MEMORIA REDES DE COMPUTADORES – Nanofiles

Ángel Ruiz Fernandez & Marco Aurelio Santos Gómez
2° de Informatica – Grupo 2.2 – Profesor Eduardo Salvador Iniesta Soto

TABLE OF CONTENTS

1. Introducción

El presente documento tiene el objetivo de describir el desarrollo del proyecto de prácticas Nano-Files de la asignatura Redes de Comunicaciones. Cursada en el segundo año del Grado Ingeniería en Informática de la Universidad de Murcia.

El proyecto consiste en el desarrollo de un sistema de compartición y transferencia de archivos en red usando una metodología Peer-to-Peer. A continuación se especifican las decisiones de diseño tomadas.

2. Protocolos

El sistema está conformado por un conjunto de peers que pueden compartir archivos entre sí usando el protocolo TCP. En adición, los peers mantienen contacto con un servidor directorio que contiene información general del servicio Nano-files. Dice qué archivos están disponibles y qué peers sirven dichos archivos. La comunicación con dicho directorio se efectúa mediante protocolo UDP.

En esta sección se especifican los formatos de los mensajes usados en la comunicación Peer-Peer y Peer-Directorio. Así como el comportamiento de estos en cada momento del intercambio de información.

Notación
- asdf: static data
- < >: mandatory field
- [ ]: optional field
- [...]: last object N times
- All sizes are in bytes

2.1. Directorio

Codificados en texto plano UTF-8 siguiendo el formato codigo:valor sin espacios en blanco y sobre protocolo UDP.

2.1.1. Mensajes de cliente

2.1.1.1. Ping request

Prueba la comunicación con el directorio y la compatibilidad del protocolo.

- Operación: ‘ping‘
- Campos: Identificador de protocolo.
- Respuesta: ping reply

operation: ping
protocol: <protocol id>

2.1.1.2. Filelist request

Solicita información de los archivos conocidos por el directorio.

- Operación: ‘filelist‘
- Campos: ninguno.
- Respuesta: filelist reply

operation: filelist

2.1.1.3. Publish request

Informa al directorio de la lista de archivos disponibles para descargar desde este cliente.

- Operación: ‘publish‘
- Campos: (ver más abajo), la lista puede estar vacía.
- Respuesta: publish response

```
operation: publish
[port: <port>]
file: <hash1>; <filename1>; <size1>
file: <hash2>; <filename2>; <size2>
[...]
```

2.1.2. Mensajes del directorio

2.1.2.1. Ping reply

Respuesta de ping informando la compatibilidad del protocolo.

- Operación: ‘pingok‘
- Campos: ninguno.
- Responde a: ping request

```
operation: pingok
```

2.1.2.2. Ping bad reply

Notifica al cliente de que usa el protocolo incorrecto.

- Operación: ‘pingbad‘
- Campos: ninguno.
- Respuesta a: ping request

```
operation: pingbad
```

2.1.2.3. Filelist reply

Devuelve una lista con información de todos los ficheros conocidos por el directorio (nombre, firma hash, tamaño y peers)

- Operación: ‘filelistres‘
- Campos: (ver más abajo), la lista puede estar vacía.
- Respuesta a: filelist request

```
operation: filelistres
file: <hash1>; <filename1>; <size1>; <server1a>, <server1b> [...]
file: <hash2>; <filename2>; <size2>; <server2a>  <server2b> [...]
[...]
```

2.1.2.4. Publish reply

Confirmación del mensaje publish request.

- Operación: ‘publishack‘
- Campos: ninguno.
- Respuesta a: publish request

operation: publishack

2.2. Peer-to-Peer

Mensajes con formato binarios representados en little-endian sobre protocolo TCP.

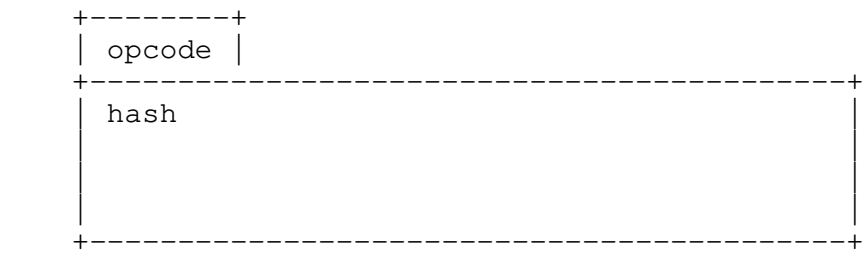
Todos los mensajes comienzan con un código de operación.

