REDES DE COMUNICACIONES

Departamento de Ingeniería y Tecnología de Computadores

UNIVERSIDAD DE MURCIA

INFORME DE PRÁCTICAS
DEL PROTOCOLO NANOP2P

Miembros:

- MOURAD ABBOU AAZAZ
- PABLO JOSÉ ROCAMORA ZAMORA

Grupo: 2

Profesor: ÓSCAR CÁNOVAS REVERTE

ÍNDICE

DESCRIPCIÓN

Esta practica de Redes de Comunicaciones consiste en el diseño y la implementación de un sistema de transmisión de información entre pares (peers) y un servidor(tracker) que cuenta con una base de datos que contiene todos los metadatos e información de los peers que quieren compartir sus ficheros.

La realización de este proyecto la llamaremos NanoP2P, ya que simula al protocolo P2P pero de manera mas simple y reducida.

El mecanismo principal es que cuando un peer quiere compartir sus ficheros, debe darse alta en el tracker, y enviarle la lista de ficheros que contiene en su carpeta local. El tracker recibe información de sus ficheros y los almacena en su base datos.
Este proceso también es necesario para los peers que quieren descargarse ficheros de otros peers. Primero deben darse de alta en el servidor, para acceder a la información de los peers que también están dados de alta. Cuando un peer se descarga un fichero, debe notificar al tracker que lo va a incluir en su lista de ficheros locales.
Durante la descarga de ficheros entre peers, uno hará de servidor (seeder) y otro de cliente (downloader).

La realización de esta practica consta de dos partes:

- **Diseño** : Especificación de formatos de mensaje y autómatas que definan la pasos que debe seguir el protocolo.

- **Implementación**: Implementación en Java de la practica y de los formatos de la parte de diseño, siguiendo los estados del autómatas definidos en el diseño.

DISEÑO

En esta fase, se diseñan los nuevos formatos que usaremos para los mensajes que se utilizaran para la comunicación entre peers.
Se diseña modo de operación de la ejecución del protocolo, tanto en UDP como en TCP, usando autómatas finitos deterministas. El diseño de los autómatas ya lo definimos en la primera entrega de la parte de diseño.
Dentro de esta parte también están especificado los campos de los formatos de los mensajes que utilizaremos para la comunicación TCP.

FORMATOS DE MENSAJE

En nuestra práctica de diseño hemos, diseñado y especificado dos tipos de formatos:

- ***CHUNK_QUERY***: Es te formato de mensaje será utilizado para que un Peer solicite una lista de chunks de un fichero a otro peer, o para solicitar un chunk específico.

- ***CHUNK_QUERY_RESPONSE***: Este formato será utilizado par como respuesta a los mensajes ***CHUNK_QUERY***. Este tipo de mensaje se utilizará para que el Peer cliente envíe una lista de chunks de un fichero o un chunk que se le ha solicitado.

Los campos que pueden aparecer en los tipos de mensaje son:

- **Type:** Indica el tipo de mensaje. Su tamaño es de 1 byte:
Dependiendo del número que sea puede ser :

  - 1: ***GET_CHUNK*** :Un Peer solicita al otro Peer la lista de Chunks que tiene de un determinado fichero, que indica a través de su Hash.

  - 2: ***GET_CHUNK_RESPONSE***: Un Peer informa a otro Peer de la lista de chunks que tiene de un determinado fichero.

  - 3 : ***CHUNK***: Un Peer solicita al otro Peer un Chunk de un fichero específico indicado a través de su Hash.

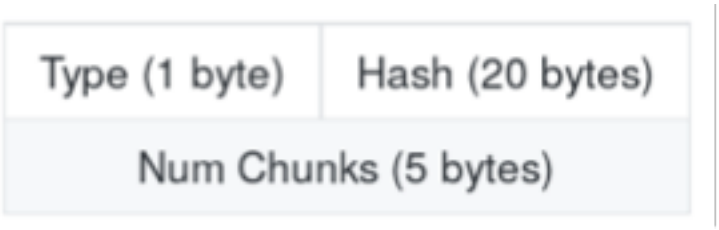
- 4 : **CHUNK_RESPONSE:** Un Peer manda a otro Peer el chunk que le ha solicitado.

- Hash : Es de 20 bytes e indica el hash que identifica a un fichero.

- Num Chunk: Número del chunk del que procede el dato. Es de 5 bytes.

- Chunk : Dato del chunk, el tamaño se establece acorde al tamaño que nos indica el Tracker.

