REDES DE COMUNICACIONES

MEMORIA PROYECTO NANOFILES

UNIVERSIDAD DE MURCIA

GÁLVEZ DÍAZ, JOAQUÍN
URBELZ ALONSO-CORTÉS, JORGE

REDES DE COMUNICACIONES
GRUPO 3.1 y 3.3
CURSO 2024/25 - ENERO
GAONA RAMÍREZ, EPIFANIO

Contenido

Introducción

Para una buena realización de la práctica de NanoFiles, debemos tener primero en cuenta el funcionamiento de esta. Además, es nuestra labor como creadores de la práctica poner en contexto a los corregidores de las distintas ideas que hemos tenido durante el diseño de esta. Es por esto en esta entrega vamos a dejar constancia de los distintas partes de la práctica que hemos ido trabajando. Por un lado, dejaremos definido el formato de los mensajes de cada uno de los protocolos: en la comunicación con el Directorio (ClienteßàDirectorio) y en la transferencia de ficheros (Cliente actuando como PeerßàCliente).

Debido a que el proyecto lo hemos entregado en enero, decidimos en verano hacer las mejoras opcionales que los profesores nos proponían. Hemos considerado hacer todas, menos las del comando download. Vamos a explicar cada una de ellas.

- El puerto variable de fgserve (0,5 puntos): ahora podremos iniciar un servidor en primer plano en distintos puertos. En el proyecto se nos pedía que se iniciase de manera estándar en el puerto 10000. Por cuestión de diseño, creamos un bucle for en el cual el usuario intentará conectarse en el puerto 10000 si está vacío. Si no, lo intentará con el 10001. Y así sucesivamente hasta un máximo de 10 intentos. Si llega al tope, desistirá.
- Buscar por nickname en downloadfrom: ahora el Cliente podrá escribir en el comando el nickname (antes debía escribir la IP y el puerto de la máquina donde el servidor se alojaba) para descargar archivos. Esta mejora pasó a ser obligatoria en la convocatoria de junio.
- Bgserve en secuencial y multihilo: hemos realizado el comando bgserve desde cero. Además, ahora podrá escuchar a más de un Cliente a la vez, ya que crea un hilo para cada conexión. Estas mejoras pasaron a ser obligatorias en la convocatoria de junio.
- Fgstop y stopserver (0,5 puntos): para parar los servidores que estén corriendo los Peers, deben usar fgstop (en el caso de que haya iniciado un servidor en primer plano) o stopserver (si inició uno en segundo plano).
- Bgserve en puerto efímero (0,5 puntos): cuando un Cliente quiere iniciar un servidor en segundo plano, este se alojará en un puerto aleatorio.
- Ampliación comando userlist (0,5 puntos): cuando un Cliente o un Peer quiera saber la lista de usuarios conectados al Directorio, usará userlist. La ampliación permite saber si ese usuario es Peer (solamente si ha usado el comando bgserve).
- Comandos publish, filelist y search (0,5 puntos): lo bonito de usar el comando bgserve viene cuando haces los demás comandos opcionales. El comando publish permite publicar al Directorio los archivos que el Peer está compartiendo. Así, otros Clientes o Peers podrán preguntarle al Directorio que archivos pueden descargar (filelist también muestra que Peer ha publicado cada archivo) y buscar que Peers han compartido un archivo en concreto (buscando por el hash o una subcadena del hash) usando el comando search.
- Ampliación filelist y publish (0,5 puntos): como he dicho en el anterior párrafo, el comando filelist muestra que Peer está compartiendo cada archivo en concreto. Además, los usuarios podrán usar el comando search escribiendo una parte del hash o completo.

- Ampliación stopserver (0,5 puntos): cuando un Peer quiere dejar de compartir y apagar su servidor en segundo plano, se lo anunciará al Directorio. El Directorio actualizará la lista de archivos compartidos junto al nombre de su Peer. Esta mejora tan solo funciona usando el comando filelist, como se nos pide en los comentarios del código.

Formato de los mensajes del protocolo de comunicación con el Directorio

En esta fase del diseño aplicaremos las distintas decisiones que hemos tomado para el envío de mensajes entre el Cliente y el Directorio. La base de la comunicación ronda sobre el Cliente, ya que será este el que se encargue de enviar un mensaje y esperar su respectiva respuesta. El Directorio solamente enviará mensajes como respuesta a los mensajes que reciba de cada Cliente.