2.2.1. Mensajes de cliente

2.2.1.1. File request

Consulta la disponibilidad de un archivo para ser descargado.

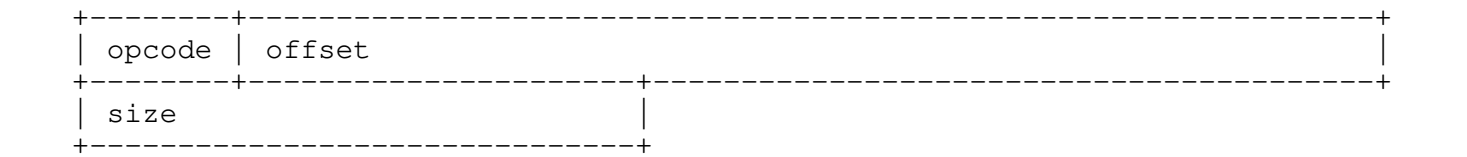
- Código: 0x01
- Campos:
- - Hash[20]: firma hash del fichero.
- Respuestas: accepted o file not found error



2.2.1.2. Chunk request

Solicita al servidor el envío de un archivo.

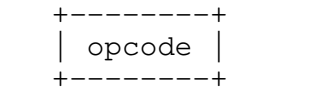
- Código: 0x02
- Campos:
- - Offset[8]: byte de inicio del chunk.
- - Size[4]: tamaño del chunk.
- Respuestas: chunk o bad chunk request error



2.2.1.3. Stop download

Termina la descarga actual de un fichero.

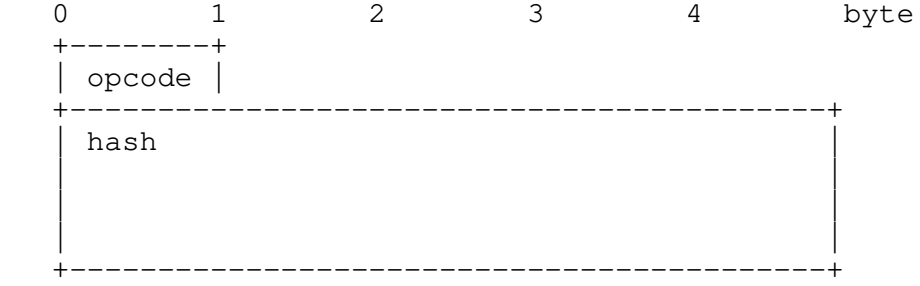
- Código: 0x03
- Campos: ninguno.
- Respuestas: ninguna.



2.2.1.4. Upload request

Solicita a un servidor la subida de un archivo.

- Código: 0x04
- Campos:
- - hash[20]: firma hash del fichero.
- - Respuestas: 'accepted' o 'File already exists'.

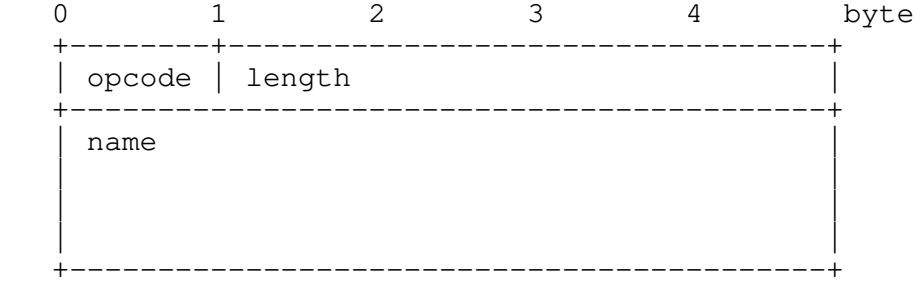


2.2.1.5. File name to save

Da al servidor un nombre con el que guardar un archivo.

Este mensaje es enviado solo si el mensaje upload request fue aceptado previamente.