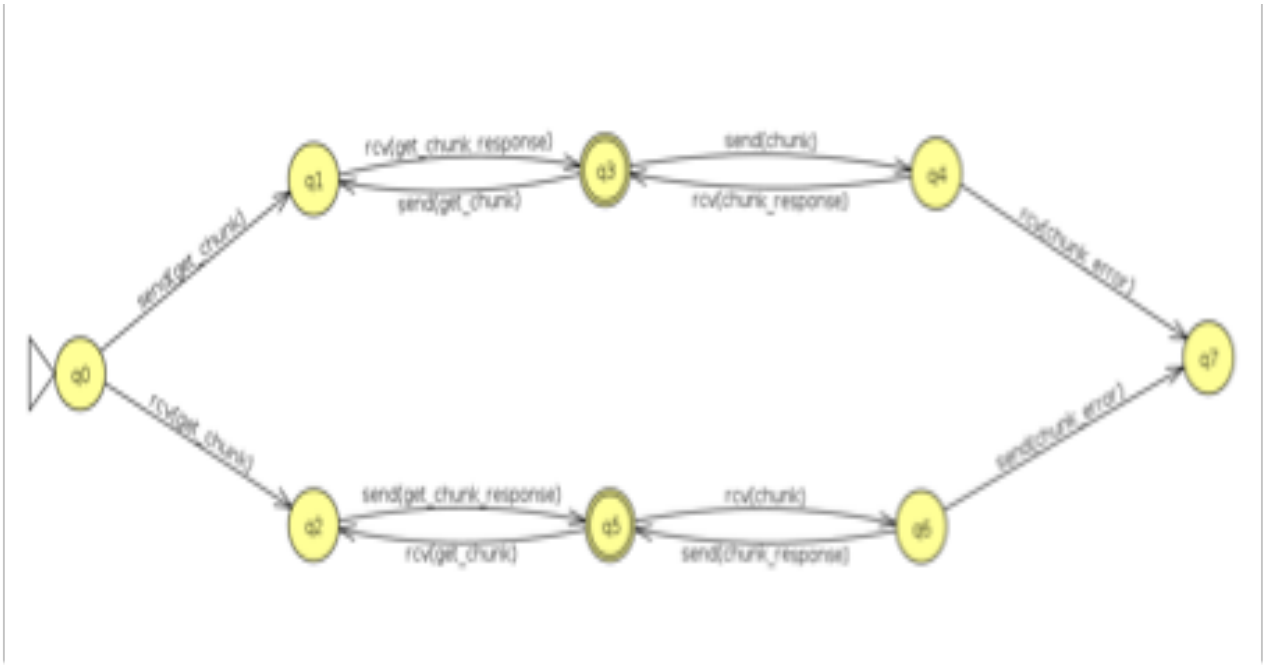
FORMATO GET_CHUNK Y CHUNK



FORMATO GET_CHUNK_RESPONSE Y CHUNKRESPONSE

La función que desempeñan cada uno de los campos en ambos formatos, se encuentra especificada en la práctica de diseño.

AUTÓMATAS DEL PROTOCOLO PEER-PEER



Estado *q0*: Es el estado inicial. En este estado el peer, puede tiene dos opciones:

- Pasar al estado cliente (*q1*)

- Pasar al estado servidor (*q2*)

Estado *q1*: Este estado se da cuando un peer cliente manda una petición de get_chunk al peer servidor. Si no hay errores, el peer cliente debe recibir una lista de chunks y pasa al estado q3.

Estado *q2*: Este estado se da cuando un peer servidor recibe una petición get_chunk de un peer cliente. Si no hay errores, el peer servido debe enviar una lista de chunks y pasa al estado q5.

Estado *q3*: Este estado se da cuando un cliente ha recibido una lista lista de chunks de un fichero. A partir de ahí podrá solicitar trozos de fichero.

Estado *q4*: Este estado ocurre cuando se ha enviado una solicitud un trozo de fichero (chunk). En este estado, volverá al estado q3 si ha recibido el trozo de fichero, o irá al estado q7 en caso de que ocurra algún error.

Estado $q5$: Este estado se da cuando el peer servidor envía una lista de chunks de un fichero al peer cliente. Se mantiene en el estado hasta que se recibe una solicitud de trozo o se vuelve a pedir de nueva una lista de trozos disponibles

Estado $q6$: Este estado se da cuando el peer servidor recibe una solicitud de trozo de fichero. En este caso el peer servidor manda un trozo de fichero al cliente en el que vuelve al estado anterior, o bien notifica un error pasando al estado siguiente.

