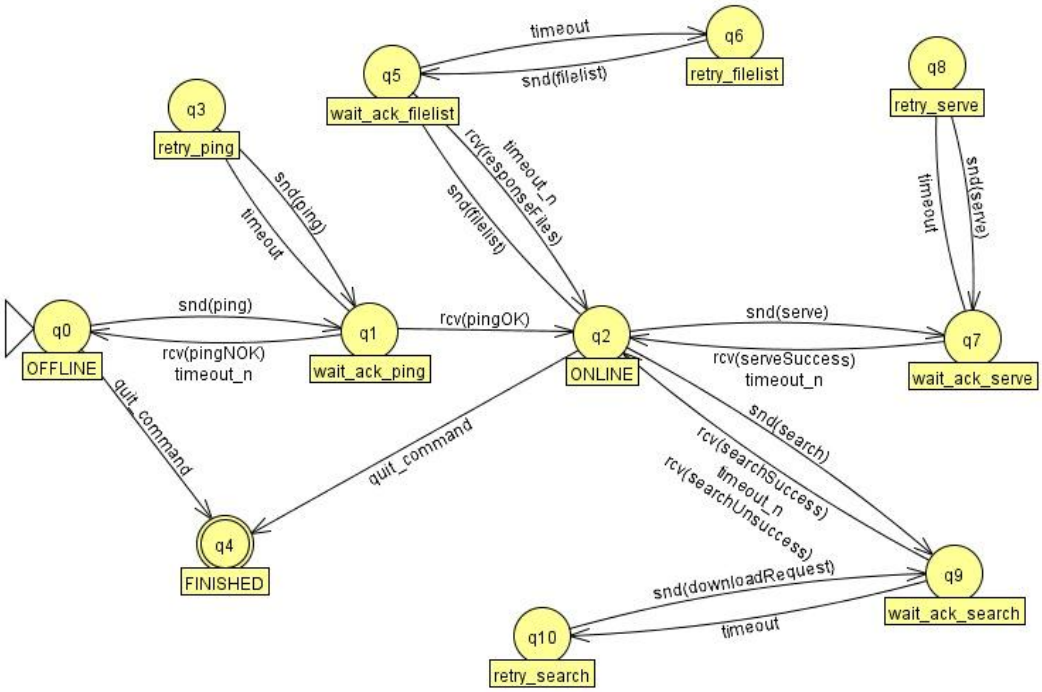
- Formato:

operation: searchUnSuccess \n

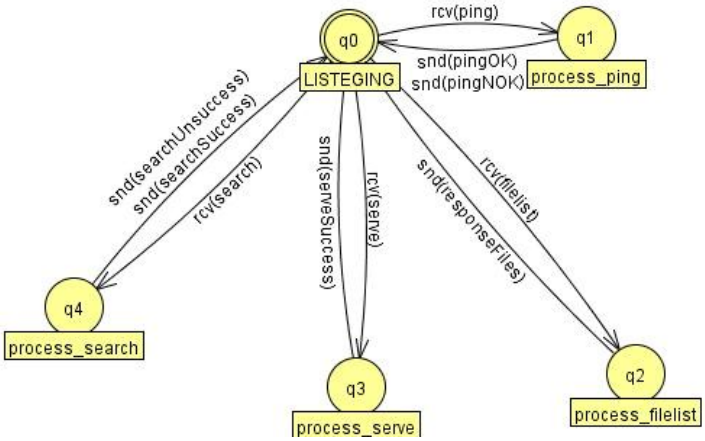
\n

Autómata cliente y servidor

Autómata rol cliente de directorio



Autómata rol servidor de directorio



Peer-to-peer

Formato de los mensajes y ejemplos de los mismos

Comando "Download"

Mensaje: Download_File (opcode = 1)

Sentido de la comunicación: Cliente -> Servidor

Descripción: Solicita la descarga de un archivo identificado por su nombre.

Opcode (1 byte)	Longitud del nombre	Nombre
1		

Mensaje: File_Metadata (opcode = 2)

Sentido de la comunicación: Servidor -> Cliente

Descripción: Confirma que el archivo solicitado está disponible para la descarga y manda el tamaño del fichero y el hash.

Opcode (1 byte)	Tamaño del fichero	Hash
2		

Mensaje: Ambiguous_Name (opcode = 3)

Sentido de la comunicación: Servidor -> Cliente

Descripción: Informa que existe más de un fichero con el mismo nombre.