Tipos y descripción de los mensajes

Mensaje: login
Sentido de la comunicación: Cliente → Directorio
Descripción: Este mensaje lo envía el Cliente de NanoFiles al Directorio para solicitar "iniciar sesión" y registrar el nickname indicado en el mensaje. Este mensaje es necesario para que cualquier usuario inicie sesión con su nombre en el Directorio.
Ejemplo:

```
operation: login\n
nickname: pepe\n
\n
```

Mensaje: loginOK
Sentido de la comunicación: Directorio → Cliente
Descripción: Cuando el Directorio reciba el mensaje, tiene que comprobar que el usuario no está
registrado en el Directorio. Si no está registrado, le asigna una clave de sesión y se la envía mediante un mensaje al usuario, que la necesitará para solicitudes posteriores.
Ejemplo:

```
operation: loginOK\n
sessionKey: 5000\n
\n
```

Mensaje: loginFAIL
Sentido de la comunicación: Directorio → Cliente
Descripción: En caso de que la operación "login" resulte en un error, debido a, por ejemplo, que ya existe otro usuario con ese nombre, el Directorio debe enviar al Cliente un mensaje expresando que se éste ha producido. Debido a que la creación de usuario con su correspondiente nickname ha resultado en error, no se le vinculará ningún sessionKey.
Ejemplo:

```
operation: loginFAIL\n
sessionKey: -1\n
\n
```

Mensaje: logout
Sentido de la comunicación: Cliente → Directorio
Descripción: Un usuario enviaría este mensaje al Directorio. Este mensaje es necesario para que cualquier usuario que esté conectado al Directorio cierre su sesión.
Ejemplo:

```
operation: logout\n
sessionKey: 5000\n
\n
```

Mensaje: logoutOK
Sentido de la comunicación: Directorio → Cliente
Descripción: Cuando el Directorio reciba el mensaje "logout", tiene que comprobar que el usuario está registrado en el Directorio. Si está registrado, eliminará su nickname y clave de sesión asociada en el Directorio.
Ejemplo:

```
operation: logoutOK\n
\n
```

Mensaje: logoutFAIL
Sentido de la comunicación: Directorio → Cliente
Descripción: En el caso de que el usuario no esté registrado deberá mandar un mensaje loginFAIL informando de que este usuario (sessionKey) no existe.
Ejemplo:

```
operation: logoutFAIL\n
\n
```

Mensaje: registeredUsers
Sentido de la comunicación: Cliente → Directorio
Descripción: Este mensaje lo envía el Cliente de NanoFiles al Directorio cuando quiere saber los usuarios registrados y cuales de ellos son Peer. Realmente el usuario escribe en la terminal "userlist", pero se envía al Directorio "registeredUsers" debido a que pensamos que era una buena manera de dejar claro que habíamos realizado la mejora del comando. Como tal, se pedía que devolviese una lista de usuarios registrados. Si añadías la mejora, te dice si usuario es Peer, o no.
Ejemplo:

```
operation: registeredUsers \n
\n
```

Mensaje: registeredUsersResp
Sentido de la comunicación: Directorio → Cliente
Descripción: Cuando el Directorio reciba el mensaje "registeredUsers", buscará en un hash si cada uno de los usuarios registrados es Peer, o no. Esto solo funciona para aquellos Clientes que hagan uso del comando "bgserve". Si utilizan "fgserve", seguirá apareciendo como "false". Esta idea fue dada por los profesores.
Ejemplo:

```
operation: registeredUsersResp\n
sessionKey: 5000\n
user: pepe\n
isPeer:true\n
user: juan\n
isPeer:false\n
…
user: alicia\n
isPeer:false\n
\n
```

Mensaje: registerServer

Sentido de la comunicación: Cliente → Directorio
Descripción: Este mensaje lo envía el Cliente de NanoFiles al Directorio cuando el Cliente ejecuta la orden "bgserve" (se convierte en servidor Peer). La función del Directorio es llevar un registro actualizado de los usuarios que se convierte en Peer.
Ejemplo:

```
operation: registerServer\n
sessionKey: 5000\n
port: 1080\n
\n
```

Mensaje: registerServerOK
Sentido de la comunicación: Directorio → Cliente
Descripción: Este mensaje lo envía el Directorio cuando uno de los Clientes pasa a ser Peer y simplemente queremos devolver la confirmación. En este caso, el Cliente no producirá ningún fallo al convertirse en Servidor de ficheros, por lo que nos centramos en devolver un mensaje de confirmación del resultado. El Directorio guardará en un hash la correspondencia entre la clave de sesión y el puerto en el que se ha montado el socket del servidor de fondo.
Ejemplo:

```
operation: registerServerOK\n
\n
```

