Memoria Proyecto *NanoFiles*

Redes de Comunicaciones

Curso 2022/23 – Convocatoria de Junio

Jose Luis Galiano Gómez
Juan Botía López
Grupo 1.3

ÍNDICE

1. Introducción

En esta memoria presentamos el proyecto de prácticas de la asignatura Redes de Comunicaciones, la aplicación NanoFiles, cuyo objetivo es la comunicación Cliente-Servidor con un Directorio en Internet, así como la comunicación y transmisión de ficheros *Peer-to-Peer*, apoyada en ese Directorio. Para ello, se hace uso del protocolo sin conexión y no confiable UDP para la comunicación con el Directorio, y el protocolo orientado a conexión y confiable TCP para la comunicación entre *peers*. La funcionalidad básica de la aplicación incluye, entre otras, conectarse y registrarse en el servidor, ofrecer tus ficheros a otros *peers*, conectarse a otros *peers* y descargar sus ficheros, etc.

2. Diseño de los Protocolos

En esta sección se especifica el diseño de los protocolos utilizados para el correcto funcionamiento de la aplicación NanoFiles. El comportamiento de esta y más concretamente la comunicación entre el directorio y los clientes, así como entre clientes, será manejado principalmente mediante mensajes con formatos y contenidos previamente definidos. Además, contaremos con un autómata tanto para el directorio como para el cliente que modelará el paso de mensajes así como otros detalles de la implementación.

2.1 Formato de los mensajes del protocolo de comunicación con el Directorio

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

NOTA: Un mismo formato de mensaje puede ser utilizado por varios tipos de mensajes diferentes (opcodes diferentes)

Formatos de mensajes

Formato: Control

Opcode (1 byte)

Formato: OneParameter

Opcode (1 byte)	**Parámetro (4 bytes)**

Formato: TLV

Opcode (1 byte)	**Length (4bytes)**	**Campo (n bytes)**

Formato: TLV_VAR

Opcode (1 byte)	**K Numero Campos (4 bytes)**	**Length 1 (4bytes)**	**Campo 1 (n_1 bytes)**	**…**	**Length k (4bytes)**	**Campo k (n_k bytes)**

Formato: FICH_VAR

Opcode (1 byte)	**K Numero Campos (4 bytes)**	**Length Nombre 1 (4 bytes)**

Nombre 1 (f_1 bytes)	**Length Hash 1 (4 bytes)**	**Hash 1 (h_1 bytes)**	**Size 1 (8 bytes)**	**. . .**	**Length Nombre k (4 bytes)**

Nombre k (f_k bytes)	**Length Hash k (4 bytes)**	**Hash k (h_k bytes)**	**Size k (8 bytes)**

Formato: IPPORT_FICH_VAR

Opcode (1 byte)	**Length Nickname (4 bytes)**	**Nickname (n bytes)**	**PORT (4 bytes)**	**K Numero Campos (4 bytes)**	**Length Nombre 1 (4 bytes)**

Nombre 1 (f_1 bytes)	**Length Hash 1 (4 bytes)**	**Hash 1 (h_1 bytes)**	**Size 1 (8 bytes)**	**. . .**	**Length Nombre k (4 bytes)**

Nombre k (f_k bytes)	**Length Hash k (4 bytes)**	**Hash k (h_k bytes)**	**Size k (8 bytes)**

Tipos y descripción de los mensajes

Mensaje: **Login (opcode = 1)**
Formato: Control
Sentido de la comunicación: Cliente → Directorio
Descripción: Este mensaje lo envía el cliente de NanoFiles al Directorio para solicitar “iniciar sesión” y obtener el número de servidores que hay disponibles en ese momento. El valor asignado al opcode es 1.
Ejemplo:

Opcode (1 byte)
1

Mensaje: **LoginOk (opcode = 2)**
Formato: OneParameter
Sentido de la comunicación: Directorio → Cliente
Descripción: Este mensaje lo envía el Directorio al cliente para confirmar que el "login" se ha realizado correctamente. El campo "Parámetro" es un entero (4 bytes) que en este caso contiene el número de servidores de ficheros que hay dados de alta en Directorio. El valor asignado al opcode es 2.
Ejemplo: Confirmación de login que indica que hay 0 servidores.