Opcode (1 byte)
3

Mensaje: File_Not_Found (opcode = 4)

Sentido de la comunicación: Servidor -> Cliente

Descripción: Informa que el fichero solicitado no existe en el servidor.

Opcode (1 byte)
4

Mensaje: Get_Chunk (opcode = 5)

Sentido de la comunicación: Cliente -> Servidor

Descripción: Solicita un chunk con un tamaño desde una posición específica (offset).

Opcode (1 byte)	Longitud del nombre	Nombre
5		

Tamaño del chunk	Offset

Mensaje: Receive_Chunk (opcode = 6)

Sentido de la comunicación: Servidor -> Cliente

Descripción: Envia el chunk correspondiente junto con el contenido del chunk del fichero solicitado.

Opcode (1 byte)	Tamaño del chunk	Contenido de dicho chunk
6		

Comando “Upload” (Opcional)

Mensaje: Upload_File (opcode = 7)

Sentido de la comunicación: Cliente -> Servidor

Descripción: Solicita la subida de un fichero.

Opcode (1 byte)	Nombre del fichero	Longitud del nombre
7		

Tamaño del archivo	Hash

Mensaje: File _Found (opcode = 8)

Sentido de la comunicación: Servidor -> Cliente

Descripción: Informa que el fichero solicitado existe en el servidor.

Opcode (1 byte)
8

Mensaje: Send_Chunk (opcode = 9)

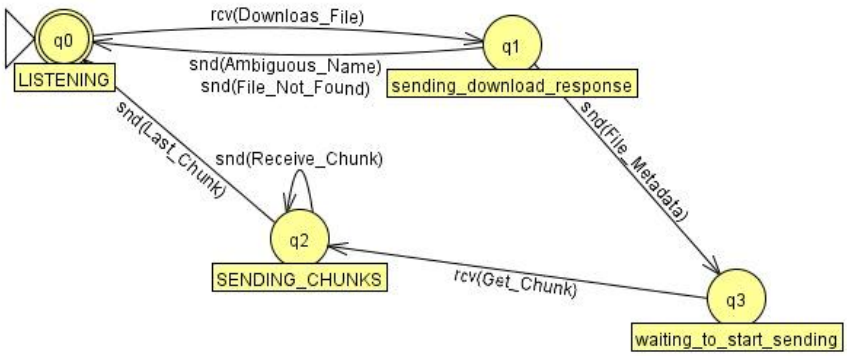
Sentido de la comunicación: Cliente -> Servidor

Descripción: Sube el chunk correspondiente junto con el contenido de dicho chunk.

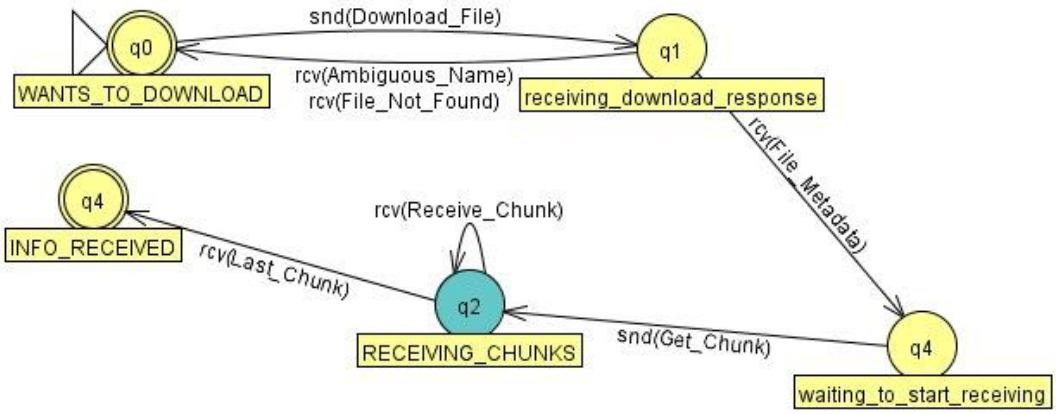
Opcode (1 byte)	Tamaño del chunk	Contenido de dicho chunk
9		

Autómata cliente y servidor

Autómata rol cliente de ficheros



Autómata rol servidor de ficheros



Mejoras implementadas y breve descripción sobre su programación.

