Documento de diseño del programa NanoFiles

Participantes: Juan Jesús Ortiz García, Gonzalo Vicente Pérez.
Subgrupo 2.3

1. Introducción.

En este documento se especifica el diseño del protocolo de comunicación entre peer y directorio, y el de comunicación entre peers, junto con sus autómatas.

Además se incluye un listado con las mejoras implementadas, así como una breve descripción de su implementación.

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

1. Solicitud de login

Mensaje: Login

Campos: operation: login\n
nickname: <nickname>\n
\n

Operacion: Solicitud de login.

Emisor: Cualquier peer.

Receptor: Servidor de directorio.

Acciones: a) Se comprueba si <nickname> está en el diccionario "nicks".
b) Si no está, se genera la clave de sesión y se actualizan los diccionarios "nicks" y "sessionKeys". Después, se envía al peer un mensake de "loginOk", en el que se incluye la clave de sesión.
c) Si ya está, se envía un mensaje de "loginFail".

2. Login satisfactorio

Mensaje: LoginOk

Campos: operation: loginOk\n
sessionKey: <sessionkey>\n
\n

Operación: Respuesta de "login satisfactorio".

Emisor: Servidor de directorio.

Receptor: Cualquier peer.

Acciones: a) Se extrae la clave de sesión y se guarda en la variable de instancia de la clase "DirectoryConnector".
b) El método que DirectoryConnector utiliza para el login debe devolver "true".

3. Login repetido

Mensaje: LoginFail

Campos: operation: loginFail\n
\n

Operación: Respuesta de "login repetido".

Emisor: Servidor de directorio.

Receptor: Cualquier peer.

Acciones: a) El método que DirectoryConnector utiliza para el login debe devolver "false".

4. Cierre de sesión

Mensaje: logout

Campos: operation: logout\n
sessionkey: <sessionkey>\n
\n

Operación: Respuesta de "cierre de sesión".

Emisor: Cualquier peer.

Receptor: Servidor de directorio.

Acciones: a) Se extrae la clave de sesión y se comprueba si dicha clave de sesión está en el diccionario "sessionKeys".
b) Si la clave de sesión está en "sessionKeys", dicha clave se da de baja en el diccionario y se envía un mensaje de "logoutOk". También hay que dar de baja el usuario en el diccionario de nicks.
c) Si la clave de sesión no está en "sessionKeys", se envía un mensaje de "logoutFail".

5. Logout satisfactorio

Mensaje: LogoutOk

Campos: operation: logoutOk\n
\n

Operación: Respuesta de "logout satisfactorio".

Emisor: Servidor de directorio.

Receptor: Cualquier peer.

Acciones: a) La variable que el DirectoryConnector del peer emplea para guardar la clave de sesión debe ser inicializada a "null".
b) El método que DirectoryConnector utiliza para el logout debe devolver "true".

6. Logout fallido

Mensaje: LogoutFail

Campos: operation: logoutFail\n
\n

Operación: Respuesta de "logout repetido".

Emisor: Servidor de directorio.

Receptor: Cualquier peer.

Acciones: a) El método que DirectoryConnector utiliza para el logout debe devolver "false".

7. Listado de usuarios

Mensaje: userlist

Campos: operation: userlist\n
\n

Operación: Respuesta de "listado de usuarios".

Emisor: Cualquier peer.

Receptor: Servidor de directorio.

Acciones: a) Se comprueba si el diccionario "nicks" no está vacío.
b) Si no lo está, se extraen todas las claves del diccionario de usuarios y del de servidores y se devuelven.

c) Si está vacío, se envía un mensaje de "userlistFail".

8. Listado satisfactorio

Mensaje: UserlistOk

Campos: operation: userlistOk\n
users: user1,user2,user3,\n
servers: user1,user2,user3,\n
\n

Operación: Respuesta de "userlist satisfactorio".

Emisor: Servidor de directorio.

Receptor: Cualquier peer.

Acciones: a) El método que DirectoryConnector utiliza para el userlist debe devolver "true".

9. Userlist fallido

Mensaje: UserlistFail

Campos: operation: userListFail\n
\n

Operación: Respuesta de "userlist fallido".

Emisor: Servidor de directorio.

Receptor: Cualquier peer.

Acciones: a) El método que DirectoryConnector utiliza para el userlist debe devolver "false".

10. Registro de servidor

Mensaje: registerServer

Campos: operation: registerServer\n
sessionkey: <sessionkey>\n
port: <port>\n
\n

Operación: Petición de "registro de servidor".

Emisor: Cualquier peer.

Receptor: Servidor de directorio.

Acciones: a) Se comprueba si la sessionkey está en el diccionario, y se registra el servidor en caso de que sí lo esté.

11. Registro satisfactorio

Mensaje: registerServerOk

Campos: operation: registerServerOk\n
\n

Operación: Respuesta de "registro satisfactorio".

Emisor: Servidor de directorio.

Receptor: Cualquier peer.

Acciones: a) El método que DirectoryConnector utiliza para el registro debe devolver "true".