Opcode (1 byte)	Parámetro (4 bytes)
2	0

Mensaje: **LookupUsername (opcode = 3)**
Formato: TLV
Sentido de la comunicación: Cliente → Directorio
Descripción: Este mensaje lo envía el cliente al directorio para realizar la petición de recibir la IP y Puerto que utiliza un cierto usuario para servir sus ficheros. El valor asignado al opcode es 3.
Ejemplo:

Opcode (1 byte)	Length_Nick (4bytes)	Nick (n bytes)
3	4	Jose

Mensaje: **LookupUsernameFound (opcode = 4)**
Formato: TLV
Sentido de la comunicación: Directorio → Cliente
Descripción: Este mensaje lo envía el directorio al cliente como respuesta al mensaje LookupUser, informando de la cadena IP:PORT que contiene la IP y puerto en la que está sirviendo el usuario solicitado. El valor asignado al opcode es 4.
Ejemplo:

Opcode (1 byte)	Length_IP:PORT (4bytes)	IP:PORT (n bytes)
4	4	192.168.0.18:10000

Mensaje: **LookupUsernameNotFound (opcode = 5)**
Formato: Control
Sentido de la comunicación: Directorio→ Cliente
Descripción: Este mensaje lo envía el directorio al cliente como respuesta al mensaje LookupUser, informando de que no se ha podido resolver la cadena IP:PORT a partit del usuario solicitado (el usuario no era un servidor, o ni existe). El valor asignado al opcode es 5.
Ejemplo:

Opcode (1 byte)
5

Mensaje: **Register (opcode = 6)**
Formato: TLV
Sentido de la comunicación: Cliente → Directorio
Descripción: Este mensaje lo envía el cliente al directorio para realizar la petición de añadir su nick de la lista de usuarios. El único carácter no válido para un nick son los dos puntos. El valor asignado al opcode es 6.
Ejemplo:

Opcode (1 byte)	**Length_Nick (4bytes)**	**Nick (n bytes)**
6	4	Juan

Mensaje: **RegisterOk (opcode = 7)**
Formato: Control
Sentido de la comunicación: Directorio → Cliente
Descripción: Este mensaje lo envía el directorio al cliente para confirmar que el "register" se ha realizado correctamente. El valor asignado al opcode es 7.
Ejemplo:

Opcode (1 byte)
7

Mensaje: **RegisterFail (opcode = 8)**
Formato: Control
Sentido de la comunicación: Directorio → Cliente
Descripción: Este mensaje lo envía el directorio al cliente para notificar que no se pudo registrar el nombre de usuario, al estar este ya en uso por otro usuario. El valor asignado al opcode es 8.
Ejemplo:

Opcode (1 byte)
8

Mensaje: **ServeFiles (opcode = 9)**
Formato: IPPORT_FICH_VAR
Sentido de la comunicación: Cliente → Directorio
Descripción: Este mensaje lo envía el cliente al directorio para realizar la petición de empezar a funcionar como servidor con sus respectivos ficheros a compartir, así como metadatos (hash y tamaño en bytes). El directorio necesitará recibir el puerto (la IP la extrae del datagrama, no necesita estar en el mensaje) que utilizará el servidor para poder ofrecer al cliente dichos valores a partir del nick del servidor. El valor asignado al opcode es el 9.

Ejemplo:

Opcode (1 byte)	Length Nickname (4 bytes)	Nickname (n bytes)	Length PORT (4 bytes)	PORT (p bytes)	K Numero Campos (4 bytes)	Length Nombre 1 (4 bytes)
9	4	Juan	4	8667	2	6

Nombre 1 (f_1 bytes)	Length Hash 1 (4 bytes)	Hash 1 (h_1 bytes)	Size 1 (8 bytes)	Length Nombre 2 (4 bytes)
hola.c	5	e96et	2030	9

Nombre 2 (f_2 bytes)	Length Hash 2 (4 bytes)	Hash 2 (h_2 bytes)	Size 2 (8 bytes)
mundo.cpp	5	87y5r	1789

Mensaje: **ServeFilesOk (opcode = 10)**
Formato: Control
Sentido de la comunicación: Directorio → Cliente
Descripción: Este mensaje lo envía el directorio al cliente para notificar que la petición de empezar a funcionar como servidor se ha realizado correctamente. El valor asignado al opcode es 10.
Ejemplo:

Opcode (1 byte)
10

Mensaje: **ServeFilesStop (opcode = 12)**
Formato: TLV
Sentido de la comunicación: Cliente → Directorio
Descripción: Este mensaje lo envía el cliente al directorio para indicar que quiere dejar de actuar como servidor de ficheros y volver a ser un cliente. Al igual que en quit, se necesita pasar el nombre previamente guardado en un atributo al registrarse para que el directorio pueda eliminarlo del mapa de servidores. El valor asignado al opcode es 12.
Ejemplo:

Opcode (1 byte)	Length_Nick (4bytes)	Nick (n bytes)
12	4	Juan

Mensaje: **ServeFilesStopOK (opcode = 13)**
Formato: Control
Sentido de la comunicación: Directorio → Cliente
Descripción: Este mensaje lo envía el directorio al cliente para notificar que se ha eliminado el usuario como servidor del directorio correctamente. El valor asignado al opcode es 13.
Ejemplo:

Opcode (1 byte)
13

Mensaje: **GetUsers (opcode = 14)**
Formato: Control
Sentido de la comunicación: Cliente → Directorio
Descripción: Este mensaje lo envía el cliente al directorio para realizar la petición de la lista de los usuarios. El valor asignado al opcode es 14.
Ejemplo:

Opcode (1 byte)
14

Mensaje: **Userlist (opcode = 15)**
Formato: TLV_VAR
Sentido de la comunicación: Directorio → Cliente
Descripción: Este mensaje lo envía el directorio al cliente para enviar la lista de usuarios. El valor asignado al opcode es 15.
Ejemplo:

Opcode (1 byte)	K Numero Nicks (4 bytes)	Length 1 (4bytes)	Nick 1 (n_1 bytes)	Length 2 (4bytes)	Campo 2 (n_2 bytes)
15	2	15	Juan <SERVER>	6	Joselu

Mensaje: **GetFiles (opcode = 17)**
Formato: Control
Sentido de la comunicación: Cliente → Directorio
Descripción: Este mensaje lo envía el cliente al directorio para realizar la petición de la lista de los ficheros que tiene el directorio. El valor asignado al opcode es 17.
Ejemplo:

Opcode (1 byte)
17

Mensaje: **Filelist (opcode = 18)**
Formato: FICH_VAR
Sentido de la comunicación: Directorio → Cliente
Descripción: Este mensaje lo envía el directorio al cliente para enviar la lista de ficheros, indicando para cada uno su nombre. El valor asignado al opcode es 18.
Ejemplo:

Opcode (1 byte)	K Numero Ficheros (4 bytes)	Length Nombre1 (4bytes)	Nombre 1 (n_1 bytes)	Length Hash1 (4bytes)	Hash1 (h1 bytes)
18	**2**	**10**	**prueba.txt**	**5**	**3f4t6**

Size1 (8 bytes)	Length Nombre2 (4 bytes)	Nombre2 (n2 bytes)	Length Hash2 (4 bytes)	Hash2 (h2 bytes)	Size 2 (8 bytes)
34	**prueba2.txt**	**11**	**5**	**78f54**	**23**

Mensaje: **LogOff (opcode = 19)**
Formato: TLV
Sentido de la comunicación: Cliente → Directorio
Descripción: Este mensaje lo envía el cliente al directorio para realizar la petición de borrar su nick de la lista de usuarios, para después cerrar el programa. El nick lo guardará el cliente al registrarse, no podrá escribir el nick el propio usuario. El valor asignado al opcode es 19.
Ejemplo:

Opcode (1 byte)	Length_Nick (4bytes)	Nick (n bytes)
19	4	Juan

Mensaje: **Quit (opcode = 20)**
Formato: Control
Sentido de la comunicación: Cliente → Directorio
Descripción: Este mensaje lo envía el directorio al cliente para confirmar que se ha borrado el nick y se puede cerrar el programa. El valor asignado al opcode es 20.
Ejemplo:

Opcode (1 byte)
20

2.2 Formato de los mensajes del protocolo de comunicación entre cliente y servidor de ficheros

NOTA: Un mismo formato de mensaje puede ser utilizado por varios tipos de mensajes diferentes (operation diferentes)

Formatos de mensajes

Formato: Control

```
operation: xxxxxxx\n
\n
```

Formato: FileHash

```
operation: xxxxxxx\n
hash: xxxxxxxx\n
\n
```

Formato: File

```
operation: xxxxxxx\n
seq: xxxxxxxxxxxx\n
data: xxxxxxxxxxxx\n
\n
```

Formato: ServedFiles

```
operation: xxxxxxx\n
fileName: xxxxxxxxxxx\n
fileSize: xxxxxxxxxxx\n
fileHash: xxxxxxxxxxx\n
...
fileName: xxxxxxxxxxx\n
fileSize: xxxxxxxxxxx\n
fileHash: xxxxxxxxxxx\n
...
\n
```

Tipos y descripción de los mensajes

Mensaje: **Download**
Formato: FileHash
Sentido de la comunicación: Cliente→ Servidor de ficheros
Descripción: Este mensaje lo envía el cliente al servidor de ficheros para solicitar la descarga de un fichero cuyo hash viene dado en el campo “filehash”.
Ejemplo:

```
operation: download\n
filehash: 87071b18127a076887613fa2b38511ec\n
\n
```