Comando “Upload”

Para la implementación del comando upload hemos tenido que hacer lo siguiente:

- **Validación de ficheros.** Verificamos que el archivo que buscamos exista localmente y este dentro de los archivos compartidos.
- **Conexión al servido.** Creamos en la función uploadFileToServer un InetSocketAddress con la ip y el puerto tcp que hemos pasado como parámetros en el comando upload. En el NFConnector tenemos una función llamada uploadFile en el que enviamos un mensaje de solicitud de upload en el que le pasamos el tamaño del archivo, el hash, la longitud del nombre y el nombre del archivo.
- **Comprobación del servidor.** El servidor verifica si tiene ya un archivo con ese mismo nombre o hash y en ese caso mandaria un mensaje diciendo que ya existe dicho archivo. Si no existiese dicho archivo, mandaria que no existe y entonces el cliente subiría el archivo.
- **Envió del archivo.** Ahora el cliente, manda un mensaje de envio del chunk (el cual tiene tamaño de 1024 bytes) al que le pasamos el tamaño del chunk y el contenido de dicho chunk. El servidor escribiría en un fichero, el contenido que vamos subiendo de 1024 en 1024.
- **Finalización.** El cliente cerraría la conexión y daría por terminado la subida del archivo.

Comando serve con puerto efímero

Tras estar buscando en la documentación de la clase de InetSocketAddress, nos dimos cuenta que hay un apartado que pone que si se pone el puerto con el numero 0, deja al sistema operativo elegir un puerto efímero por lo que lo único hemos tenido que hacer ha sido cambiar el puerto y ponerlo a 0.

La pagina donde lo hemos encontrado ha sido la siguiente:

https://docs.oracle.com/javase/8/docs/api/java/net/InetSocketAddress.html

Como el servidor de directorio tiene que saber el puerto efímero en el que esta expuesto el servidor de ficheros. Tenemos que modificar tanto la estructura del mensaje enviado por el servidor de ficheros como la estructura para mantener la información dentro del servidor de directorio.

Para enviarle al servidor de directorio que se ha expuesto un puerto TCP determinado, extendemos el mensaje *serve* con un nuevo campo que indique el puerto TCP efímero que se ha expuesto.

De la misma forma, el servidor de directorio no tendrá que almacenar para cada fichero solamente una lista de IPs de los servidores de ficheros que publican dicho fichero, sino que tendría que incluir una tupla IP/TCP_port

Comando filelist ampliado

El comando filelist original, imprimía solamente los ficheros que tenía el servidor de directorio. Como tenemos que enviar al cliente además de los ficheros disponibles, los servidores candidatos a descargar dicho fichero. Tenemos que extender el formato de mensaje filelist para que devuelva para cada fichero una lista de IP/TCP_port (aunque se indica en el enunciado solo la IP) de los servidores. Para ello, usamos la misma estructura que hemos usando en el puerto efimero.

Comando "Quit"

Para la implementación del comando quit hemos tenido que hacer lo siguiente:

- Mensaje de desconexion. El cliente crea un mensaje de desconexión asociado a un puerto desde el cual se estaban sirviendo los archivos. Este mensaje se construye en la función unregisterFileServer.
- Desconexión por parte del servidor. Si el mensaje recibido por parte del cliente es correcto, el servidor mandaría un mensaje de exito en la desconexión. Tras ello, el servidor buscaría todos los archivos que incluían al peer que se está desconectando, eliminaría dicho cliente de su lista de clientes y si algún archivo no tuviese ningún cliente asociado, se eliminaría también de su lista de archivos.
- Confirmacion de la desconexion. Una vez recibida el éxito por parte del servidor, el cliente puede cerrar su proceso tranquilamente.

Comando download paralelo

Para la implementación del comando download paralelo hemos tenido que dividirla en tres etapas:

- La primera etapa. Esta primera etapa es en la que recibíamos y confirmábamos la búsqueda y descarga del fichero.
- La segunda etapa. En esta etapa confirmábamos que no existiese ninguna ambigüedad entre ficheros. Por ejemplo, si dos servidores tienen el mismo nombre de fichero, pero tienen diferente hash.
- La tercera etapa. En esta ultima etapa, descargamos el fichero de todos los servidores que lo contengan.