Estado $q7$:Estado de error que ocurre cuando ha habido algún fallo en el socket de transmisión. Esto puede deberse a que un peer se da de baja o hay un fallo de red.

IMPLEMENTACIÓN DE LOS FORMATOS MENSAJE

Dentro del paquete en donde están todos los ficheros relacionados con la comunicación Peer-Peer, hemos implementado dentro del paquete Message, los formatos de mensaje que hemos diseñado y definido en la parte del diseño.
Así es como hemos implementado los mensajes:

Clase *Message.java*: Es una clase abstracta que representa de forma general los campos de los formatos de mensaje, así como aquellos campos que sean comunes a todos los tipos de formato de mensaje diseñados.

En ella tenemos declaradas las siguientes constantes :

FIELD_OPCODE_BYTES: Representa el byte de operación, y dependiendo del número que sea puede ser:

OP_GET_CHUNK: Indica que se trata de una operación de una solicitud de lista de chunks.

OP_GET_CHUNK_ACK: Indica que se trata de una operación de respuesta a una solicitud de lista de chunks.

OP_CHUNK: Indica que se trata de una solicitud de un chunk

OP_CHUNK_ACK: : Indica que se trata de un envío de un chunk solicitado por el Peer cliente.

FIELD_FILEHASH_BYTES: Representa el tamaño, en bytes, del hash del fichero.

FIELD_NUMCHUNKSIZE: Representa el tamaño de trozo o chunk.

INVALID_OPCODE: Representa una operación invalida o no definida.

Esta clase cuenta con solo un atributo, ***opCode*** (Código de operación), común para todas las clases.

Entre sus métodos principales, están los metodos ***parseRequest*** y ***parseResponse***, que toman como parámetro un DataInputStream y el tamaño de chunk, y se encargan de procesar un mensaje de respuesta o un mensaje de solicitud.

Esta clase, además, se encarga de crear los mensajes necesarios de cada tipo:

makeGetChunkRequest: Toma como parámetros un hash y un long, y crea un mensaje de tipo MessageChunkQuery.

makeGetChunkResponseRequest: Toma como parámetros el número de Chunk, y un array de bits, y crea un mensaje de tipo MessageChunkQuery-Response.

makeChunkRequest : Toma como parámetros el hash y el número de Chunk y crea un mensaje de solicitud de chunk del formato MessageChunkQuery.

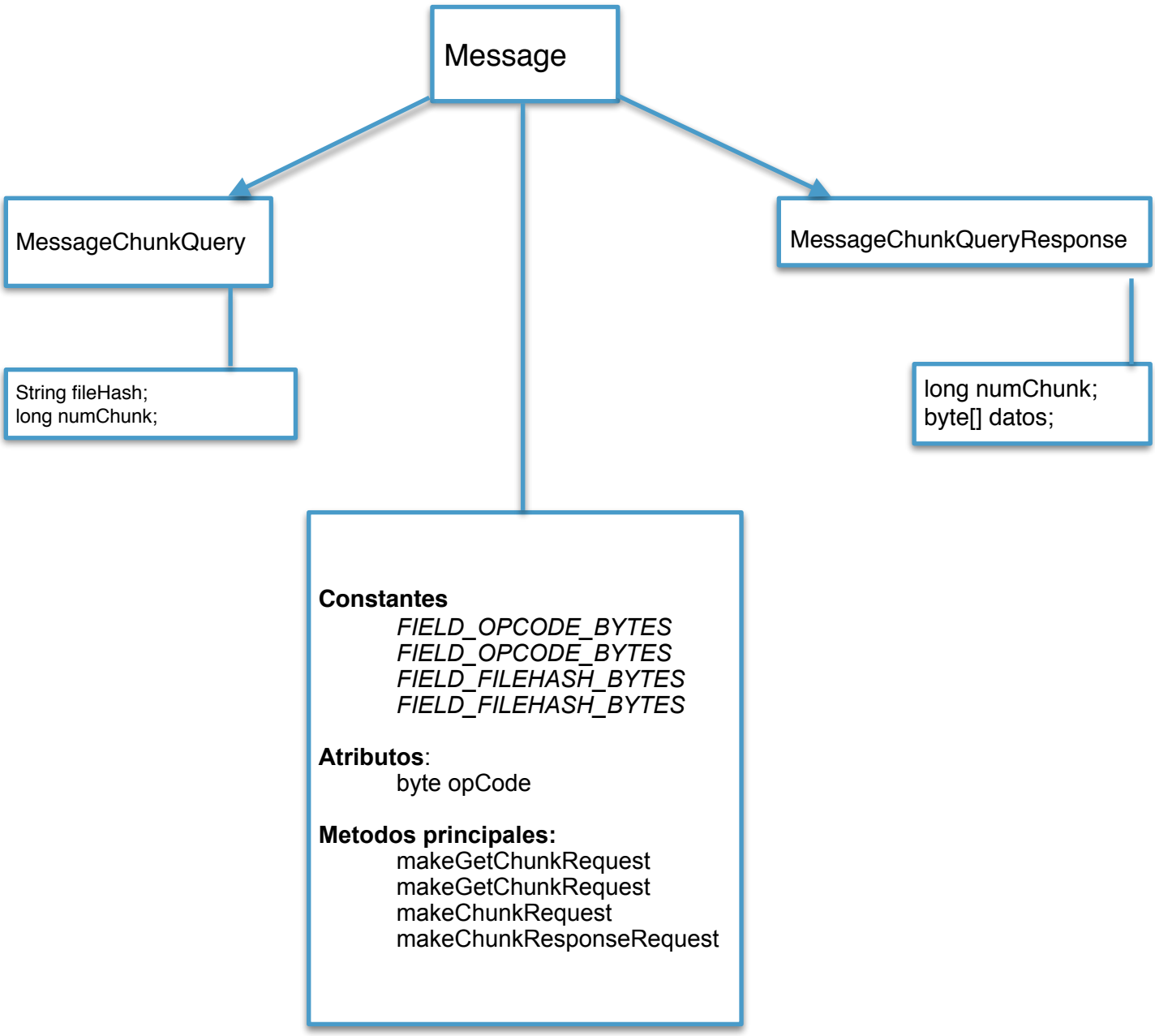
makeChunkResponseRequest: Toma como parámetros, el número de chunk y un array de bits, y devuelve , en respuesta de una solicitud de trozo, con mensaje MessageChunkQueryResponse.

