## 0. Portada

Alumno: Sergio Sánchez López
DNI: 49245005G
Subgrupo: 3.2

**INDICE**

## 1. Introducción.

En este documento se especifica el diseño de los protocolos de comunicación, entre un Peer (NanoFiles) y el servidor (Directory) y entre dos Peers. Así como el formato y diseño de los diferentes mensajes que intercambian ambas partes para poder llevar a cabo las tareas del enunciado.
Así, también se hará una breve descripción de las mejoras implementadas y sus detalles en la implementación.

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

La manera en las que se comunicaran los NanoFiles y Directorio será mediante mensajes textuales, que como bien se pide en su respectivo boletín se usara el formato “field:value”.
Para poder descodificar el mensaje y extraer los diferentes campos, así como para saber ante que tipo de mensaje estamos, tendremos el campo “operation”, cuyo valor indicara el formato y tipo de los siguientes campos del mensaje, si es que los tuviera.

**Tipos y descripción de los mensajes**

Mensaje: D**irMessageInvalid**
Sentido de la comunicación: Cliente ← Directorio
Descripción: Este mensaje sirve para indicar que se ha intentado hacer una operación invalida.
Ejemplo:

```
operation: invalid_operation\n
```

Mensaje: D**irMessagePing**
Sentido de la comunicación: Cliente → Directorio
Descripción: Con este mensaje el cliente nanoFiles trata de logearse, enviando su protocolId para ver si coincide con el del directorio.
Ejemplo:

```
operation: ping\n
protocolId = (DNI)
```

Mensaje: D**irMessagePingOk**
Sentido de la comunicación: Cliente ← Directorio
Descripción: El directorio comprueba que el protocolId coincide y da OK al nanoFiles, el cual queda ya logeado para poder enviar cualquier otra petición.
Ejemplo:

```
operation: ping_ok\n
```

- Mensaje: D**irMessageRequestFileList**
  Sentido de la comunicación: Cliente → Directorio
  Descripción: Mensaje usado por el nanoFiles para pedirle al directorio una lista con los ficheros disponibles para descargar.
  Ejemplo:

```
operation: request_file_list\n
```

```
Mensaje: DirMessageRequestFileListOk
```
  Sentido de la comunicación: Cliente ← Directorio
  Descripción: Respuesta del direcotrio al requestFileList, en el cual informa que todo ha ido bien y devuelve la lista solicitada
  Ejemplo:

```
operation: request_file_list_ok\n
files : <array con datos de los ficheros disponibles>
```

- Mensaje: D**irMessageRequestFileListFail**
  Sentido de la comunicación: Cliente ← Directorio
  Descripción: Respuesta del direcotrio al requestFileList, en el cual informa que ha habido un error y no se ha podido pasar la lista de ficheros
  Ejemplo:

```
operation: request_file_list_fail\n
```

- Mensaje: D**irMessagePublishFiles**
  Sentido de la comunicación: Cliente → Directorio
  Descripción: Mensaje que usar nanoFiles para solicitar que se publiquen los ficheros que manda en el campo files en el directorio.
  Ejemplo:

```
operation: publish_files\n
files: <array con los datos de los ficheros a publicar>
port: <puerto por el que se estan sirviendo los ficheros>
```

- Mensaje: D**irMessagePublishFilesOk**
  Sentido de la comunicación: Cliente ← Directorio
  Descripción: Respuesta del direcotrio al publish_files, en el cual informa que todo ha ido bien y que sus ficheros han sido publicados con éxito
  Ejemplo:

```
operation: publish_files_ok\n
```

Mensaje: **DirMessagePublishFilesFail**
Sentido de la comunicación: Cliente ← Directorio
Descripción: Respuesta del direcotrio al publish_files, en el cual informa que no se han podido publicar los ficheros

```
operation: publish_files_fail\n
```

- Mensaje: **DirMessageRequestServersList**
  Sentido de la comunicación: Cliente → Directorio
  Descripción: Mensaje que usa nanoFiles para pedir al directorio una lista de los servidores disponibles que están sirviendo el fichero que contiene la string que se pasa en el campo filenameSubstring.
  Ejemplo:

```
operation: request_servers_list\n
filenameSubstring:<subcadena del nombre del fichero deseado>
```