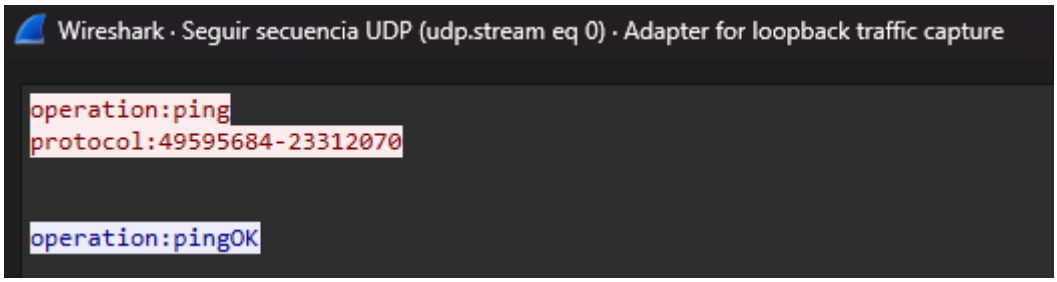
Para poder realizar todo esto, tenemos que entender que todas las etapas están sincronizadas entre si para poder descargar archivos paralelamente. En la primera etapa, se enviarían todos los mensajes a todos los servidores implicados. En la segunda etapa, analizaríamos los mensajes enviados en la primera etapa serializadamente. Finalmente, en la tercera etapa, si el servidor contiene dicho fichero (no ambiguo y encontrado) solicitaríamos paralelamente la descarga de bloques (chunks) a dichos servidores.

Para realizar la descarga de bloques, cada uno de los hilos pide bloques de 1024 bytes de contenido del fichero. Cuando un bloque es solicitado, ningún hilo puede volver a solicitarlo. Para verificarlo, se piden bloques de forma incremental (con un offset inicial a 0 hasta el tamaño

del fichero) por cada uno de los hilos. Para verificar que diferentes hilos no cogen el mismo offset, utilizamos una variable global siendo esta una sección critica.

Capturas de pantalla que muestren mediante Wireshark un intercambio de mensajes con el directorio

Comando “Ping”



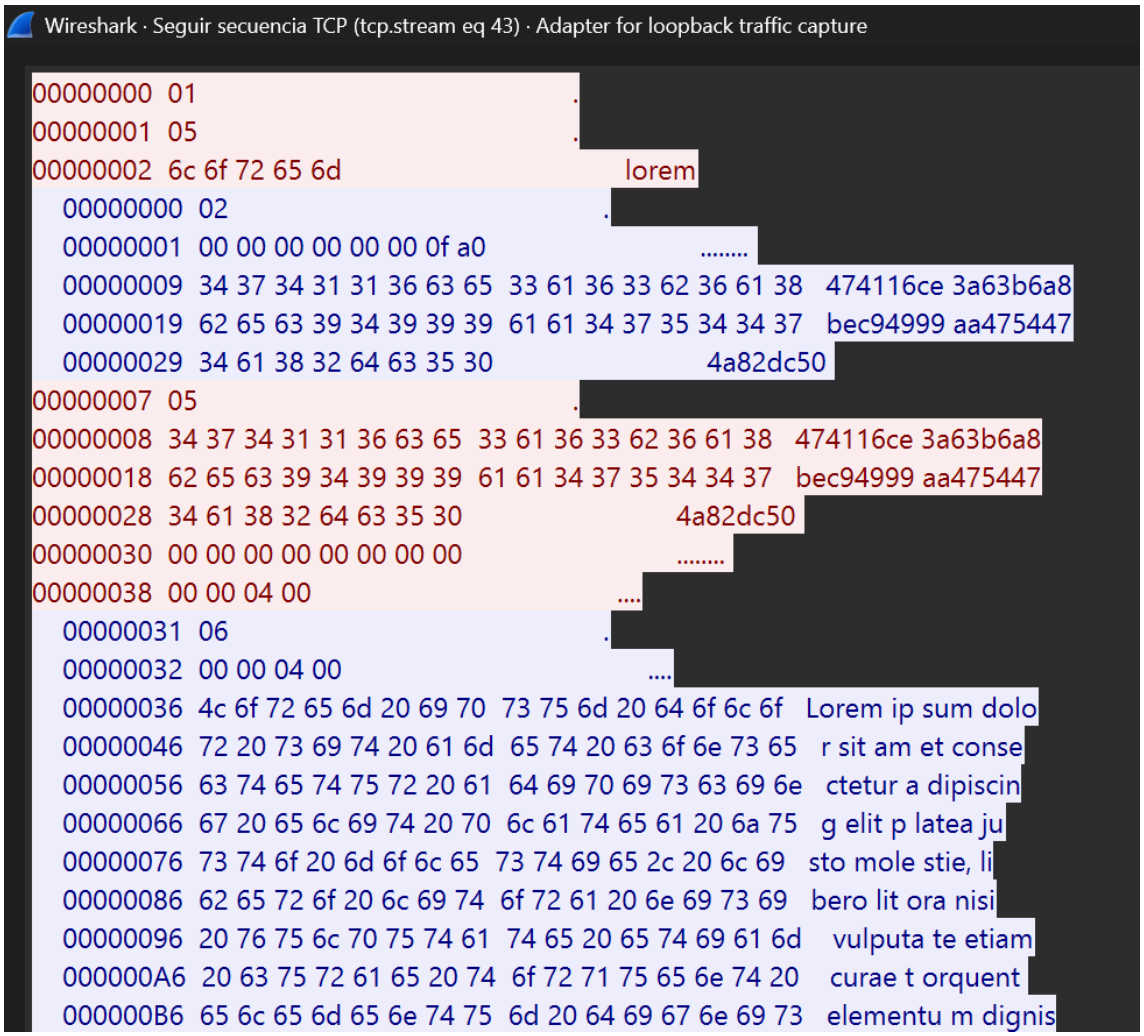
Comando “Filelist”