Clase *MessageChunkQuery.java*: Esta clase desciende de la clase Message.java. Con esta clase implementamos el formato MessageChunkQuery de la parte del diseño, con sus campos específicos. Esta clase es la que se utiliza para los mensajes necesarios para solicitar una lista de chunks o un chunk especifico a un peer en la comunicación TCP.
Esta clase tiene como atributos el Hash del fichero, y el número de chunk que ser quiere descargar.
Para crear un mensaje de tipo MessageChunkQuery, se pueden utilizar dos formas:

1. Utiliza un DataInputStream para recoger la información y procesarla.

2. Pasando como atributos un byte que representa el código, el hash del fichero, y el número de chunk. Con esa información, inicializamos los campos.

Clase *MessageChunkQueryResponse.java*: Esta clase también desciende de la clase Message.java. Se utiliza para crear mensajes para el envío de una lista de ficheros o de un chunk específico en respuesta a una solicitud de un Peer cliente.
Se compone de los atributos numChunk, que representa el número de chunk, y un array de bytes que será la información que el mensaje transmitirá al peer cliente. Esta información puede ser la representación en bytes de una lista de chunks, o un chunk específico.



GESTIÓN DE CHUNKS EN LA COMUNICACIÓN TCP

Cuando un peer quiere descargar un fichero, consulta al tracker para que este le diga los peers que contienen ese fichero. El tracker le responde con una lista de seed que contienen el fichero o parte del fichero. Entonces el peer se pone a descargar trozos de cada seed. Cuando recibe el primer chunk del fichero que está descargando, se debe actualizar su lista de ficheros compartidos, añadiendo la información del fichero a su lista y dándose de alta otra vez como seed, con la nueva lista actualizada.

El downloaderThread es el hilo de descarga, y se lanzaran tantos como seeds hayan disponibles. Los *donwloaderThreads* tomarán como variable compartida el downloader, que es controlador principal de la descarga y llevara la cuenta y los estados de los chunks.
Las clases downloaderThread y SeederThread serán las encargadas de llevar a cabo la gestión de trozos. Nuestra clase Downloader recogerá la información necesaria de los downloaderThreads.
Para nuestra gestión de trozos hemos definido varias estructuras de datos en nuestra clase Downloader:

- **chunksDownloadedFromSeeds**: Es HashSet en el que se almacenan los números de chunk que se van descargando.
- **mapaEstados**: Un mapa que empareja un número de chunk con el estado en el que se encuentra: NO_DESCARGADO, EN_PROCESO, DESCARGADO.
- **numChunksDownloaded**: Representa el número de chunks descargados. Inicialmente se encuentra a 0.
- **TotalNumChunks**: El número total de chunks a descargar.