12. Registro fallido

Mensaje: registerServerFail

Campos: operation: registerServerFail\n
\n

Operación: Respuesta de "registro fallido".

Emisor: Servidor de directorio.

Receptor: Cualquier peer.

Acciones: a) El método que DirectoryConnector utiliza para el userlist debe devolver "false".

13. Borrado de servidor

Mensaje: unregisterServer

Campos: operation: unregisterServer\n
sessionkey: <sessionkey>\n
\n

Operación: Petición de "borrado de servidor".

Emisor: Cualquier peer.

Receptor: Servidor de directorio.

Acciones: a) Se comprueba si la sessionkey está en el diccionario, y se pone el puerto del servidor a -1 caso de que sí lo esté.

14. Borrado satisfactorio

Mensaje: unregisterServerOk

Campos: operation: registerServerOk\n
\n

Operación: Respuesta de "Borrado satisfactorio".

Emisor: Servidor de directorio.

Receptor: Cualquier peer.

Acciones: a) El método que DirectoryConnector utiliza para el registro debe devolver "true".

15. Borrado fallido

Mensaje: unregisterServerFail

Campos: operation: unregisterServerFail\n
\n

Operación: Respuesta de "borrado fallido".

Emisor: Servidor de directorio.

Receptor: Cualquier peer.

Acciones: a) El método que DirectoryConnector utiliza para el userlist debe devolver "false".

16. Registro de servidor

Mensaje: registerIP

Campos: operation: registerIP\n
sessionkey: <sessionkey>\n
ip: <ip>\n
\n

Operación: Petición de "registro de ip de servidor".

Emisor: Cualquier peer.

Receptor: Servidor de directorio.

Acciones: a) Se comprueba si la sessionkey está en el diccionario, y se registra el servidor en caso de que sí lo esté.

17. Registro satisfactorio

Mensaje: registerIPOk

Campos: operation: registerIPOk\n
\n

Operación: Respuesta de "registro satisfactorio".

Emisor: Servidor de directorio.

Receptor: Cualquier peer.

Acciones: a) El método que DirectoryConnector utiliza para el registro debe devolver "true".

18. Registro fallido

Mensaje: registerIPFail

Campos: operation: registerIPFail\n
\n

Operación: Respuesta de "registro fallido".

Emisor: Servidor de directorio.

Receptor: Cualquier peer.

Acciones: a) El método que DirectoryConnector utiliza para el userlist debe devolver "false".

19. Petición de dirección de servidor

Mensaje: requestAddress

Campos: operation: requestAddress\n
nickname: <nickname>\n
\n

Operación: Petición de "dirección de servidor".

Emisor: Cualquier peer.

Receptor: Servidor de directorio.

Acciones: a) Se comprueba si la sessionkey está en el diccionario, y se extrae la ip y el puerto del

nickname.

20. Petición satisfactoria

Mensaje: requestAddressOK

Campos: operation: requestAddressOk\n
ip: <ip>\n
port:<port>\n
\n

Operación: Respuesta de "petición satisfactoria".

Emisor: Servidor de directorio.

Receptor: Cualquier peer.

Acciones: a) El método que DirectoryConnector utiliza para el registro debe devolver "true".

21. Petición fallida

Mensaje: requestAddressFail

Campos: operation: requestAddressFail\n
\n

Operación: Respuesta de "petición fallida".

Emisor: Servidor de directorio.

Receptor: Cualquier peer.

Acciones: a) El método que DirectoryConnector utiliza para el userlist debe devolver "false".

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

Diseño de mensaje de solicitud de descarga de fichero:

Mensaje: Download

Campos:

OpCode	HashLength	FileHash
1 byte	1 byte	n bytes

Sentido: Peer cliente -> Peer servidor

Descripción: Pedir la descarga de un fichero a partir de su hash.

Ejemplo:

OpCode	HashLength	FileHash
1	4	8g6s

Diseño de mensaje de respuesta de descarga (ok):

Mensaje: DownloadOk

Campos:

OpCode	DataLength	DataBytes	HashLength	FileHash
1 byte	8 bytes	n bytes	1 byte	n bytes

Sentido: Peer servidor -> Peer cliente

Descripción: Comunicar que el fichero se ha encontrado y devolver sus datos junto con el hash original para poder verificar su integridad .

Ejemplo:

OpCode	DataLength	DataBytes	HashLength	FileHash
2	624	624 bytes	4	8g6s

Diseño de mensaje de respuesta de descarga (fail):

Mensaje: DownloadFail

Campos:

OpCode
1 byte

Sentido: Peer servidor -> Peer cliente

Descripción: Comunicar que el fichero no se ha encontrado.