Comando “Serve”

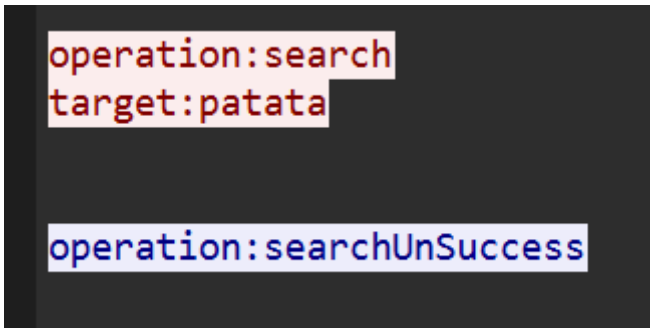
```
operation:serve
serverport:61350
filehash:474116ce3a63b6a8bec94999aa4754474a82dc50
filename:lorem.txt
filesize:4000
filehash:523dc15ec387fc3d9e46b0cc04a73bc3a188c8de
filename:ejemplo.txt
filesize:18


operation:serveSuccess
```

Comando “Download”

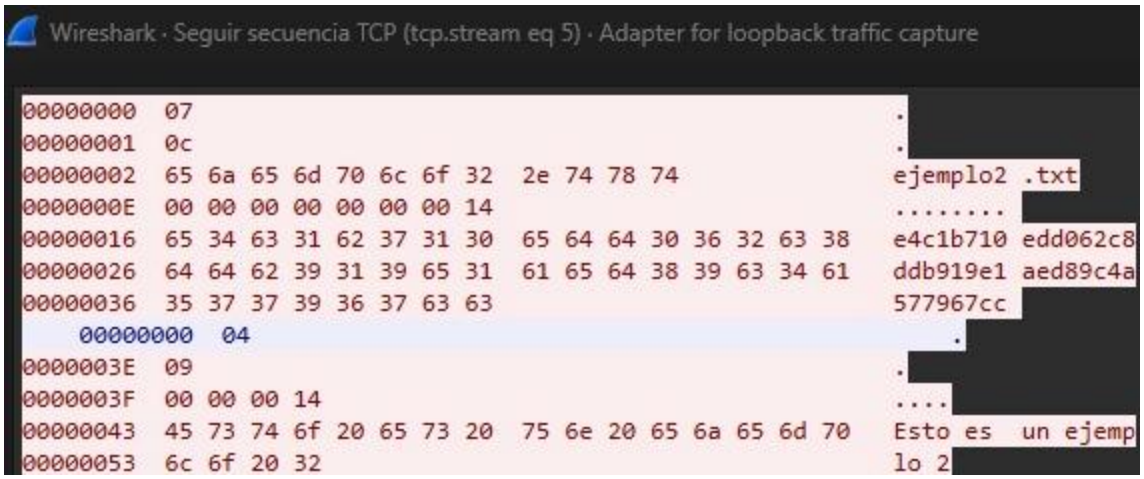
```
operation:search
target:lorem


operation:searchSuccess
candidateip:127.0.0.1
candidatetcpport:59296
```



Archivo no encontrado:



Comando “Upload”