Mensaje: **FileNotFound**
Formato: Control
Sentido de la comunicación: Servidor de ficheros → Cliente
Descripción: Este mensaje lo envía el servidor de ficheros al cliente para informar de que el fichero solicitado no ha sido encontrado.
Ejemplo:

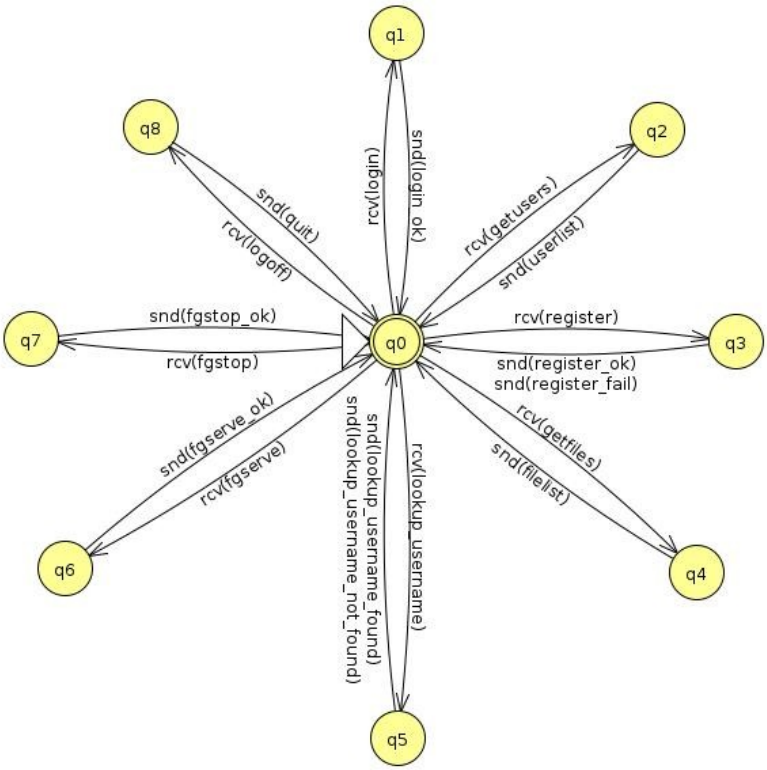
```
operation: filenotfound\n
\n
```

Mensaje: **File**
Formato: File
Sentido de la comunicación: Servidor de ficheros → Cliente
Descripción: Este mensaje lo envía el servidor al cliente como respuesta a una solicitud de descarga válida, enviando los datos del fichero como bytes en el campo data. Puesto que habrá ficheros que necesitarán enviar sus datos en varios mensajes debido a su elevado tamaño, se añade el campo seq que indica el orden de los conjuntos de bytes de manera descendente (…, 2, 1, 0), de manera que el cliente espera a que llegue el siguiente de manera ordenada.

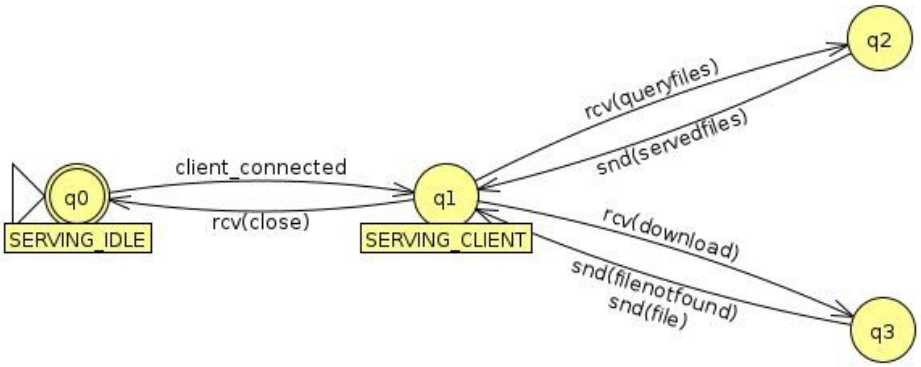
```
operation: file\n
seq: 2\n
data: cHJ1ZWJhIHBhcmEgbGEgZG9jdW1lbnRhY2nDs24K\n
\n
```