Ejemplo:

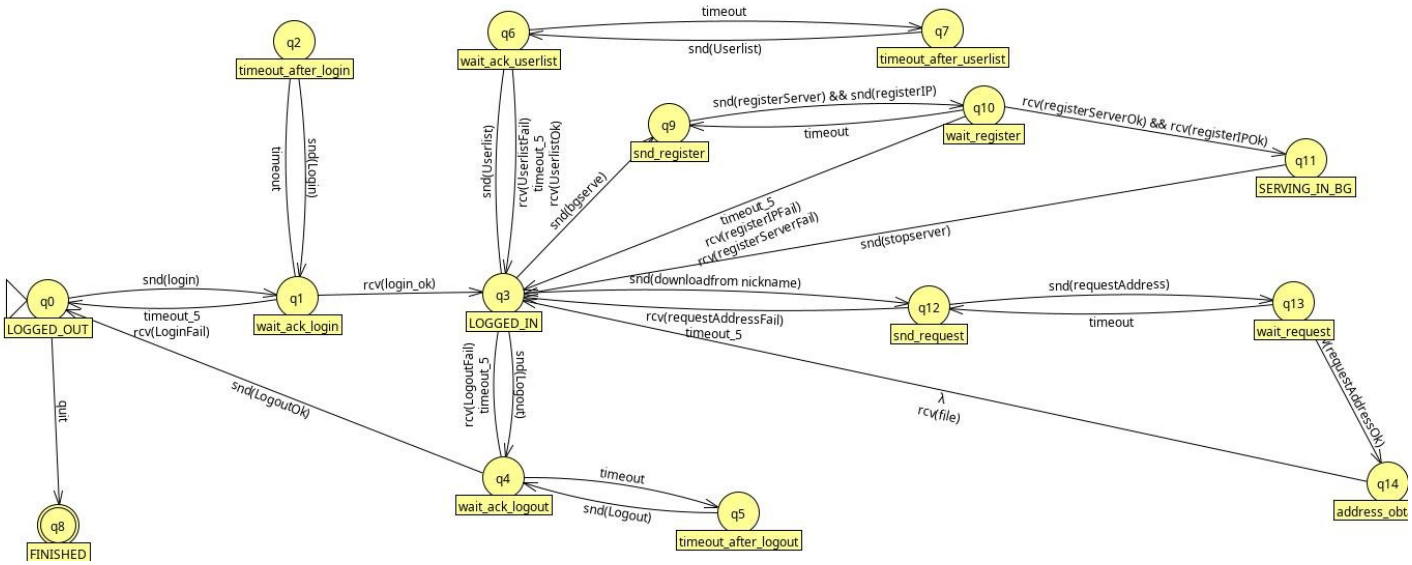
OpCode
3

4. Autómatas de protocolo

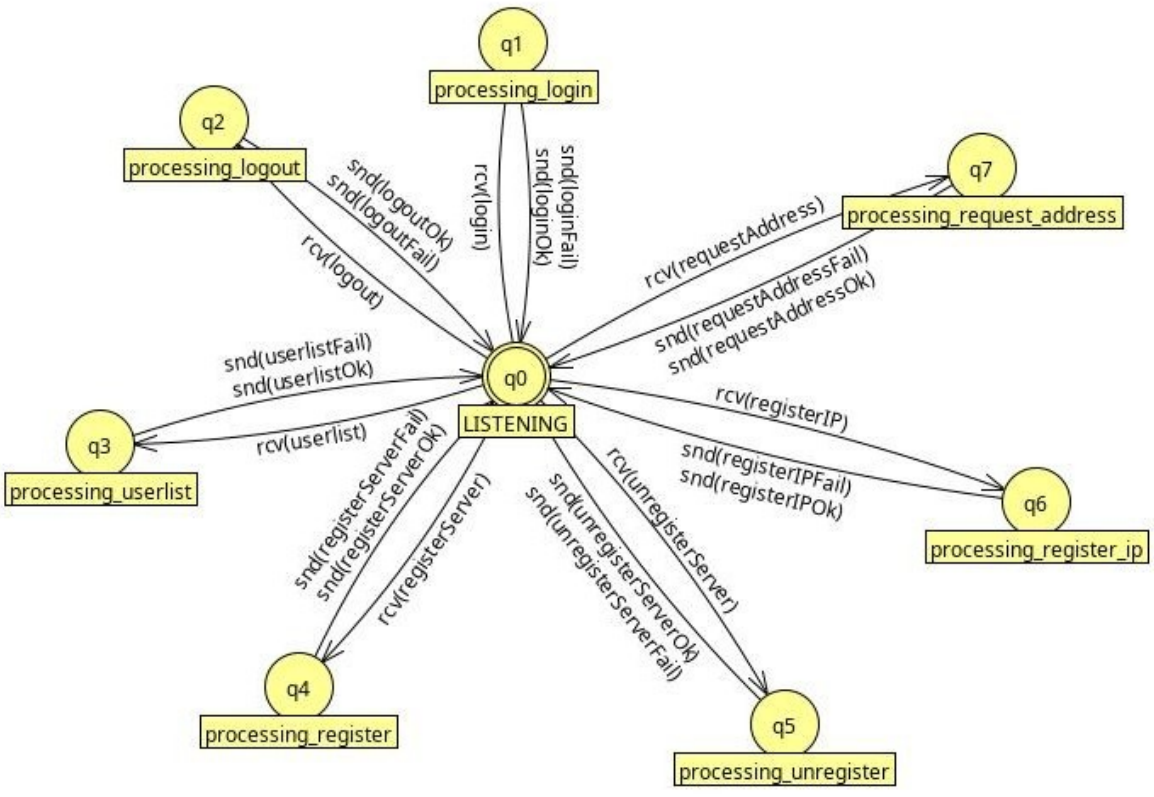
Con respecto a los autómatas, hemos considerado las siguientes restricciones:

- Un cliente del directorio no puede pedir la lista de usuarios,hacer logout o servir ficheros en segundo plano si no ha iniciado sesión previamente.
- Para parar un servidor en segundo plano, primero hay que estar sirviendo en segundo plano.
- Si ya se está sirviendo en segundo plano, no se puede volver a servir en segundo plano.
- Para salir o hacer login, se debe estar logged out.
- El servidor de directorio siempre estará escuchando, salvo que le llegue una petición, que atenderá y volverá a escuchar.

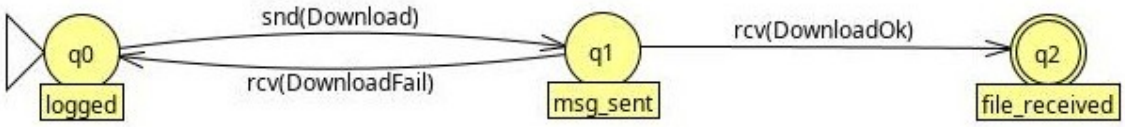
Autómata rol cliente de directorio



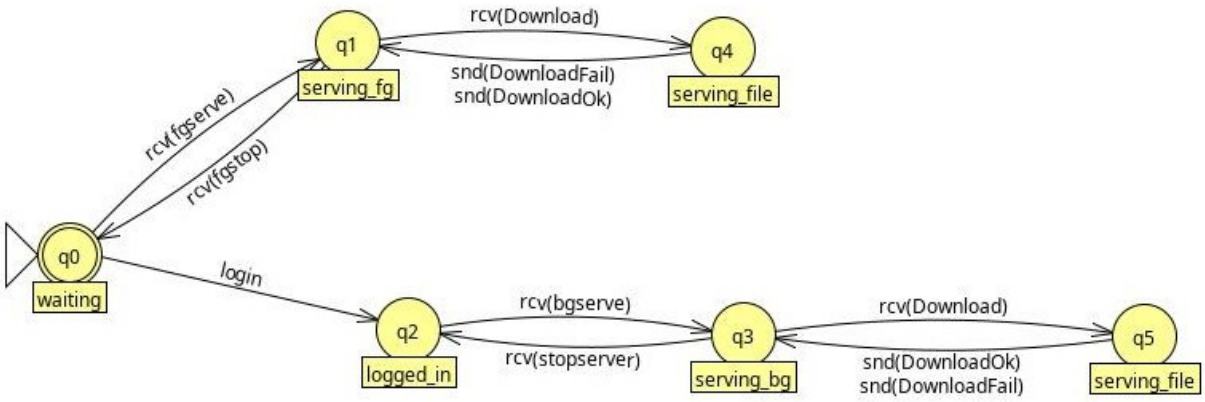
Autómata rol servidor de directorio



Autómata rol cliente de ficheros



Autómata rol servidor de ficheros



5. Ejemplo de intercambio de mensajes

Incluir en esta sección ejemplos de “conversaciones” ficticias (con valores inventados) haciendo uso de los mensajes definidos en las secciones anteriores y comentando cómo el autómata restringe qué mensaje(s) puede enviar recibir cada extremo de la comunicación en cada instante de la conversación (estado del autómata).

CLIENTE: inicia sesión
```
operation: login
nickname: alumno
```

DIRECTORIO: confirma inicio de sesión
```
operation:loginOk
sessionkey:666
```

CLIENTE: inicia sesión
```
operation: login
nickname: alumno
```

DIRECTORIO: deniega inicio de sesión (ya está registrado)
```
operation: loginFail
```

CLIENTE: sirve en segundo plano
```
operation:  registerServer
sessionkey: 666
port: 24424
operation:  registerIP
sessionkey: 666
ip: 192.168.1.34
```

DIRECTORIO: acepta
```
operation:  registerServerOk
operation:  registerIPOk
```

CLIENTE: pide la lista de usuarios
```
operation: userlist
```

DIRECTORIO: devuelve userlist
```
operation: userlistOk
users: juanje,gonzalo,
servers: juanje,
```

CLIENTE: Deja de servir en segundo plano.
```
Operation: unregisterServer
```

DIRECTORIO: acepta
```
operation: unregisterServerOk
```

Cliente: Quiere descargar un fichero por su nick
```
operation: requestAddress
nickname: alumno
```

Directorio: devuelve IP y puerto asociados al nick
```
operation: requestAddressOk
ip:192.168.1.34
port: 24424
```

CLIENTE: Cierra sesión
```
operation: logout
sessionkey: 666
```

DIRECTORIO: Confirma logout
```
operation: logoutOk
```

CLIENTE: Pide userlist
```
operation: userlist
```

DIRECTORIO: La deniega (no ha hecho login)
```
operation: userlistFail
```

6. Listado de mejoras implementadas

1. **Comando fgstop (0,5 puntos):**
-Fichero/s trabajado/s: NFServerSimple.java (Paquete tcp server).