11	2.785816	127.0.0.1	127.0.0.1	TCP	51737 → 51643 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
12	2.785914	127.0.0.1	127.0.0.1	TCP	51643 → 51737 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
13	2.785938	127.0.0.1	127.0.0.1	TCP	51737 → 51643 [ACK] Seq=1 Ack=1 Win=65280 Len=0
14	2.786045	127.0.0.1	127.0.0.1	TCP	51737 → 51643 [PSH, ACK] Seq=1 Ack=1 Win=65280 Len=1
15	2.786066	127.0.0.1	127.0.0.1	TCP	51643 → 51737 [ACK] Seq=1 Ack=2 Win=65280 Len=0
16	2.786099	127.0.0.1	127.0.0.1	TCP	51737 → 51643 [PSH, ACK] Seq=2 Ack=1 Win=65280 Len=1
17	2.786107	127.0.0.1	127.0.0.1	TCP	51643 → 51737 [ACK] Seq=1 Ack=3 Win=65280 Len=0
18	2.786136	127.0.0.1	127.0.0.1	TCP	51737 → 51643 [PSH, ACK] Seq=3 Ack=1 Win=65280 Len=12
19	2.786143	127.0.0.1	127.0.0.1	TCP	51643 → 51737 [ACK] Seq=1 Ack=15 Win=65280 Len=0
20	2.786166	127.0.0.1	127.0.0.1	TCP	51737 → 51643 [PSH, ACK] Seq=15 Ack=1 Win=65280 Len=8
21	2.786172	127.0.0.1	127.0.0.1	TCP	51643 → 51737 [ACK] Seq=1 Ack=23 Win=65280 Len=0
22	2.786189	127.0.0.1	127.0.0.1	TCP	51737 → 51643 [PSH, ACK] Seq=23 Ack=1 Win=65280 Len=40
23	2.786195	127.0.0.1	127.0.0.1	TCP	51643 → 51737 [ACK] Seq=1 Ack=63 Win=65280 Len=0
24	2.786656	127.0.0.1	127.0.0.1	TCP	51643 → 51737 [PSH, ACK] Seq=1 Ack=63 Win=65280 Len=1
25	2.786671	127.0.0.1	127.0.0.1	TCP	51737 → 51643 [ACK] Seq=63 Ack=2 Win=65280 Len=0
26	2.786895	127.0.0.1	127.0.0.1	TCP	51737 → 51643 [PSH, ACK] Seq=63 Ack=2 Win=65280 Len=1
27	2.786908	127.0.0.1	127.0.0.1	TCP	51643 → 51737 [ACK] Seq=2 Ack=64 Win=65280 Len=0
28	2.786992	127.0.0.1	127.0.0.1	TCP	51737 → 51643 [PSH, ACK] Seq=64 Ack=2 Win=65280 Len=4
29	2.787002	127.0.0.1	127.0.0.1	TCP	51643 → 51737 [ACK] Seq=2 Ack=68 Win=65280 Len=0
30	2.787027	127.0.0.1	127.0.0.1	TCP	51737 → 51643 [PSH, ACK] Seq=68 Ack=2 Win=65280 Len=20
31	2.787036	127.0.0.1	127.0.0.1	TCP	51643 → 51737 [ACK] Seq=2 Ack=88 Win=65280 Len=0
32	2.787140	127.0.0.1	127.0.0.1	TCP	51737 → 51643 [FIN, ACK] Seq=88 Ack=2 Win=65280 Len=0
33	2.787151	127.0.0.1	127.0.0.1	TCP	51643 → 51737 [ACK] Seq=2 Ack=89 Win=65280 Len=0
34	2.787324	127.0.0.1	127.0.0.1	TCP	51643 → 51737 [FIN, ACK] Seq=2 Ack=89 Win=65280 Len=0
35	2.787348	127.0.0.1	127.0.0.1	TCP	51737 → 51643 [ACK] Seq=89 Ack=3 Win=65280 Len=0

Enlace a grabación de pantalla mostrando los programas en funcionamiento

Conclusiones

Con este proyecto hemos aprendido cómo funciona una red P2P con un servidor centralizado que coordina los archivos compartidos por los usuarios. Hemos implementado comandos para registrar archivos, ver qué archivos hay disponibles, subirlos, descargarlos y salir de la red de forma ordenada. Se ha trabajado con conexiones TCP y UDP, y con hilos para permitir que varios usuarios se conecten al mismo tiempo. El sistema permite que los archivos y servidores se actualicen automáticamente, mostrando siempre información actual al usuario. En general, ha sido un proyecto útil para entender cómo se comunican los programas en red y cómo se pueden compartir archivos entre distintos ordenadores.