- Mensaje: **DirMessageRequestServersListOk**
  Sentido de la comunicación: Cliente ← Directorio
  Descripción: Respuesta del directorio para indicar que se ha encontrado el fichero deseado y ademas de la lista de los host que lo estan sirviendo
  Ejemplo:

```
operation: request_servers_list_ok\n
serversList:<array de host que comparten el fichero>
```

- Mensaje: **DirMessageRequestServersListFail**
  Sentido de la comunicación: Cliente ← Directorio
  Descripción: Respuesta del directorio que indica que o no se ha encontrado el fichero deseado o que ha ocurrido otro error
  Ejemplo:

```
operation: request_servers_list_fail\n
```

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

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

**Tipos y descripción de los mensajes**

Mensaje: **FileNotFound (opcode = 1)**
Sentido de la comunicación: Servidor de ficheros → Cliente
Descripción: Este mensaje lo envía el par servidor de ficheros al par cliente (receptor) de

fichero para indicar que no es posible encontrar el fichero con la información proporcionada en el mensaje de petición de descarga.

Ejemplo:

| **Opcode (1 byte)** |
|---|
| **1** |

Mensaje: AmgiguousName **(opcode = 2)**
Sentido de la comunicación: Servidor de ficheros → Cliente
Descripción: Este mensaje lo envía el par servidor de ficheros al par cliente (receptor) de fichero para indicar que con el substring para identificar el fichero no es suficiente ya que es ambiguo.

| **Opcode (1 byte)** |
|---|
| **2** |

Mensaje: CorruptDownload **(opcode = 9)**
Sentido de la comunicación: Servidor de ficheros ← Cliente
Descripción: Con este mensaje el cliente de ficheros indica al servidor de ficheros que la descarga ha sido corrupta, que se han perdido bytes o alguno ha sido modificado y por lo tanto el hash del fichero no es igual que el servidor proporciono.

| **Opcode (1 byte)** |
|---|
| **9** |

Mensaje: UploadFail **(opcode = 12)**
Sentido de la comunicación: Servidor de ficheros → Cliente
Descripción: Cuando el cliente intenta subir un fichero al servidor de ficheros y este rechaza la subida manda este mensaje al cliente para indicar que no puede llevar a cabo la subida.

| **Opcode (1 byte)** |
|---|
| **12** |

Mensaje: UploadOk **(opcode = 11)**
Sentido de la comunicación: Servidor de ficheros → Cliente
Descripción: Cuando el cliente intenta subir un fichero al servidor de ficheros y este acepta la subida manda este mensaje al cliente para indicar que puede llevar a cabo la subida.

| **Opcode (1 byte)** |
|---|
| **12** |

Mensaje: DownloadFile **(opcode = 3)**
Sentido de la comunicación: Servidor de ficheros ← Cliente
Descripción: Mensaje con el que el cliente informa al servidor el fichero que quiere descargar, mandando un substring con el que identificar el fichero.

| **Opcode (1 byte)** | **Substring Lenght (4 bytes)** | **Substring n bytes** |
|---|---|---|
| **3** | **n** | **Ascci bytes** |

Mensaje: DownloadAprove **(opcode = 4)**
Sentido de la comunicación: Servidor de ficheros → Cliente
Descripción: Cuando el cliente solicita descargar x fichero, el servidor manda este mensaje para indicar que puede empezar las descarga, además manda el código hash del fichero para una vez completada la descarga comprobar que el fichero no ha tenido ninguna mutación.

| **Opcode (1 byte)** | **FileSize (8 bytes)** | **Hash Code (40 bytes)** |
|---|---|---|
| **4** | **size** | **hash_code** |

Mensaje: GetChunk **(opcode = 5)**
Sentido de la comunicación: Servidor de ficheros ← Cliente
Descripción: Mensaje usado por el cliente para indicar al servidor que chunk debe de mandar.

| **Opcode (1 byte)** | **FileOffset (8 bytes)** | **ChunkSize (4 bytes)** |
|---|---|---|
| **5** | **offset** | **size** |

Mensaje: SendChunk **(opcode = 6)**
Sentido de la comunicación: Servidor de ficheros → Cliente
Descripción: Este es el mensaje que usan los servidores de ficheros para mandar la data al cliente.

| **Opcode (1 byte)** | **ChunkSize (4 bytes)** | **Data (n bytes)** |
|---|---|---|
| **6** | **n** | **data** |

Mensaje: Upload **(opcode = 10)**
Sentido de la comunicación: Servidor de ficheros → Cliente
Descripción: Este es el mensaje que usan los servidores de ficheros para mandar la data al cliente.