Descripción de la implementación: Lo que hemos hecho ha sido utilizar dos hilos, uno servidor que se encarga de aceptar conexiones y usando el método NFServerComm.serveFilesToClient(socket) servir el fichero al cliente, y otro en el que se lee de teclado, y si se introduce el comando fgstop, se pone a true la variable que indica que hay que parar el servidor. Por tanto, el método run de la clase NFServerSimple se encarga de arrancar ambos hilos, posteriormente se queda en un bucle que no hace nada y solo termina cuando la variable stopServer sea true, en cuyo caso se cierra el socket y se vuelve al shell.
Un detalle importante es que al encerrar el servicio de ficheros en un hilo, la parada del servidor es inmediata, pues no hay que esperar a que se salga del accept (que es bloqueante), si no que simplemente se saldrá del bucle, pudiendo cerrarse el socket (lo que generará una excepción en el hilo servidor que se trata adecuadamente) y terminado la ejecución del servidor en primer plano.

2. **fgserve con puerto variable (0,5 puntos):**
-Fichero/s trabajado/s: NFServerSimple.java (Paquete tcp server).

En el constructor de la clase, se pone una variable que indique si el socket ha hecho bind a false, y se ejecuta un bucle en el que mientras que no se pueda hacer el bind, se capture la excepción del bind y se incremente en 1 el puerto. Cuando el bind se ejecute, se pondrá dicha variable a true y se saldrá del bucle.

3. **bgserve (1 punto):**
-Fichero/s trabajado/s: NFServer.java (Paquete tcp server).
-NFControllerLogicP2P.java (Paquete logic).

En el controller P2P, se implementa el método backgroundServeFiles(). Primero se crea una variable de clase de tipo NFServer. Ya dentro del método, se comprueba que sea nula para saber si ya hay un servidor ejecutándose. En caso de que no, se crea un objeto NFServer, y se arranca el servidor. Posteriormente se comprueba que el puerto sea válido, y se imprime la dirección en la que se ejecuta.
En NFServer, se sigue un esquema similar al de NFServerSimple. Se implementa su constructor, y su método run, y algunos métodos auxiliares de utilidad.

4. **bgserve con puerto efímero (0,5 puntos):**
-Fichero/s trabajado/s: NFServer.java (Paquete tcp server).

La lógica es la misma que en el fgserve, se usa una variable para saber si se ha hecho bind y mientras que no se haya hecho se prueba con una dirección con un puerto aleatorio entre 10000 y 50000.

5. **stopserver (0,5 puntos):**
-Fichero/s trabajado/s: NFServer.java (Paquete tcp server).
-NFControllerLogicP2P.java (Paquete logic).

En el controllerP2P se llama a un método de la clase NFServer para parar el servidor, y se pone a null la variable del servidor en segundo plano.
En la clase NFServer, al igual que en el fgserve, en el método run, se encierra el servicio de ficheros en un hilo, de tal forma que si se introduce el comando stopserver, se sale del bucle y se cierra el socket.

6. **bgserve multihilo (0,5 puntos):**
-Fichero/s trabajado/s: NFServer.java (Paquete tcp server).
-NFServerThread.java (Paquete tcp server).

En el método servidor() de NFServer, cuando se acepta una conexión, en vez de llamar al método NFServerComm.serveFilesToClient(socket), se crea un nuevo hilo NFServerThread, que se encargará de hacer la llamada al método serveFilesToClient.

7. **userlist con servidores (0,5 puntos):**
-Fichero/s trabajado/s (principalmente): DirectoryConnector.java (Paquete udp client).
-NFDirectoryServer.java (Paquete udp server).
-DirMessage.java (Paquete udp message).

Cuando se hace un bgserve, se llama a un método para notificar al directorio. Desde el NFController, se acaba ejecutando el método registerServerPort(int serverPort) del DirectoryConnector. Este método manda unn mensaje al directorio para guardar en el diccionario de servidores el puerto asociado a la sessionKey. También cuando se hace el stopserver, hay un método unregisterServerPort(int sessionKey) que notifica al directorio para que borre el puerto asociado a la sessionKey.

El NFDirectoryServer lo que hace es una vez que recibe el mensaje registerServer guarda en el diccionario de servidores el puerto asociado a la sessionKey. Cuando se quiere dar de baja, se asocia a la sessionKey el puerto -1, que no es válido.
En la clase DirMessage está la implementación de los mensajes de comunicación con el directorio (mensajes registerServer y unregisterServer).
También se modifican los métodos asociados a la obtención de la userlist para obtener los servidores además de los usuarios.