Mensaje: stopServer
Sentido de la comunicación: Cliente → Directorio
Descripción: Este mensaje lo envía el Cliente cuando quiere parar el servidor de fondo que había lanzado anteriormente. Para que el Directorio sepa que puerto estaba ocupando y así poder eliminarlo del hash, el Cliente le hará llegar su clave de sesión.
Ejemplo:

```
operation: stopServer\n
sessionKey: 5000\n
\n
```

Mensaje: stopServerOK
Sentido de la comunicación: Directorio → Cliente
Descripción: Este mensaje lo envía el Directorio cuando quiere confirmar al Cliente la acción de borrar su puerto del hash. De esta manera, si el Cliente tenía montado un servidor de fondo y decide pararlo, debe avisar al Directorio que deja de ser Peer y, por tanto, ya puede borrar el puerto de su socket del hash.
Ejemplo:

```
operation: stopServerOK\n
\n
```

Mensaje: stopServerFAIL
Sentido de la comunicación: Directorio → Cliente
Descripción: Este mensaje lo envía el Directorio cuando quiere decirle al Cliente que no fue capaz de borrar su puerto del socket de hash. Como el Directorio no ha sido capaz de encontrar el puerto, no puede borrarlo.
Ejemplo:

```
operation: stopServerFAIL\n
```

```
\n
```

Mensaje: getAddrFromNick
Sentido de la comunicación: Cliente → Directorio
Descripción: Este mensaje lo envía el Cliente cuando quiere obtener la dirección IP y el puerto del socket de un Peer. Para ello, enviará dentro del mensaje el nombre de usuario del Peer que está buscando.
Ejemplo:

```
operation: getAddrFromNick\n
nickname: pepe\n
\n
```

Mensaje: getAddrResp
Sentido de la comunicación: Directorio → Cliente
Descripción: Este mensaje lo envía el Directorio cuando un Cliente busca encontrar la dirección de IP y puerto asociado al socket de un Peer. De esta manera, cuando el Directorio recibe un mensaje "getAddrFromNick", buscará en un hash el IP y el puerto asociado al socket donde el Peer tiene montado su servidor de fondo.
Ejemplo:

```
operation: getAddrResp\n
ip: 127.0.0.1\n
port: 1080\n
\n
```

Mensaje: getAddrFAIL
Sentido de la comunicación: Directorio → Cliente
Descripción: Este mensaje lo envía el Directorio cuando en el procedimiento anterior el Directorio no es capaz de encontrar los datos necesarios para montar el mensaje. Esto puede ocurrir debido a que el usuario que le pasa el Cliente resulta no ser un Peer, por lo que su información no se encuentra almacenada dentro de los hashes.
Ejemplo:

```
operation: getAddrFAIL \n
\n
```

Mensaje: publish
Sentido de la comunicación: Cliente → Directorio
Descripción: Este mensaje lo envía el Cliente cuando quiere publicar al Directorio los archivos que está compartiendo. Solo podrán hacer uso de este comando aquellos Clientes que sean Peer, ya que no tendría sentido que cualquier Cliente publicase sus archivos al Directorio.
Ejemplo:

```
operation: publish\n
sessionKey: 5000\n
numFiles: 2\n
publish:resultado.txt+31+4731f36b5e5bca2ce7a139b4ab0cc1ca44c1eb6
a, ,foto.png+20657+5870dafa799721697a144c8bb845d6e83ce80685\n
\n
```

Mensaje: publishOK
Sentido de la comunicación: Directorio → Cliente

Descripción: Este mensaje lo envía el Directorio cuando quiere confirmar al Cliente que se han publicado sus archivos correctamente.
Ejemplo:

```
operation: publishOK\n
\n
```

Mensaje: publishFAIL
Sentido de la comunicación: Directorio → Cliente
Descripción: Este mensaje lo envía el Directorio cuando en el procedimiento anterior ha ocurrido algún tipo de error a la hora de enviarse desde el Cliente al Directorio. Si no tiene ningún archivo a compartir, no se le permite realizar el comando.
Ejemplo:

```
operation: publishFAIL\n
\n
```

Mensaje: getFileList
Sentido de la comunicación: Cliente → Directorio
Descripción: Este mensaje lo envía el Cliente cuando quiere obtener la lista de ficheros publicados por los Peer al Directorio. Para que la lista de ficheros perteneciente al Directorio tenga archivos, primero los Peer deben haber hecho uso del comando "publish".
Ejemplo:

```
operation: getFileList\n
\n
```