- Código: 0x05
- Campos:
- - length[4]: lenght of name in bytes
- - name[length]
- Respuestas: 'accepted' o 'file not found error'

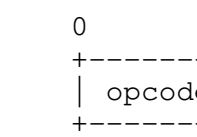


2.2.2. Mensajes de servidor

2.2.2.1. Accepted

Informa de la disponibilidad de un fichero para ser descargado vía chunk requests.

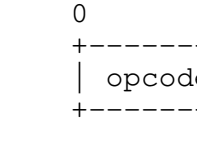
- Código: 0x11
- Campos: ninguno.
- Respuesta a: file request



2.2.2.2. Bad file request error

Fichero no disponible o no encontrado.

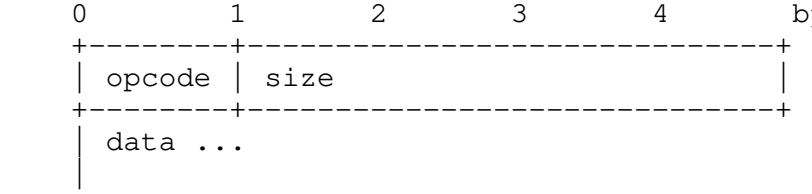
- Código: 0x12
- Campos: ninguno.
- Respuesta a: file request



2.2.2.3. Chunk

Chunk de datos del fichero.

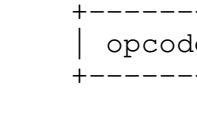
- Código: 0x13
- Campos:
- - size[4]: tamaño del chunk.
- - data[size]: datos.
- Respuesta a: chunk request



2.2.2.4. Bad chunk request error

Fichero no disponible o no encontrado.

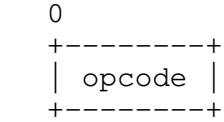
- Código: 0x14
- Campos: ninguno.
- Respuesta a: chunk request



2.2.2.5. File already exists

El fichero ya existe en este peer.

- Código: 0x15
- Campos: ninguno
- Respuesta a: 'upload request'



3. Mejoras

3.1. Quit

El se mejoró el comportamiento del comando quit para que al ejecutarse se actualice en el directorio la información relativa al peer que hace el quit: se da de baja a dicho peer junto con los ficheros que exclusivamente este compartía.

3.2. Serve con puerto efimero

Originalmente el comando serve montaba un servidor escuchando en el puerto 10000. Con esta mejora el servidor queda escuchando en un puerto aleatorio que se registra en el directorio.

3.3. Filelist

La version basica de filelist solo muestra información basica de los ficheros registrados en el servidor (nombre, firma hash, tamaño). Con está ampliación también se muestra el listado de peers (direciones) que disponen de dichos ficheros.

3.4. Upload

Comando que permite a un servidor enviar un documento a otro servidor siempre y cuando este último no lo tenga ya. Para poder ejecutar este comando el peer debe darse de alta como servidor en el directorio previamente.