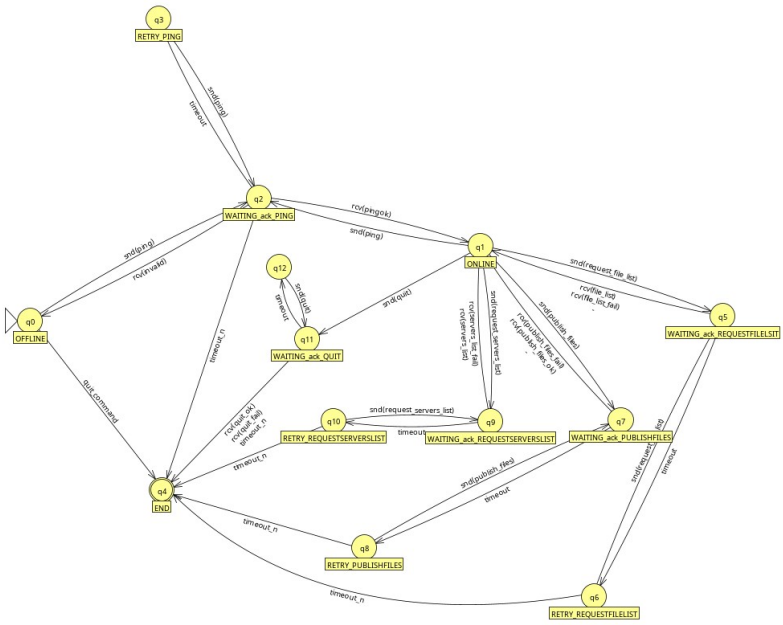
| **Opcode (1 byte)** | **Substring Lenght (4 bytes)** | **Substring n bytes** |
|---|---|---|
| **10** | **n** | **Ascci bytes** |

## 4. Autómatas de protocolo

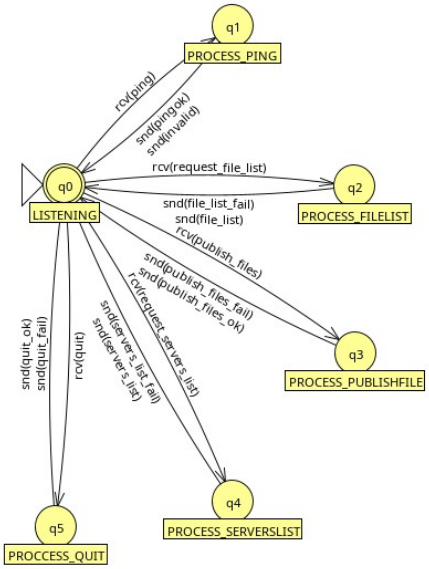
Con respecto a los autómatas, hemos considerado las siguientes restricciones:

- En cliente de directorio no puede ejecutar ninguna consulta la Directory sin antes haber echo un ping y recibir un ping_ok.
- Tras reeintentar n veces el reenvío de un mensaje y que salte el tiemout he decidido que el proceso de nanofiles termina debido a que el servidor no esta en servicio.
- Si ya has hecho un ping y recivido ping_ok cuando se lanza quit este se procesa para actualizar la lista de ficheros disponibles.
- También he concebido los estados necesarios para las mejoras como upload o el mantenimiento actualizado de la lista de ficheros.