Mensaje: getFileListResp
Sentido de la comunicación: Directorio → Cliente
Descripción: Este mensaje lo envía el Directorio al Cliente con toda la información de los archivos hasta el momento publicados. Hemos realizado la mejora opcional: junto a la información de cada archivo, se añadirá el Peer poseedor de ese archivo en concreto.
Ejemplo:

```
operation: getFileListResp\n
numFiles: 2\n
published:resultado.txt+31+4731f36b5e5bca2ce7a139b4ab0cc1ca
44c1eb6a+pepe,foto.png+20657+5870dafa799721697a144c8bb845d6
e83ce80685+pepe\n
\n
```

Mensaje: getFileListFAIL
Sentido de la comunicación: Directorio → Cliente
Descripción: Este mensaje lo envía el Directorio al Cliente que solicitó la lista de archivos publicados. El error se produce porque el hash que contiene la lista está vacío, ya que ningún Peer ha publicado nada.
Ejemplo:

```
operation: getFileListFAIL\n
\n
```

Mensaje: getSearched
Sentido de la comunicación: Cliente → Directorio

Descripción: Este mensaje lo envía el Cliente cuando quiere disponer del nombre de los Peer que hayan publicado, entre sus archivos, uno específico. El Cliente enviará al Directorio el hash o una subcadena de este del archivo que quiere descargar.
Ejemplo:

```
operation: getSearched\n
fileHash: b1g2fh3jy52nk4\n
\n
```

Mensaje: getSearchedResp
Sentido de la comunicación: Directorio → Cliente
Descripción: Este mensaje lo envía el Directorio al Cliente un Array de cadenas con el nickname de aquellos Peer que tengan el archivo que el Cliente solicitó. El Directorio tiene un hash donde almacena la información de los Peer y los archivos que tiene. Así, el Directorio puede buscar en el hash por un archivo específico (usando el hash del archivo que envía el Clilente), y así obtener el nickname de los Peer.
Ejemplo:

```
operation: getSearchedResp\n
server: [pepe, juan]\n
\n
```

Mensaje: getSearchedFAIL
Sentido de la comunicación: Directorio → Cliente
Descripción: Este mensaje lo envía el Directorio al Cliente cuando ha ocurrido un error a la hora de buscar el archivo. El error pudo ser provocado porque el hash donde se busca está vacío, o porque no hay ningún archivo con el hash que envía el Cliente.
Ejemplo:

```
operation: getSearchedFAIL\n
\n
```

Formato de los mensajes del protocolo de transferencia de ficheros

Ahora os detallaremos todos aquellos mensajes enviados entre un Cliente y un servidor de ficheros Peer. De momento, solo se comunicarán cuando el Cliente quiera descargar un fichero del servidor y este le conteste afirmativamente o con un mensaje de error. Como ocurría con la comunicación entre el Cliente y el Directorio, el servidor únicamente se comunicará con el Cliente para enviarle una respuesta, ya sea afirmativa o negativa.

Estos mensajes pueden tener un tamaño distinto, dependiendo de la información que decidamos enviar en cada mensaje. Algunas veces, nos interesará enviar un mensaje de control (un mesaje simple con un único campo de un byte para el opcode o código de operación) Otras, sin embargo, enviaremos más que eso. Por ejemplo, el envío del hash de un fichero será pieza clave de nuestro protocolo de transferencia de ficheros. Por ello, el código de operación se encontrará acompañado de uno o más parámetros. El lector de esta memoria podrá comprobar que, en algunos casos, hemos enviado información que no resulta importante o necesaria, pero queríamos asegurarnos de poder tener al alcance todos los datos posibles.

El formato será el siguiente para los mensajes con un solo parámetro (el código de operación):

Opcode (1 byte)

En el resto de los casos, se añadirán más parámetros, en función de lo que se exija para una buena comunicación. En cuanto a los bytes que requiere cada parámetro, dependerá del tamaño de lo que queramos enviar. No es lo mismo, un trozo de hash de un fichero, que el fichero en sí:

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

Opcode (1 byte)	**Parámetro 1 (m bytes)**	**Parámetro n (m bytes)**

Tipos y descripción de los mensajes

Mensaje: InvalidOpCode (opcode=0)
Sentido de la comunicación: Servidor de ficheros → Cliente y Cliente → Servidor de ficheros
Descripción: Este mensaje puede ser enviado en ambos sentidos y se usa cuando se envía previamente un opcode inválido debido a que no existe o no hay ninguno asignado a ese byte.
Ejemplo:

Opcode (1 byte)
0

Mensaje: DownloadFrom (opcode=1)
Sentido de la comunicación: Cliente → Servidor de ficheros
Descripción: Este mensaje es enviado por el Cliente a un servidor de ficheros cuando quiera obtener un fichero. Para ello, deberá enviar un array de bytes, el cual contendrá el hash o trozo de hash desde el cual partimos.
Ejemplo:

Opcode (1 byte)	Hash (n bytes)
1	9348afj3

Mensaje: DownloadFromRespHS (opcode=2)
Sentido de la comunicación: Servidor de ficheros → Cliente
Descripción: Este mensaje es enviado por el servidor de ficheros a un Cliente cuando la operación anterior (DownloadFrom) ha sido aceptada. Mediante este mensaje, el servidor le indica al Cliente cuantos fragmentos del fichero le enviará y el hash completo del fichero solicitado. Se trata de una especie de handshake. Primero se envía este mensaje y a continuación, tantos mensajes "DownloadFromResp" como fragmentos de fichero se hayan acordado.
Ejemplo:

Opcode (1 byte)	FragmentosFichero (long)	Hash (n bytes)
2	3	9348afj3…

Mensaje: DownloadFromResp (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 proporcionarle un fragmento del contenido del fichero solicitado. El tamaño máximo de un fragmento de fichero son 8KB, ya que nos ofrece un equilibrio entre la eficiencia del uso de la memoria y la velocidad de transmisión, al ser suficientemente pequeño para no saturar la memoria y suficientemente grande para mantener la transmisión eficiente y fluida.
Ejemplo:

Opcode (1 byte)	Datos (n bytes)
3	El Renacimiento…

Mensaje: DownloadFromWhich (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 el trozo de hash que se le ha enviado coincide con el hash de varios ficheros.
De esta manera, el servidor le da la opción al cliente de seleccionar el fichero adecuado.
Para ello, deberá volver a enviar un "DownloadFrom" con el hash completo del fichero que solicita.
Ejemplo:

Opcode (1 byte)	Hashes (n bytes)	NombresFicheros (n bytes)
4	f93e35… , 9348af…	factura1.pdf , factura2.pdf

Mensaje: DownloadFromFAIL (opcode=5)
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 el fichero solicitado no está disponible.
Ejemplo:

Opcode (1 byte)
5

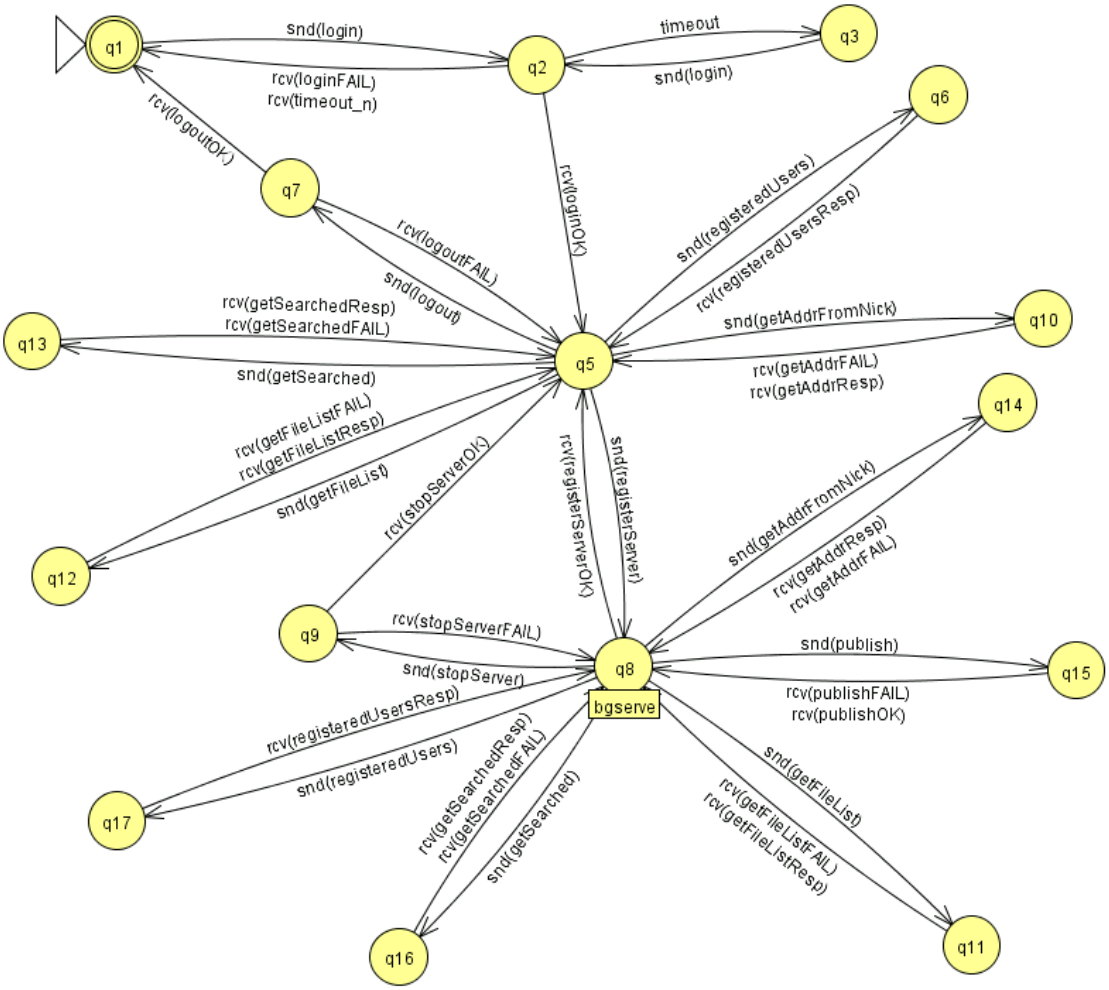
Autómatas de protocolo

Hemos ideado las siguientes restricciones:

- Un Cliente no podrá enviar ningún mensaje específico hasta que inicia conexión con el Directorio (login) y reciba una respuesta exitosa (loginOK).
- Un usuario no podrá actuar de Peer hasta que no envíe un mensaje pidiéndolo (fgserve o bgserve). Tampoco podrá publicar sus archivos hasta que no se convierta en Peer (utilizando el comando bgserve).
- El usuario no podrá irse (utilizando el comando quit) hasta que no cierre sesión. Tampoco podrá cerrar sesión si es un Peer (deberá parar el servidor en primer o segundo plano con fgstop y stopserver, respectivamente).

Autómata rol Cliente de Directorio

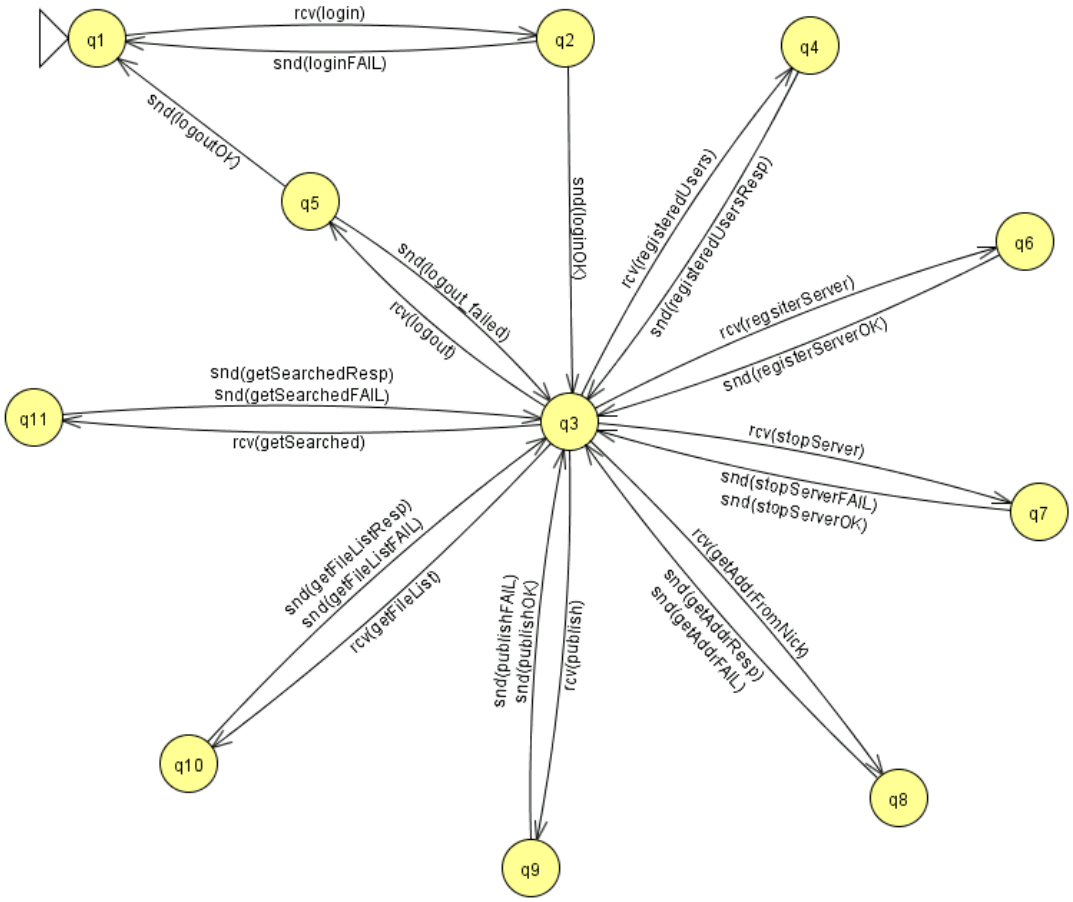
Desde el punto de vista del Cliente, este podrá hablar con el Directorio, además de convertirse en un servidor de ficheros. Tal como se plantea el proyecto, si el Cliente se convierte en Peer utilizando fgserve, no podrá seguir escribiendo comandos. Es cuando se convierte mediante bgserve cuando puede publicar sus archivos y mirar la lista de archivos del Directorio.