4. Capturas de pantalla de Wireshark

Véase directorio de imagenes.

```
$ tcpdump -A -v -i lo udp port 6868
tcpdump: listening on lo, link-type EN10MB (Ethernet), snapshot length 262144 bytes
20:00:25.320314 IP (tos 0x0, ttl 64, id 18336, offset 0, flags [DF], proto UDP (17)
, length 64)
    localhost.42185 > localhost.6868: UDP, length 36
E..@G.@.@..
.............,.?operation: ping
protocol: 20032005


20:00:25.359745 IP (tos 0x0, ttl 64, id 18338, offset 0, flags [DF], proto UDP (17)
, length 47)
    localhost.6868 > localhost.42185: UDP, length 19
E../G.@.@...................operation: pingok


20:00:45.575357 IP (tos 0x0, ttl 64, id 19297, offset 0, flags [DF], proto UDP (17)
, length 64)
    localhost.42220 > localhost.6868: UDP, length 36
E..@Ka@.@..I.............,.?operation: ping
protocol: 20032005


20:00:45.576676 IP (tos 0x0, ttl 64, id 19298, offset 0, flags [DF], proto UDP (17)
, length 47)
    localhost.6868 > localhost.42220: UDP, length 19
E../Kb@.@..Y................operation: pingok


20:00:51.952667 IP (tos 0x0, ttl 64, id 19761, offset 0, flags [DF], proto UDP (17)
, length 225)
    localhost.42185 > localhost.6868: UDP, length 197
E...M1@.@...................operation: publish
port: 36739
file: a3f6200e5178f9c4591735e878c8716b46a90a47; FreeBSD-14.1-RELEASE-amd64-disc1.is
o; 1173886976
file: f24656bbedcc2a584fee3b405e19e935f7c171a3; ihatejava.txt; 1212


20:00:51.968637 IP (tos 0x0, ttl 64, id 19765, offset 0, flags [DF], proto UDP (17)
, length 51)
    localhost.6868 > localhost.42185: UDP, length 23
E..3M5@.@...................2operation: publishack


20:00:59.416424 IP (tos 0x0, ttl 64, id 20543, offset 0, flags [DF], proto UDP (17)
, length 49)
    localhost.42220 > localhost.6868: UDP, length 21
E..1P?@.@..z................0operation: filelist


20:00:59.450216 IP (tos 0x0, ttl 64, id 20546, offset 0, flags [DF], proto UDP (17)
, length 251)
    localhost.6868 > localhost.42220: UDP, length 223
E...PB@.@...................operation: filelistres
file: f24656bbedcc2a584fee3b405e19e935f7c171a3; ihatejava.txt; 1212; localhost:3673
9
file: a3f6200e5178f9c4591735e878c8716b46a90a47; FreeBSD-14.1-RELEASE-amd64-disc1.is
o; 1173886976; localhost:36739
```

```
20:01:14.705968 IP (tos 0x0, ttl 64, id 21560, offset 0, flags [DF], proto UDP (17)
, length 49)
    localhost.42220 > localhost.6868: UDP, length 21
E..1T8@.@.................0operation: filelist


20:01:14.708010 IP (tos 0x0, ttl 64, id 21561, offset 0, flags [DF], proto UDP (17)
, length 251)
    localhost.6868 > localhost.42220: UDP, length 223
E...T9@.@.................operation: filelistres
file: f24656bbedcc2a584fee3b405e19e935f7c171a3; ihatejava.txt; 1212; localhost:3673
9
file: a3f6200e5178f9c4591735e878c8716b46a90a47; FreeBSD-14.1-RELEASE-amd64-disc1.is
o; 1173886976; localhost:36739
```

Fig 1. Captura de tcpdump

5. Conclusiones

La compartición de archivos peer-to-peer es un gran metodo para compartir archivos, es extremadamente eficiente y robusta gracias a su descentralización y división de contenido en piezas (chunks) (siempre y cuando hayan peers) y permite que los operadores de los directorios se desmarquen en caso de que se compartan archivos cuyo contenido sea por lo que sea, legalmente gris.

Sin embargo, no todos los protocolos de compartición P2P son igual de buenos. En este caso se ha hecho lo que se ha podido. A continuación, una breve opinión sobre lo que se podría mejorar.

5.1. Valoracion personal

Por qué el proyecto base de Nanofiles es perjudicial: Un analisis

5.1.1. Elección del lenguaje

Java era un lenguaje ampliamente usado para escribir software nuevo... hace 15 años. Ahora se considera como mala opción, como deuda tecnica (por razones fuera del alcance de este documento) no se escribe nuevo software en el, solo se mantiene el antiguo porque es mas barato que reescribirlo.

Especificamente, Java es una opción terrible para este caso porque es un obstaculo para el objetivo del proyecto: aprender a escribir aplicaciones con sockets.

Java, siendo OOP puro, está sujeto a limitaciones que le impiden representar fielmente como funcionan las cosas. La API tiene una serie de clases para representar datagramas UDP por ejemplo: los datagramas existen, pero la clase DatagramPacket es algo muy abstracto que en ningún momento representa el paquete real, sino juntar datos en una unidad porque sí. Java no sabe lo que es un datagrama realmente.

Que queremos enviar un cacho de datos a varias direcciones: tenemos que construir un DatagramPacket para cada vez, con los mismos datos, pero con una dirección de destino distinta. Java está lleno de estas construcciones, copias copias y copias y destrucciones completamente innecesarias.

Cada día nos alejamos mas de C: La API de sockets de Java está demasiado alejada de la verdad: la API de sockets POSIX. Esa que todos los sitemas operativos reales implementan de una forma u otra desde BSD 4.2 (incluso Windows), que es lo que la JVM realmente usa por debajo para hablar con la red.