2.3 Autómatas de protocolo

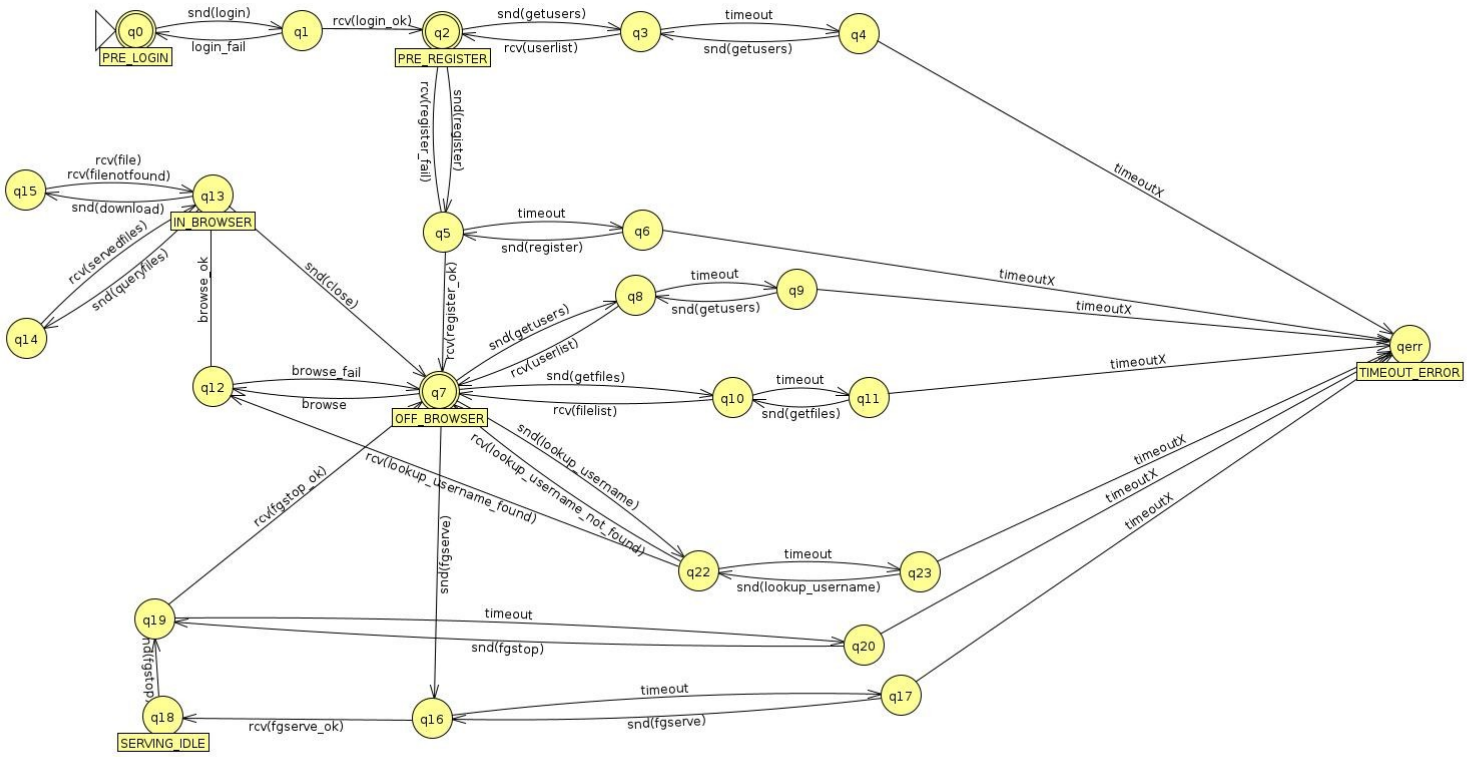
Autómata Servidor de Directorio



Autómata Servidor de ficheros



Autómata Cliente (Unificado UDP y TCP)



NOTA IMPORTANTES:

- Para evitar complicar aun más el autómata, se han omitido las transiciones debidas a la orden quit. Esta se puede ejecutar en los estados que se han marcado como finales, por lo que desde q0, q2 y q7 tendríamos una transición a un nuevo estado con snd(logoff), de este estado podemos o bien recibir rcv(quit) en cuyo caso pasamos al estado final, o un timeout con su estado correspondiente desde el que o se vuelve a enviar el logOff o salta el timeoutX y finaliza el programa con TIMEOUT_ERROR.

- Los autómatas cliente y servidor son en el fondo el mismo, el punto de unión siendo el estado marcado con SERVING_IDLE.