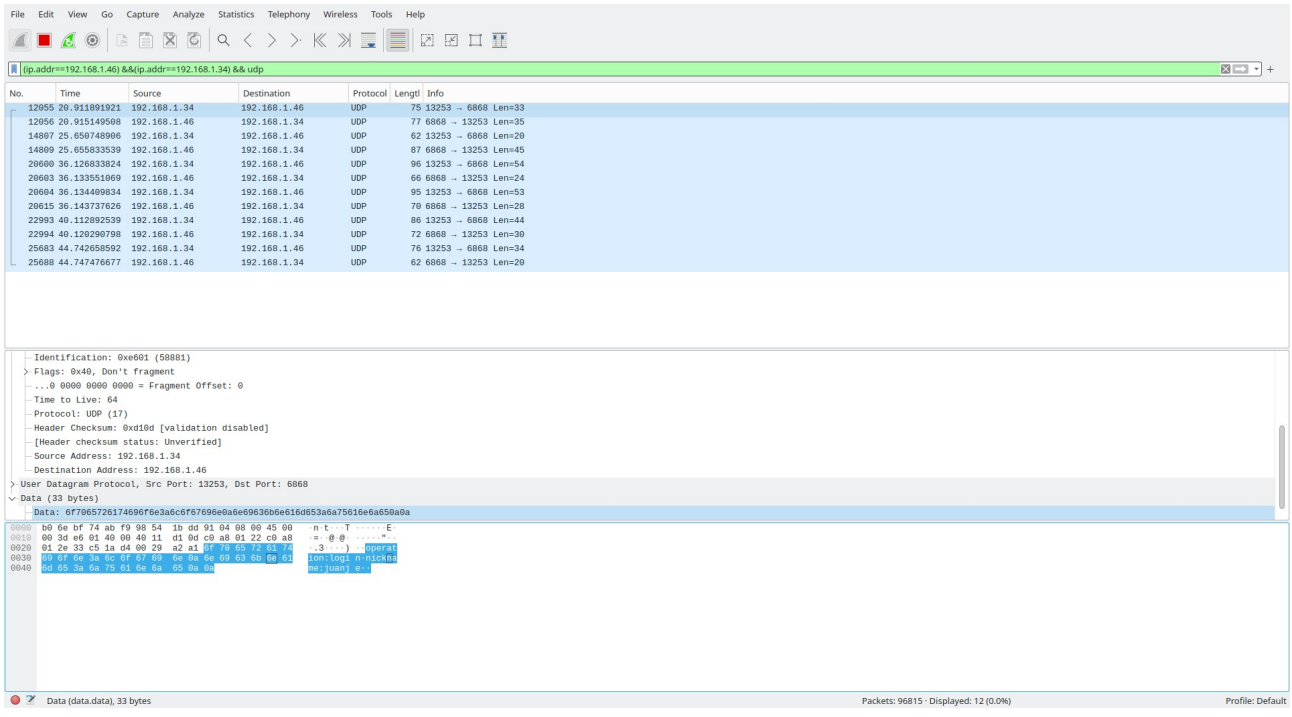
8. **downloadfrom por nickname (1 punto):**
-Fichero/s trabajado/s (principalmente): DirectoryConnector.java (Paquete udp client).
-NFDirectoryServer.java (Paquete udp server).
-DirMessage.java (Paquete udp message).

Cuando se hace un bgserve, se ejecutan los métodos necesarios en el NFController para que el directorio registre IP y puerto asociados al nick. El registro del puerto ya ha sido descrito en la mejora anterior, y el registro de la IP sigue una lógica prácticamente idéntica. Una vez que el directorio conoce IP y puerto asociados al nick, cuando en el NFController se llama al método getServerAddress antes de ejecutar el downloadFile, se acaba ejecutando el método lookupServerAddrByUsername(String nick) del DirectoryConnector, que recuperará del directorio la ip y el puerto asociadas al nick, y los pasará como una InetSocketAddress al método downloadFile.