El mecanismo de gestión que nosotros implementamos es el siguiente:
Nuestra clase Downloader dispone de una lista de seeds, de los que se puede descargar chunks de un fichero.
Esta clase lanza tantos hilos DownloaderThreads como número de seed se tengan disponibles. Cada downloaderThread le mandara una petición al Seeder, para que le muestre los chunks que tiene del fichero a descargar. Cuando el Seeder acepte la conexión del socket creará un nuevo SeederThread. Entonces, se podrá iniciar la descarga de chunks.

En ese momento cada downloaderThread le enviará una solicitud de lista de trozos al SeederThread. Cada downloaderThread se ejecutará de manera concurrente, en la que tendrá que esperar a que otro downloaderThread que esté en exclusión mutua, haya terminado de interactuar con el seederThread. El seederThread recibirá la solicitud de lista de chunks. En ese momento tiene pueden ocurrir 2 cosas:

- El Peer servidor también está descargando el mismo fichero. En ese caso SeederThread conectará con su downloader para que le pase los chunks que lleva descargados. Estos chunks se añadirán en una lista y se enviarán al downloaderThread del Peer cliente.
- En caso de que no estar descargando el mismo fichero al mismo tiempo que el Peer cliente, el seederThread busca en su base de datos local hasta encontrar el fichero que se le pide y entonces envía un mensaje de respuesta notificando que tiene todos los chunks de ese fichero

Cuando nuestro downloaderThread disponga de la lista de chunks, con el método *bookNextChunkNumber,* que se ejecuta en exclusión mútua, hará la siguiente comprobación:

1. Si la lista de ficheros está vacía, hacemos un recorrido de los números de Chunk, y si nuestro mapa de estados no contiene alguno, lo insertamos. Por defecto un número de Chunk aparece como estado NO_DESCARGADO. Después, lo retornamos, indicando que es el próximo chunk a descargar. En el mapa de estados lo ponemos en estado EN_DESCARGA.

2. Si la lista contiene chunks sueltos, recorremos dicha lista y si algún número de chunk contenido en la lista no aparece en nuestro mapa de estados, lo añadimos, lo ponemos en descarga dentro de nuestro mapa de estados y lo retornamos.

De esta forma cada downloaderThread, irá comprobando de manera concurrente que chunks faltan por descargar. Es decir, si un downloaderThread se encarga de descargar un chunk, los demás downloaderThreads deberán encargarse de buscar otros chunks que queden por descargar.

Un downloaderThread, tras obtener el chunk que tiene que descargar, crea un mensaje de solicitud del chunk y lo envía por el socket, esperando una respuesta.

Entonces el **seederThread** recibe el un mensaje de tipo MessageChunkQuery del downloader, lo procesa y lo interpreta. Al tratarse de una solicitud de trozo, debe enviar el chunk que se le pide. Para ello busca la ruta de fichero dentro de la lista de ficheros compartida del Peer. De ese fichero coge

el chunk, desde del número de posición que se le pide y lo envía en un mensaje MessageQueryChunkResponse.

Nuestro downloaderThread que está actualmente descargando, al recibir la respuesta con un mensaje MessageQueryChunkResponse, crea un nuevo fichero de tipo `File`, que albergará al nuevo fichero descargado y ahí se escribirá la información. Si el fichero ya está creado, se insertará la información recibida en la posición correspondiente. Después de recibir e insertar el chunk incrementamos el número de chunks descargados.

A continuación, con el método `setChunkDownloaded`, el cual notifica que un chunk se ha descargado satisfactoriamente, comprobamos que el chunk descargado es el primero del fichero. De ser así, actualizamos nuestra lista de ficheros y notificamos al tracker que tenemos un nuevo fichero que compartir. En cualquier caso, actualizamos nuestro mapa de estados poniendo el chunk como DESCARGADO y retornamos un valor *true* en caso de que la descarga del chunk sea un éxito. En otro caso, si no se ha descargado satisfactoriamente, se retorna un valor *false*.

Cada downloaderThread va ejecutando este proceso de petición de chunks, recibir chunks y escribirlos en el nuevo fichero, hasta que el número de chunks descargados sea igual al número total de chunks que hay que descargar.
Después se cerraran los sockets de descarga, y se eliminaran los seeds de la lista de seeds para la descarga.