2.4 Ejemplo(s) de intercambio de mensajes

Ejemplo 1: DIRECTORIO SE CAE

Cliente: **snd(login)** / Campos: opcode = 1

Ahora mismo el autómata nos obliga a esperar una respuesta del directorio, o en su defecto que no se reciba respuesta. En este caso, suponemos que no se consigue contactar con el directorio, por lo que volvemos al PRE_LOGIN. Lo volvemos a intentar:

Cliente: **snd(login)** / Campos: opcode = 1

Directorio: **rcv(login) → snd(login_ok) /** Campos: opcode = 2, servidores = 1

Ahora si se ha podido conectar con el directorio, y pasamos a PRE_REGISTER. Cabe destacar que no limitamos los intentos; un cliente que inicie NanoFiles puede reintentar conectarse a un directorio que no responde todas las veces que quiera.

Cliente: **rcv(login_ok) → snd(getusers)** / Campos: opcode = 14

Directorio: **rcv(getusers) → snd(userlist)** / Campos: opcode = 15, num_nicks = 2, longitud1 = 4, nombre1 = juan, longitud2 = 6, nombre2 = joselu

Permitimos que un usuario todavía no registrado pueda mirar los nicks registrados para facilitar el registro (evitar duplicados).

Cliente: **rcv(userlist) → snd(register) → timeout** → (timeout X veces) → **timeoutX → FIN**

Aquí lo que ha sucedido es que durante la ejecución del cliente, concretamente en algun momento entre el recibido de la lista de usuarios y el envío de la solicitud de registro, el directorio se ha caído. El cliente intenta conectar repetidamente con el directorio, y en el intento X desiste, finalizando el programa.

Ejemplo 2: USUARIO CLIENTE

Cliente: **snd(login)** / Campos: opcode = 1

Directorio: **rcv(login) → snd(login_ok)** / Campos: opcode = 2, servidores = 1

Cliente: **rcv(login_ok) → snd(register)** / Campos: opcode = 6, longitud = 4, nombre = juan

Directorio: **rcv(register) → snd(register_fail)** / Campos: opcode = 8

Cliente: **rcv(register_fail) → snd(register)** / Campos: opcode = 6, longitud = 4, nombre = pepe

Directorio: **rcv(register) → snd(register_ok)** / Campos: opcode = 7

Cliente: **rcv(register_ok) → snd(getfiles)** / Campos: opcode = 17

Directorio: **rcv(getfiles) → snd(filelist)** / Campos: opcode = 18, num_ficheros = 1, longitud_nombre1, nombre1 = a.py, longitud_hash1 = 6, hash1 = 67hgt3, size1 = 6754

Ahora se ha realizado el login y el register sin problema ninguno, llegando hasta el estado OFF_BROWSER. En este, el cliente sigue pudiendo consultar la lista de usuarios (no hay motivo para lo contrario) y además puede consultar la lista de ficheros, cosa que no se permitía en PRE_REGISTER ya que eso sí lo consideramos una violación de la seguridad / privacidad de los usuarios servidores de NanoFiles.

Cliente: **rcv(filelist) → snd(lookupUsername)** / Campos: opcode = 3, longitud = 5, nick = joose

Servidor: **rcv(lookupUsername) → snd(lookupUsernameNotFound)** / Campos: opcode = 5

El usuario intenta conectarse a un servidor, pero introduce mal el nick en el comando browse.

Cliente: **rcv(lookupUsernameNotFound) → snd(lookupUsername)** / Campos: opcode = 3, longitud = 4, nick = jose

Servidor: **rcv(lookupUsername) → snd(lookupUsernameFound)** / Campos: opcode = 4, ipport = 192.168.0.18:10000

El usuario introduce los datos correctamente esta vez y se conecta al servidor satisfactoriamente, desplazándonos al estado IN_BROWSER. Destacamos como a partir de ahora, y hasta que no cierre el browser con el comando close, el usuario no podrá ejecutar el comando quit y por consiguiente cerrar el programa.

Cliente: **rcv(serverOk) → snd(download)** / Campos: operation = download, hash = ika7saDG7

Servidor: **rcv(download) → snd(file)** / Campos: operation = file, seq = 0, data = 01010110…

Cliente: **rcv(file) → snd(download)** / Campos: operation = download, hash = 173y92

Servidor: **rcv(download) → snd(filenotfound)** / Campos: operation = filenotfound

El cliente realiza dos solicitudes de descarga; la primera válida y la segunda no (el fichero no existe).