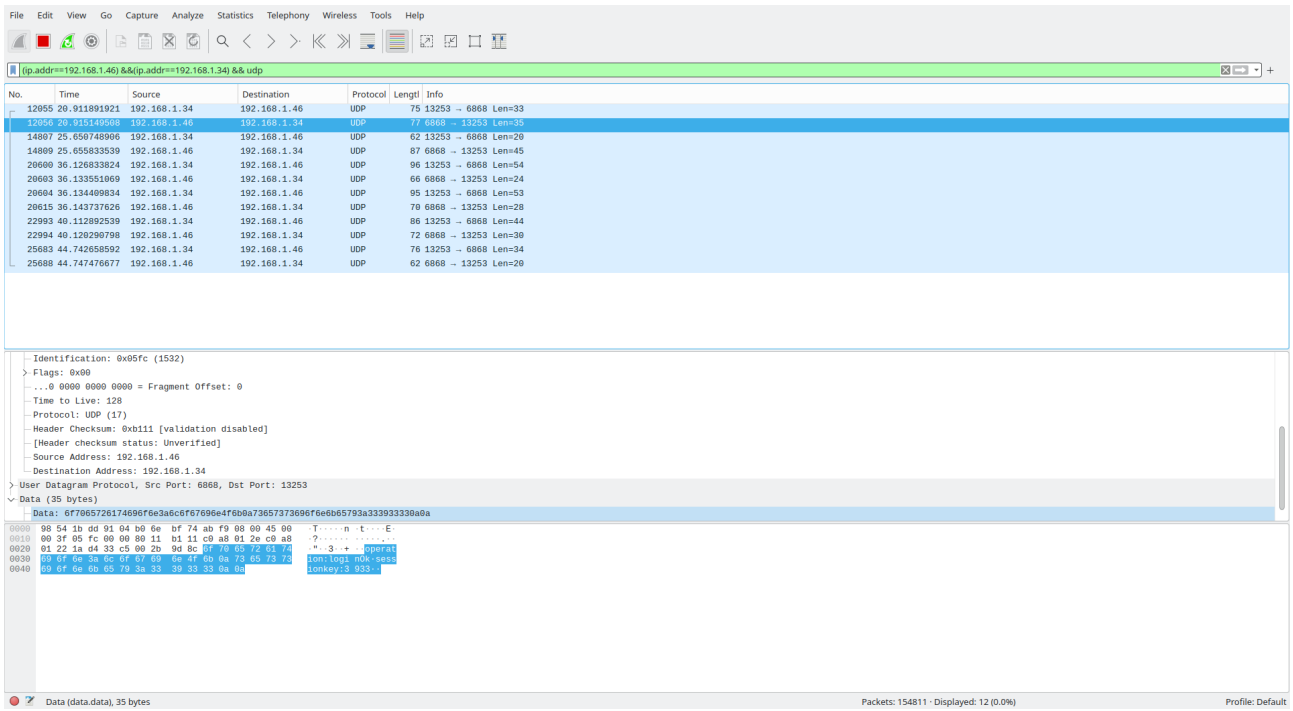
7. Capturas de pantalla de Wireshark

A continuación se muestran una serie de capturas de pantalla, con los mensajes resultado de los siguientes comandos:
login 192.168.1.46 juanje
userlist
bgserve
stopserver
logout