Autómata rol servidor de Directorio

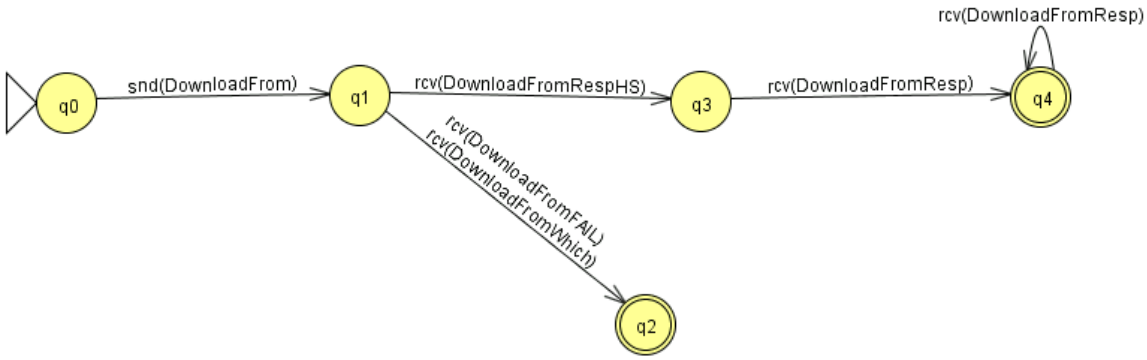
Desde el punto de vista del del Directorio, su único propósito es controlar el servicio que presta a los Clientes. Una vez el usuario inicia sesión, el Directorio se

quedará en el estado “q3” esperando los respectivos mensajes del Cliente. Una vez el cliente decida cerrar sesión, el Directorio volverá al estado “q1”.



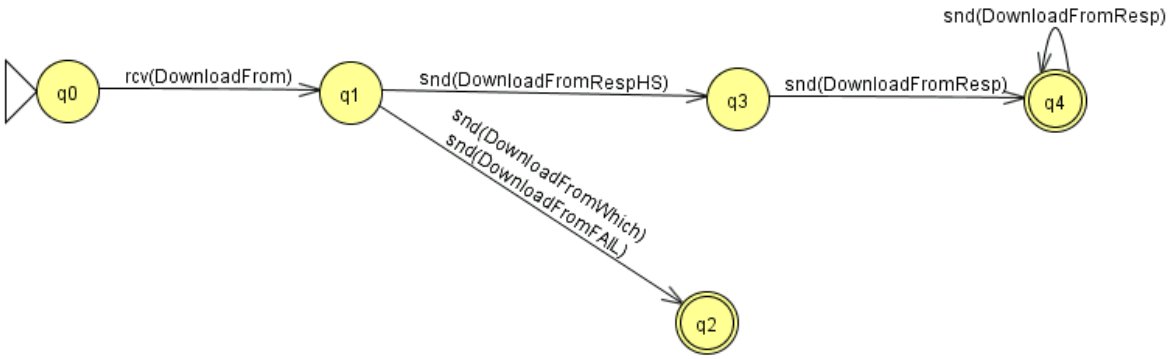
Autómata rol Cliente de ficheros

Aquí, el Cliente quiere obtener un fichero de alguno de los servidores. Se quedará en dando vueltas entre los estados “q1” y “q2” hasta que reciba el fichero.