Esta API POSIX, tenemos unos datos, y los enviamos a una direccion, en una sola llamada, y sin ningún tipo de construcción o copia de nada. Es el kernel el que coje el búfer de memoria, construye el paquete UDP, lo encapsula en un frame Ethernet y se lo pasa al driver de la tarjeta de red (generalmente).

- Java no esta diseñado para trabajar con datos binarios facilmente, teniendo que serializar y deserializar todo, y ni si quiera soporta tipos enteros sin signo explicitamente.

- Aparentemente no hay forma de coger un trozo de un archivo y mandarlo por un socket sin copiar el bufer, lenguaje inutilizable, ¿Como se permite esto en 2025?

- Java es muy inflexible en cuanto a operaciones no bloqueantes. POSIX tiene la syscall poll() para monitorear una lista de sockets en un solo thread, y Java tiene algo parecido pero crucialmente. En C puedes responder con llamadas bloqueantes sabiendo que no se van a bloquear, mientras que en Java solo puedes usar cosas no bloqueantes sobre un canal con select, lo cual es pedantico y molesto.

5.1.2. Problemas con el protocolo pedido

En respuesta a la descripción dada de los mensajes: Argumento detrás del protocolo aquí definido

5.1.2.1. Directorio

El protocolo que se describe es mas innecesariamente complicado de lo que podría ser: Es mucho mas eficiente enviar toda la información del directorio en un solo mensaje, incluyendo nombres, hashes, tamaños y peers de cada archivo, ya que en la interfaz de usuario del programa, se va a mostrar todos los ficheros disponibles y se podrán bajar cualquiera de ellos.

Y como el cliente ya tiene toda la información de un golpe, no necesita enviar ningún mensaje para pedir la lista de peers; pero si aún asi quisieramos hacerlo, no debería hacerse en el protocolo mediante una subcadena de longitud variable. Esto da lugar a errores y ambigüedades que se pueden evitar usando directamente el hash SHA1 en binario que ya conoce el cliente para referirse a un archivo concreto y especifico.

Aparte, opino que al ser un protocolo textual, sería infinitamente mas estandar, sencillo y escalable si este protocolo fuese JSON sobre HTTP, haciendo del directorio una API REST, parecido a bittorrent.

5.1.2.2. Transferencia de ficheros

Antes de empezar, remarcar que todo el punto de partir los archivos en piezas (chunks) es de poder obtener las piezas de distintos peers, idealmente de forma randomizada y uniforme, de manera que si un peer no tiene una pieza o falla al enviarla, no cueste demasiado pedirsela a otro peer.

Con ese fin, el cliente debe tener el control granular de poder pedir piezas concretas. En este protocolo se admiten pedir pieas de distintos tamaños ya que se representa el offset y el tamaño, pero como mejora, la sesión debería fijar un tamaño, y en las comunicaciones subsecuentes indicarse indices de piezas.

En el establecimiento de la sesión, al igual que al obtener los peers del directorio, es propenso a errores indicar una subcadena, otra vez. Mejor enviar el hash exacto en binario, y además asi se evita tener que tener un campo de longitud, ya que el mensaje sería de longitud fija (la longitud se infiere del tipo de mensaje). Solo habría dos respuestas posibles, hash encontrado o no, no hay lugar a la ambigüedad en este protocolo.

Se menciona un mensaje del cliente con el hash del archivo, y que sección del archivo descargar mediante dos offsets. Teniendo en cuenta lo anterior, es redundante, y lo segundo se ha exmplicado anteriormente.

Se podría llegar a pensar que el cliente al tener que pedir cada chunk individualmente daría lugar a un overhead de protocolo, pero esto se mitiga muy eficazmente enviando un numero de requests de piezas en el aire (en la ventana de recepción, o en el stack de aplicación del servidor), o lo que se llama pipelining de requests, de forma que el servidor las sirva lo mas rapido que pueda.

También se habla de que los chunks sean de 1KiB de tamaño. Esto es absurdamente pequeño ya que simplemente el header representaría aproximadamente el 1% del trafico de chunks, nada eficiente para archivos grandes o pequeños. En bittorrent se recomiendan mensajes de 16KiB para piezas de 2MiB. Aquí se usan chunks de 2MiB directamente porque no hay concepto de bloque separado de pieza. Así el kernel trabaja más (eficiencia).