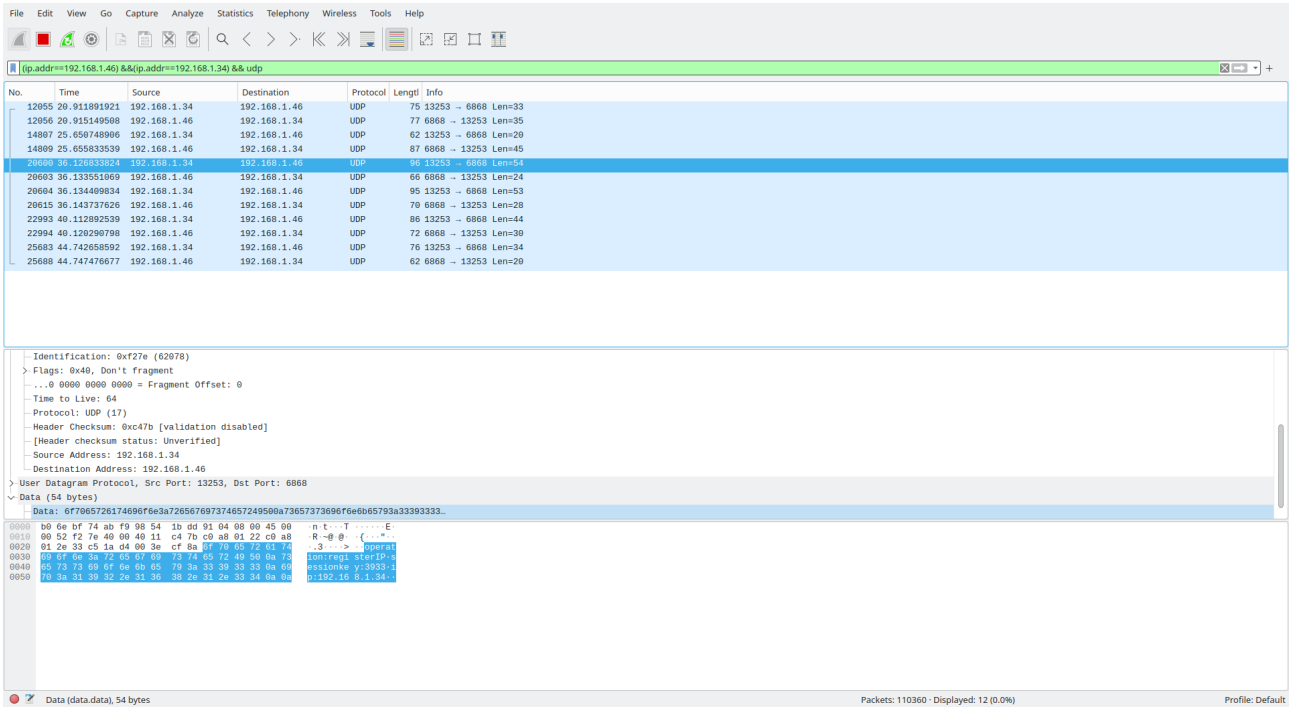
<u>Datos del mensaje de login:</u>



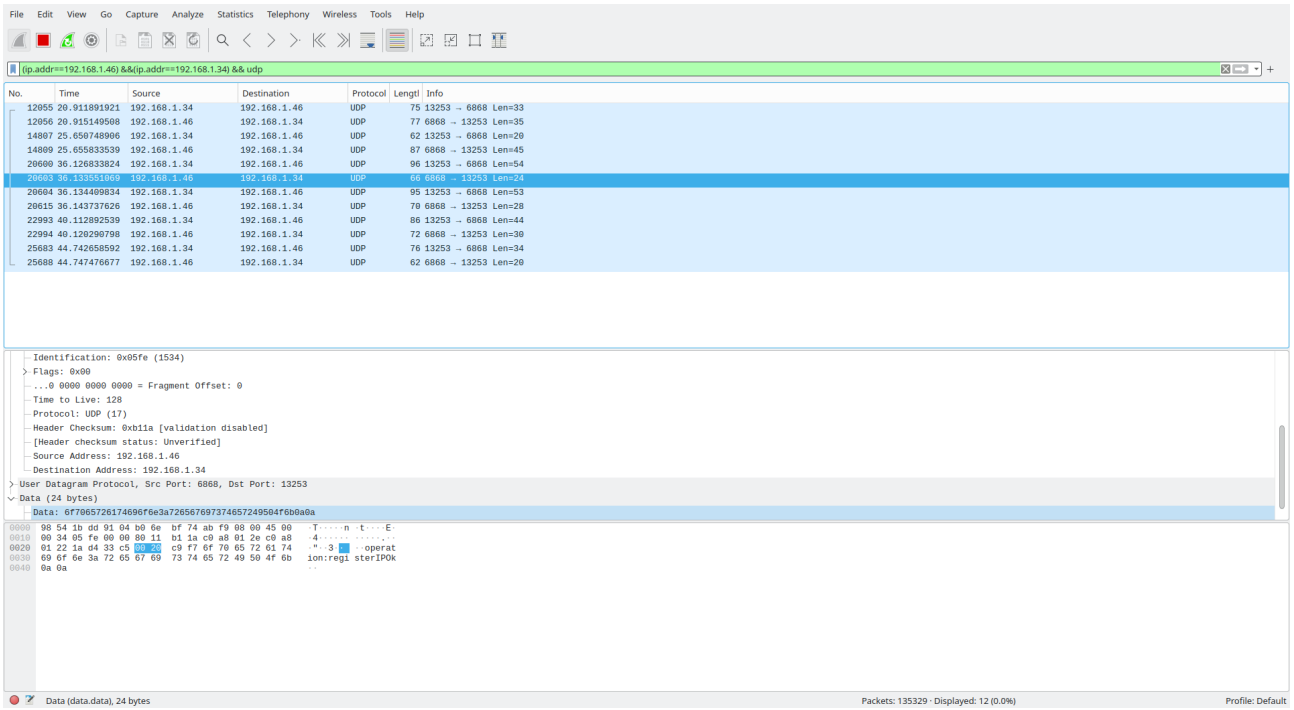
Datos de la respuesta del directorio al login:



Datos del registro de la ip en el directorio:



Datos de la respuesta del directorio al registro de ip:



8. Vídeo

Vídeo con un ejemplo de uso: usoNF.mp4

9. Conclusiones

Ha sido el proyecto más extenso al que nos hemos enfrentado hasta ahora. Desde nuestro punto de vista, ha resultado crucial seguir las sesiones de laboratorio y llevarlo más o menos al día, pues la amplia extensión del mismo hacía prácticamente imposible el reengancharse al mismo si nos quedabamos atrás.

Sin embargo, aunque nos ha planteado bastantes retos, no creemos que haya sido un proyecto altamente complejo. Tanto los TODO's, como los boletines, junto con las explicaciones en laboratorio, guiaban bastante bien a la hora de implementar el proyecto, por lo que no hemos sentido que estábamos a nuestra merced ante el proyecto.

Hemos sentido dos momentos críticos: el primero al inicio del proyecto, cuando no estábamos familiarizados con el código, las clases, los paquetes… El segundo momento crítico ha sido a la hora de la entrega, pues hemos tenido que hacer muchas pruebas y buscar el más mínimo fallo en el programa para tratar de depurarlo al máximo.

En general creemos que es un proyecto que cumple su objetivo de dar a conocer la programación utilizando elementos de redes de comunicaciones, creando una aplicación que además tiene cierta utilidad.