Cliente: **snd(close)**

Cliente: **snd(logoff)** / Campos: opcode = 19

Directorio: **rcv(logoff) → snd(quit)** / Campos: opcode = 20

Cliente: **rcv(quit)** → FIN

Ejemplo 3: USUARIO SERVIDOR Y DESPUÉS CLIENTE

Cliente: **snd(login)** / Campos: opcode = 1

Directorio: **rcv(login) → snd(login_ok)** / Campos: opcode = 2, servidores = 0

Cliente: **rcv(login_ok) → snd(register)** / Campos: opcode = 6, longitud = 3, nombre = ana

Directorio: **rcv(register) → snd(register_ok)** / Campos: opcode = 7

Cliente: **rcv(register_ok) → snd(fgserve)** / Campos: opcode = 9

Directorio: **rcv(fgserve) → snd(fgserve_ok)** / Campos: opcode = 10

El usuario pasa de ser un cliente a un servidor de ficheros. Cabe destacar que, una vez convertido en servidor, tenemos dos estados distintos: SERVING_IDLE y SERVING_CLIENT. El primero se refiere a cuando ya hemos ejecutado la orden fgserve pero no estamos en conversación con ningún cliente, y el segundo cuando sí estamos atendiendo las solicitudes de un cliente.

Servidor: **rcv(fgserve_ok)**

Servidor: **client_connected**

Cliente (otro): **snd(download)** / Campos: operation = download

Servidor: **rcv(download) → snd(filenotfound)** / Campos: operation = filenotfound

Cliente (otro): **rcv(filenotfound) → snd**(**close)**

Servidor: **rcv(close)**

Cuando un servidor esta sirviendo a un cliente y es este el que cierra la comunicación, el servidor vuelve automáticamente al estado SERVING_IDLE.

Servidor: **snd(fgstop)** / Campos: opcode = 12, longitud = 3, nombre = ana

Directorio: **rcv(fgstop) → snd(fgstop_ok)** / Campos: opcode = 13

Como podemos ver, un servidor puede volver a ser un cliente, y también ejecutar otra vez fgserve, alternando entre cliente y servidor todas las veces que quiera. Eso sí, sólo puede ser una cosa a la vez y necesita eventualmente volver a ser cliente para poder cerrar el programa.

Cliente: **rcv(fgstop_ok) → snd(logoff)** / Campos: opcode = 19

Directorio: **rcv(logoff) → snd(quit)** / Campos: opcode = 20

Cliente: **rcv(quit) → FIN**

3. Mejoras Implementadas

3.1 Comando filelist

El comando filelist necesita de dos mapas en la clase **DirectoryThread**: *files* y *owners*. El primero se trata de un mapa que vincula el hash de cada fichero con su objeto FileInfo. De esta manera, al enviar la información del objeto FileInfo de vuelta al usuario cuando este introduce el comando filelist, no sólo mostramos el nombre sino también el hash y tamaño de cada fichero. El segundo mapa asocia cada hash con el nick del servidor que lo tiene, lo que resulta necesario para poder eliminar los ficheros asociados a un cierto servidor cuando este deja de servir y vuelve a ser un cliente. En cuanto a los mensajes requeridos, el de solicitud es un sencillo mensaje de control y el de respuesta es similar al de la solicitud para servir ficheros, pero sin la necesidad de incluir también nickname y puerto.

3.2 Comando browse <username>

Para implementar esta mejora, una vez se reconoce el comando browse, en la función *browserEnter* de la clase **NFControllerLogicP2P** hay que distinguir si el argumento pasado al comando es un nickname o una cadena IP:PORT, lo cual es fácil de averiguar, ya que no se permite que un nickname contenga el carácter ":". Si resulta que el argumento es un nickname, entonces se envía el mensaje LookupUsername, de formato idéntico a Register, y entonces el Directorio busca ese nickname en su mapa *servers*, que asocia el nick de cada servidor con su objeto InetSocketAddress. Para la respuesta, el directorio convierte el objeto a una cadena IP:PORT y la envía al cliente.

3.3 Mejora en el comando userlist

A la hora de mostrar la lista de usuarios, es muy fácil añadir quién es servidor gracias a la existencia del mapa *servers* en el Directorio. Lo único que hay que hacer es, a la hora de construir el mensaje de respuesta (mensaje Userlist), ir insertando uno a uno los nombres de los nicks en el mensaje y, en el caso de estar ese nick también en el mapa *servers*, añadir a su nombre la cadena SERVING_IDENTIFIER. De esta manera el cliente que recibe la lista de usuarios no tiene que preocuparse por cuáles son servidores o no, simplemente va recibiendo los nombres, construye la lista y la muestra; si un usuario es servidor o no ya viene contenido en el nombre que le llega.

3.4 Comando fgstop

Para permitir que un servidor sin ningún cliente conectado pudiese dejar de servir, le ponemos un timeout al *accept*, y una vez salta el timeout comprobamos con un objeto *BufferedReader* si se ha escrito fgstop. De ser el caso, se activa un booleano *stopServer* que nos saca de los bucles de la clase **NFServerSimple**.