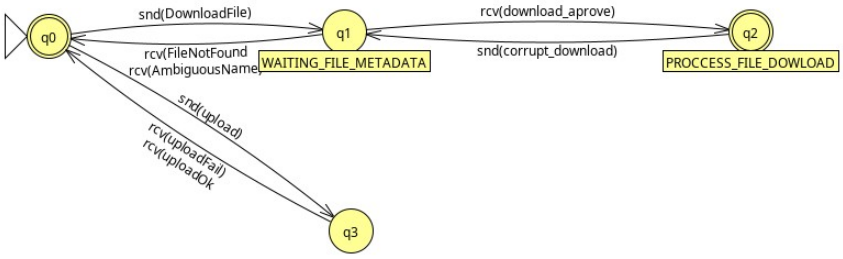
Autómata rol cliente de directorio



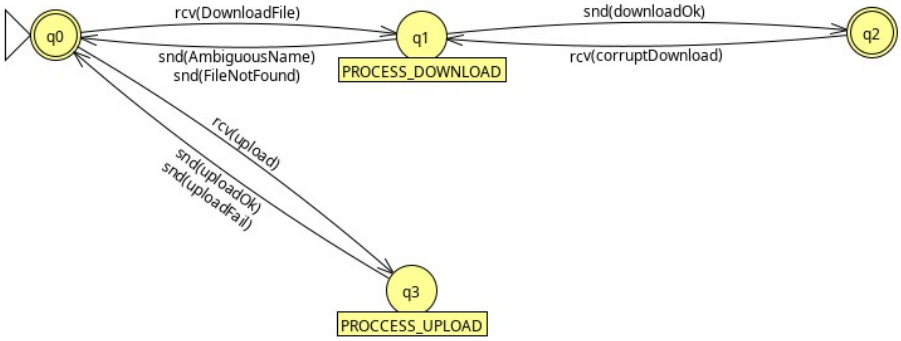

Autómata rol servidor de directorio




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: ping
```
operation: ping
protocol: 123456789Z
```

DIRECTORIO: procesa el ping → pingOK
operation: ping_ok

---

CLIENTE: request file list
```
operation: request_file_list
```

DIRECTORIO: procesa el request file list
```
operation: request_file_list_ok\n
files :
{[logo.png,234,alhk3h9a88ad83dad],
[passwd.txt,567,3a8ad838adfa8ad8ad8ad],
[um.html,56,8a88q3adfadfcnnc8]}

-----------------------------------
```

CLIENTE: solicita descarga ficheros

| Opcode (1 byte) | Substring Lenght (4 bytes) | Substring 8 bytes |
|---|---|---|
| 3 | 4 | Ascci bytes(test) |

Servidor: Nombre ambiguo

| Opcode (1 byte) |
|---|
| 2 |

## 6. Mejoras Implementadas.

En cuento a las mejoras que he implementado son las siguientes:

- serve con puerto efimero
-quit que actualiza la lista de ficheros y servidores
-download paralelo

- Para la implementación del serve con puerto efímero hay que modificar un poco los mensajes que intercambian directorio y nanoFiles a la hora de hacer un serve, ahora en mensaje publish_files también se tiene que informar del puerto en el que el peer servidor esta escuchando las conexiones.

Además, para que el socket que abre el peer servidor escoja un puerto efímero basta con inicializar la InetSocketAddress del servidor con el parametro 0, lo que hace que el sistema elija un puerto libre: `InetSocketAddress socketAddress = new InetSocketAddress(0);`
Ahora cuando el peer cliente vaya a descargar, en el mensaje que recibe del directory tendrá que hacer un getPort sobre el mensaje recibido para poder obtener el puerto y ya poder crear la lista de InetSocketAddress que están sirviendo el fichero que solicito.

- Para la mejora del quit, hay que crear más tipos de mensaje (DirMessage) de intercambio entre el directory y el peer.
Estos nuevos mensajes son:

UNREGISTER_SERVER: Este mensaje lo manda el peer al directory y con el manda su puerto, ya que ahora es efímero.

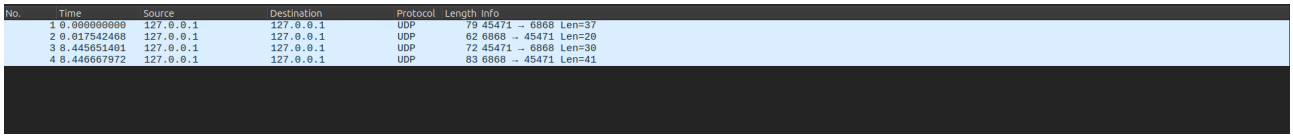
Ahora el directory tiene que recorrer sus estructuras donde tiene guardad los metadatos de los fichero y servidores disponibles y eliminar de ellos tanto el servidor como los ficheros que este tiene asociado. Una vez eliminados el directory responder con UNREGISTER_SERVER_OK o UNREGISTER_SERVER_FAIL, según sea exitosa la operación.