Autómata rol servidor de ficheros

Una vez que se ha convertido el Cliente en un servidor, ya podrá enviar ficheros al resto de Clientes. El Cliente puede convertirse en Peer mediante los comandos fgserve y bgserve. Lo que harán ambos será igual, a excepción del comando final que escriben para acabar.


q0
rcv(DownloadFrom)
q1
snd(DownloadFromRespHS)
q3
snd(DownloadFromResp)
q4
snd(DownloadFromResp)
snd(DownloadFromWhich)
snd(DownloadFromFAIL)
q2

Ejemplo de intercambio de mensajes

Para poder demostraros como se envían las trazas utilizando Wireshark, hemos hecho login con el usuario pepe. Este usuario tendrá su puerto en 56061, como podéis comprobar. También hemos hecho login con el usuario juan y se le ha asignado el puerto 56062.

```
(nanoFiles@nf-shared) login localhost pepe
Created UDP socket at local addresss /[0:0:0:0:0:0:0:0]:56061
```

En la siguiente imagen se puede comprobar cómo viajan los paquetes. Para ello, utilizan el protocolo UDP. Como lo hemos comprobado en el mismo ordenador, la dirección IP es la de localhost. A la derecha aparecen los puertos de las dos entidades: 56061 del cliente y 6868 del servidor, cuando es pepe el que se logea, y 56062 cuando el que lo hace es juan.

2	1.860647	127.0.0.1	127.0.0.1	UDP	63	56061 → 6868 Len=31
3	1.879353	127.0.0.1	127.0.0.1	UDP	67	6868 → 56061 Len=35
4	8.254864	127.0.0.1	127.0.0.1	UDP	63	56062 → 6868 Len=31
5	8.256787	127.0.0.1	127.0.0.1	UDP	67	6868 → 56062 Len=35

Para finalizar, hemos grabado un vídeo donde se ve la ejecución del programa mediante los ficheros JAR ejecutable. Hemos puesto en práctica todos los comandos implementados, incluso los de las mejoras.

Enlace vídeo: https://youtu.be/_1v3b480PkM

Además, en el vídeo se nos olvida probar el comando downloadfrom sin estar logeados, así que os dejamos una imagen para que podáis comprobar como no se nos permite. También se nos olvidó mostrar como no podemos hacer logout si somos peer.

```
(nanoFiles@nf-shared) downloadfrom localhost:4615 4e90f5f963d4f9 rgsog.png
*You cannot run this command because you are not logged into the directory
```

```
(nanoFiles@nf-shared) logout
* You cannot log out because you are a peer
```

Conclusiones

Todo el mundo gira sobre las redes de comunicaciones. Hoy en día necesitas un móvil o acceso a Internet desde un dispositivo para poder hablar con alguien o mandarle la última foto que os echasteis juntos. Cuando usamos una aplicación para pedirle ayuda a nuestros padres, lo estamos haciendo a través de un modo no físico. ¿Somos realmente conscientes de cómo se produce esa comunicación internamente?

Gracias a lo nuevo aprendido en las clases de teoría, hemos podido saber de primera mano el funcionamiento de las redes de comunicaciones. Los protocolos UDP (User Datagram Protocol) y TCP (Transmission Control Protocol), tan básicos para el intercambio de datos entre dos hosts, son implementados de primera mano. Además, somos capaces de interiorizar la diferencia entre los mensajes ASCII (American Standard Code for Information Interchange) y los binarios. Debo puntualizar, la compresión interna de la estructura de estos últimos nos resultaba un tanto confuso al principio, pero una vez le dedicamos bastante más tiempo, fuimos capaces de entenderlo a la perfección.

En conclusión, nos quedamos satisfechos de todo lo aprendido durante la realización de la práctica. Podemos asegurar con firmeza, que estamos ante uno de los proyectos más importantes de la carrera, pese a encontrarse en su segundo curso. Resulta un tanto desafiante, aunque qué no lo es en nuestra carrera. La curva de aprendizaje que presenta es de la más duras. Sin embargo, con el devenir de las semanas, acudiendo a clase y escuchando con atención al profesor, uno puede mentalizarse del funcionamiento interno de la práctica.

El lenguaje Java sigue siendo, hoy en día, muy utilizado (en todos los ámbitos). Es por eso por lo que debemos aprender muy concienzudamente todos los aspectos que desentraña. Creemos que esta asignatura es la indicada para dar ese pequeño salto, y pasar de ser simples programadores a ingenieros.