3.5 Comando queryfiles

El comando queryfiles es análogo en cuanto a formato al comando filelist, pero esta vez en formato Campo:Valor, lo cual simplifica significativamente las cosas. Esta vez, en vez de extraer los objetos FileInfo de un mapa en el directorio, una vez le llega el mensaje de petición al servidor, este crea un array de FileInfo con la información de todos los ficheros que está sirviendo, gracias a la función *NanoFiles.db.getFiles()*, y forma el mensaje a partir de dicho array. Este mensaje es más sencillo que el binario ya que no hace falta incluir ni el número de ficheros ni las longitudes de cada nombre y hash.

3.6 Actualización de la información del Directorio

Para mantener la información (es decir, los mapas) del directorio actualizada correctamente, hay que tener una serie de cosas en cuenta en la clase **DirectoryThread**, más específicamente cuando se reconoce el tipo de mensaje recibido en la clase *processRequestFromClient*: si se trata de un Register, añadimos el nuevo nickname al mapa *nicks*, cuando se recibe un ServeFiles se introduce, para cada hash de cada fichero, una entrada en *files* con su objeto FileInfo y una entrada en *owners* con el nickname recibido en el mensaje, cuando se recibe un ServeFilesStop, se recorre el mapa *owners* y, si el nombre del propietario del fichero coincide con el nombre del usuario que ha enviado el mensaje, se eliminan las entradas correspondientes de *files* y *owners*, y finalmente, cuando se recibe un LogOff se elimina el nickname de *nicks.* No es necesario revisar *servers* ya que, para que un servidor pueda ejecutar quit debe primero ejecutar fgstop y, además de actualizar los mapas *files* y *owners* como ya hemos descrito, se elimina la entrada correspondiente al nickname del servidor en el mapa *servers*.

4. Intercambio de mensajes Peer-Servidor con Wireshark

```
..operation:getFiles

.Moperation:servedFiles
name:prueba2.txt
size:11448
hash:eb23a903f15d2ceb8812478c60e75f3660201441
name:hash.txt
size:19
hash:a0c2b68770149e00e49f3ebf3da3eb20cd8af5f1
name:AphexTwin - Stone in focus -  Philosophical Ape 2 hours.mp4
size:248189256
hash:217e73860d7b217574afc59d1ac82fb6c1c8992d
name:prueba.txt
size:54
hash:6cb638eb2a2d309c22c57b51ae867868e2332296
name:100MB.bin
size:104857600
hash:2c2ceccb5ec5574f791d45b63c940cff20550f9a
name:memoria.odt
size:564586
hash:9ddd367f1fd2e1201bc42eb3fc47a4ad9fc449a4
name:ESCLAVA.mp4
size:21306253
hash:25a98e6b60571d67542ecedc5b1f4cac4e6ea44c

.Boperation:download
hash:6cb638eb2a2d309c22c57b51ae867868e2332296

.doperation:file
data:UFJVRUJBCkVzVGFNb1MgUFJPQkFORE8hIQoxMjM0NSAvLyAxMgotLWJ5ZQoKc2Ugdmll bmUK
seq:0

..operation:close
```

5. Conclusiones

Este ha sido, al mismo tiempo, un proyecto altamente disfrutable así como frustrante. Disfrutable, ya que nos ha permitido entender de manera práctica y realista cómo funciona la comunicación entre procesos en la capa de transporte, no sólo entendiéndola sino diseñándola e implementándola. Además, al ser nuestra primera aplicación con un uso práctico y tangible en lo que llevamos de carrera, era inmensamente satisfactorio cuando conseguíamos añadir correctamente cualquier funcionalidad y comprobamos que realmente funcionaba de la manera esperada.

Sin embargo, como ya hemos mencionado este proyecto también nos ha proporcionado bastante frustración, debido a la complejidad del código (la mayoría de las veces no porque las funciones y métodos fuesen complejos en sí, sino debido a la cantidad de paquetes, clase y objetos presentes en el proyecto, así como todas las relaciones entre ellos a tener en cuenta) y a algunos aspectos más tediosos (como, por ejemplo, el *parseo* de mensajes y la repetición de muchos de los mismos pasos cada vez que incluíamos un nuevo mensaje, por muy simple que fuese).

Aun así, en general la experiencia ha sido positiva y nos ha gustado desarrollar una aplicación funcional y que hace uso de Internet y los diferentes protocolos de redes como lo es NanoFiles.