- Por último la mejora del download en paralelo, una un poco más compleja porque ahora de un mismo hilo se crean varios los cuales tienen que modificar variables compartidas y escribir sobre un mismo fichero, por lo que necesitamos un mecanismo que coordine su concurrencia (para este apartado ha sido muy útil los conocimientos de la asignatura de PCD). Una breve explicación de como lo he hecho:
Una vez el peer cliente tiene la lista de InetSocketAddress depurada (es decir ha eliminado aquellos que por alguna razón no están disponibles) crea una nuevo hilo (en segundo plano) con un NFConnector por cada InetSocketAddress disponible. Antes de seguir, he de decir que he creado una clase de apoyo para controlar errores durante la descarga concurrente, la clase es

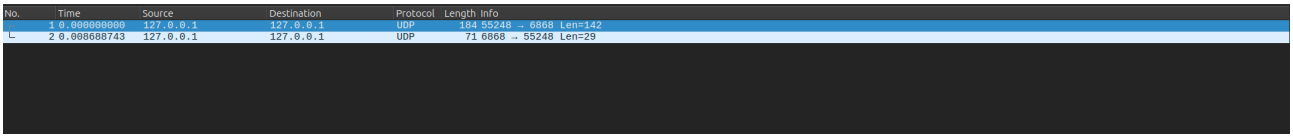
DownloadException, la cual hereda de Exception de java.
A groso modo, el hilo los que hace es leer una variable compartida, para lo cual antes pide su acceso en exclusión mutua, y comprueba que el chunk que le toca descargar a ese hilo no haya sido descargado por otro hilo, si es el caso, lo marca como descargado y realiza la operación de pedir a su peer servidor correspondiente dicho chunk. Una vez con ese chunk en formato bytes lo que hace es pedir otra exclusión mutua para poder escribir sobre el fichero destino de la descarga y una vez accede escribe esos bytes en la posición correspondiente. Si durante la descarga algo fallase cualquier hilo lanzaria una excepcion de la clase DownloadExceptión, esta es atrapada y se modifica una variable llamada “downloadFail” declarada como volatile para simular que su modificación es una instrucción atómica, esta variable es comprobada por cada hilo antes de empezar la descarga de un nuevo chunk. Por lo que si algún hilo tiene un problema durante la descarga todos los hilos abortan y no se realiza la descarga, eso si, pudiendo volver a solicitar de nuevo la descarga.

## 7. Capturas de pantalla de WireShark

A continuación una captura del intercambio entre el NanoFiles y el Direcotry (todo en local) de la ejecución del ping (primer intercambio) y de un filelist (segundo intercambio):




La siguiente captura es de un serve, el cual primero informa al direcotry que va a comenzar a servir y pasa su lista de ficheros.




## 9. Conclusión

Tras finalizar el proyecto y ver lo que hemos sido capaces de “crear” la satisfacción es muy grande ya que es el primer gran proyecto de programación al que me enfrento y del cual estoy satisfecho del resultado.

Ver como semana a semana el proyecto iba cogiendo forma y adquiriendo nuevas funcionalidades es algo que me ha mantenido con ganas de seguir avanzando, además pasar tanto tiempo programando en java ha hecho que mi conocimiento sobre el lenguaje aumente mucho, el aprender a lidiar con todos los problemas que surgen, aprender a manejar excepciones y buscar soluciones ha hecho que aprenda mucho. También de manera indirecta ha hecho que por mis medios aprenda más conceptos e investigue sobre redes, por ejemplo, durante el desarrollo pensé en implementar el cifrado mediante RSA de los mensajes lo que ha hecho que aprenda e investigue

sobre el tema y el como se cifran las cosas en la red, finalmente por tiempo y porque he considerado que un programa de este tipo no es necesario el encriptado, decidí no implantarlo.

Por ultimo y como conclusión, decir que me ha parecido un proyecto de programación muy chulo y entretenido, de primeras un poco abrumador por no entender ciertas cosas pero finalmente ver que funciona y entender plenamente su funcionamiento y el funcionamiento de servicios peer-to-peer así como lo que puede haber detrás de las comunicaciones entres hosts hace que te termine gustado el trabajo.
Eso es todo, Sergio.