Por último, no conseguimos entender la utilidad de obtener el hash del archivo pedido por el cliente al servidor tras establecer la sesión, ya que en el propio establecmimento uno de los parametros es el mismo hash. Redundante.

Como mejora al protocolo se podrían introducir conceptos de control de flujo de bittorrent, como condición de choking y interest.

5.1.3. Problemas con el código dado

5.1.3.1. General

- Legibilidad horrible
- Nombres confusos, ambiguos
- Inconsistencias de patrones
- Innecesariamente complicado
- Verbosidad (mas culpa de Java supongo)
- Exceso de comentarios y además confusos
- Capa sobre capa sobre capa de abstracciones: paso de datos entre clases que no hacen nada innecesariamente
- Enumeraciones, arrays, estructuras innecesarias, como los comandos.

5.1.3.2. Especifico

Nos dan el proyecto para Eclipse, usando el build system de eclipse. Con lo cual nos obligan a usar Eclipse. Los IDEs son un obstaculo para el aprendizaje. Tienen una cantidad de ruido visual increibles, son enormemente complicados de usar y suelen tener bugs y molestias. La alternativa es "convertirlo a gradle" y usar gradle que es otro build system terrible y innecesariamente complejo.

Ejemplo de TODO ambiguo: ¿En cual se crea el hilo?
- NFControllerP2P: "Arrancar servidor en segundo plano creando un nuevo hilo"

- NFServer: "Añadir métodos a esta clase para: 1) Arrancar el servidor en un hilo nuevo que se ejecutará en segundo plano". La verdadera respuesta es que hay que fijarse que NFServer implementa Runnable, con lo cual es esta clase la que hay que pasar a un Thread, y run() el metodo que ejecuta el Thread, siendo este declarado efectivamente en NFControllerP2P.

Ejemplo de TODO confuso e incongruente:
- NFServer implements Runnable: "Añadir métodos a esta clase para: [...] 2) Detener el servidor (stopserver)" Como vas a parar el servidor dentro del mismo thread, si el thread está bloqueado en accept()? Si no se usan sockets no bloqueantes (no los hemos visto) la unica forma de detener el thread es terminandolo a la fuerza desde el thread Main.

Ejemplo de TODO directamente incorrecto:
- NFServer.serveFilesToClient: "Los métodos lookupHashSubstring y lookupFilenameSubstring de la clase FileInfo" lookupHashSubstring no existe en ningún sitio. Además, estas utilidades deberían estar en la base de datos FileDatabase, literalmente son queries a la base de datos. Sin embargo, lookupFilePath sí que está en la FileDatabase, ¿por qué? Es como si se hubieran puesto los metodos de forma aparentemente aleatoria por el codigo. Nota: esta función tampoco sirve a un proposito porque en el servidor se necesita el resto de campos del objeto FileInfo, no solo su path. Debería de haber un solo metodo para buscar en la db por hash o por nombre y devolver el FileInfo, solo que por alguna razón divinamente desconocida, a FileInfo le falta el getter del path. (Nota: solo tiene sentido guardar el path entero en FileInfo, ya que el nombre es parte del path y por tanto redundante)

FileDatabase es AdHoc, solo sirve como base de datos del cliente:
- escanea un directorio
- no puede guardar peers para usarse en NFDirectoryServer

DirMessage intenta representar todos los tipos de mensajes, haciendo la implementación terrible: encima de que estamos en un lenguaje OOP esto deberia ser resuelto por herencia para cada tipo de mensaje.

NFDirectoryServer termina inmediatamente al recibir un mensaje mal formado, esto es una vulnerabilidad de Denial of Service.

NanoFiles (cliente) termina inmediatamente en cuanto un comando falla. Cuando haces un ping y el directorio no responde, termina, en vez de dejarte volver a intentar.

NFController.processCommand()[COM_DOWNLOAD]: A dos metodos aislados se le pasa la misma información inutil (una substring del nombre del archivo). Sin modificar estos, los dos metodos necesitan el hash del archivo -> tendrian que hacer la misma request al directorio para ver a que hash se corresponde (información util) para poder preguntar por A. los servidores y B. a los peers, los chunks del archivo. Diseño fundamentalmente defectuoso. Lo primero que se debería hacer es obtener la información util, el hash, que es con lo que se debe trabajar.