Formatos de mensajes binarios

Redes de Comunicaciones - Curso 2024/25

Introducción

Mientras que en sesiones anteriores hemos afrontado el diseño e implementación del protocolo de comunicación con el directorio, en este boletín ponemos el foco en el protocolo de transferencia de ficheros entre *peers*, en el marco de la práctica *NanoFiles*. Así, este boletín da indicaciones acerca de cómo diseñar e implementar mensajes binarios multiformato, que serán intercambiados entre un *peer* servidor de ficheros y un *peer* cliente que desea descargar los ficheros disponibles en el servidor.

Recordemos que el protocolo entre el cliente y el servidor de ficheros usará un canal confiable como TCP, por lo que tendremos la certeza de que todos los mensajes enviados llegan a su destino en orden, sin errores y sin duplicidades. Esto tendrá implicaciones en el diseño de los mensajes, como veremos.

Mensajes binarios multiformato

Se crearán distintos tipos de mensajes binarios, cada uno con unos campos, en función del propósito de los mismos y los datos que llevan aparejados. Puesto que el tamaño del mensaje será variable, es conveniente que el mensaje comience con un campo común a todos los formatos (por ejemplo, un byte a modo de *Opcode*), a partir del cual se pueda deducir cuál será la estructura restante del mismo. Date cuenta que el byte *Opcode* en **cada tipo de mensaje debe tener un valor único**. Por ejemplo, el mensaje para solicitar la descarga de un fichero podría utilizar como *Opcode* el valor 0x01, mientras que para enviar los datos de un fragmento del fichero como respuesta podría usarse el valor 0x02. A partir de valor de este primer byte, el receptor del mensaje podrá deducir el formato del mensaje, es decir, qué campos esperar tras el campo *Opcode*.

A continuación se proporciona un conjunto de ejemplos, **los cuales no tienen por qué usarse en todas las circunstancias y pueden ser modificados**, para ilustrar el contexto de una comunicación con el servidor de ficheros:

- Formato **Control**: formato de los mensajes enviados para solicitar acciones que no requieren de ningún dato adicional, o para informar sobre el resultado de una operación previamente solicitada, etc. Por ejemplo, el servidor de ficheros podría utilizar un mensaje con este formato para informar al cliente ante la imposibilidad de encontrar el fichero solicitado para su descarga.

Opcode
1 byte

Ejemplo: Mensaje `FileNotFound`, respuesta a una solicitud de descarga de un fichero que no se encuentra.

Opcode
0x01

- Formato **Operaciones**: formato de los mensajes enviados para solicitar algún tipo de operación que implique uno o varios parámetros, todos ellos de **tamaño fijo** (por ejemplo, 4 bytes). Por ejemplo, el cliente de ficheros podría utilizar un mensaje con este formato para solicitar la descarga de un fragmento de un fichero (previamente acordado mediante otros mensajes), indicando mediante sendos parámetros, el desplazamiento del primer byte en el fichero y el número de bytes que se solicitan a partir de dicho *offset*.

Opcode	Parámetro1 (long)	Parámetro2 (int)
1 byte	8 bytes	4 bytes

Ejemplo: Mensaje `GetChunk`, solicitud de descarga de un *chunk* de un fichero previamente acordado, desde el byte en el desplazamiento 65536 (0x10000), con un tamaño de 131072 bytes (0x20000).

Opcode	File offset (long)	Chunk size (int)
0x02	0x00 0x00 0x00 0x00 0x00 0x01 0x00 0x00	0x00 0x02 0x00 0x00

- Formato **TLV (Tipo-Longitud-Valor)**: formato de los mensajes de longitud variable. El tamaño en bytes del campo “Valor" vendrá dado por el valor del campo “Longitud", que le precede en el orden del mensaje. El tamaño del campo "Longitud" dictará el tamaño máximo del campo valor. En el caso de un protocolo diseñado sobre TCP, no existe más limitación acerca del tamaño máximo del mensaje que el del rango de representación del tipo de dato utilizado para codificar la longitud del valor. Por ejemplo, si en un mensaje con formato TLV se utilizase un campo "Longitud" de tamaño 2 bytes (un *entero corto* o *short*), el campo "Valor" que le sigue no podrá superar los 32767 bytes (2^{15}-1), ya que dado que el rango de representación de un entero corto es [-2^{15}, 2^{15}-1] (en Java, todos los tipos numéricos enteros son con signo).

Opcode	Longitud	Valor
1 byte	1/2/4/8 bytes	n bytes

Ejemplo: Mensaje `UploadFile`, solicitud de subida de un fichero cuyo nombre es "nanoFilesP2P.zip". El valor del campo "Longitud" (0x10=16) indica que el campo "Valor" tiene un tamaño de 16 bytes ("nanoFilesP2P.zip" es una cadena de 16 caracteres ASCII).

Opcode	Longitud (2 bytes)	Valor
0x05	0x00 0x10	nanoFilesP2P.zip

Implementación de los mensajes binarios

Ahora vamos a proporcionar indicaciones acerca de cómo crear y procesar mensajes transmitidos a través de segmentos TCP en Java. Los mensajes multiformato se expresan mediante campos binarios que contienen datos primitivos (`byte`, `int`, `long`, `float`, `double`, `boolean`, etc.). Para codificar los mensajes de esta forma, se recomienda el uso de las clases `DataInputStream` y `DataOutputStream` de Java. Estas clases son útiles para enviar y recibir datos a través de sockets al implementar la comunicación entre aplicaciones en

una red, ya que permiten leer y escribir tipos de datos primitivos de manera portable, lo que significa que garantizan una representación consistente de estos tipos de datos a través de diferentes plataformas. Esto es crucial para la comunicación de red, donde las aplicaciones que intercambian datos pueden estar ejecutándose en hardware y sistemas operativos distintos.

Uso de `DataOutputStream` para escribir datos binarios

`DataOutputStream` permite escribir datos en formatos primitivos de manera portable, directamente en un flujo de salida (p.ej, un fichero o un socket). Esto evita la necesidad de convertir manualmente estos tipos de datos a bytes y luego enviar los bytes, lo cual puede ser propenso a errores y diferencias de formato.

Por ejemplo, si quieres enviar un número entero seguido de un double, puedes hacerlo directamente con métodos como `writeInt()` o `writeDouble()`. Esto asegura que los datos se escriban en una secuencia binaria estandarizada, facilitando su correcta interpretación por el receptor.

Uso de `DataInputStream` para leer datos binarios

De manera complementaria, `DataInputStream` se utiliza para leer los datos procedentes de un flujo de entrada, por ejemplo, de un fichero o de un socket. Proporciona métodos para leer directamente tipos de datos primitivos (`readInt()` , `readDouble()` , etc.) del flujo de entrada, sin necesidad de convertir manualmente de bytes a estos tipos.

Ejemplo de uso

En el Aula Virtual tenéis como ejemplo una clase llamada `EjemploDataStreams.java`, que ilustra el uso de `DataInputStream` y `DataOutputStream`. Puesto hasta el próximo boletín de prácticas no veremos cómo programar con *sockets* TCP en Java, en dicho ejemplo los datos se escriben primero a un fichero y luego se leen de dicho fichero (en lugar de enviarlos/recibirlos a través de un socket). No obstante, esto es suficiente para ilustrar lo necesario para proseguir con el diseño e implementación de los mensajes necesarios para el protocolo de transferencia de ficheros entre *peers*.

La clase `PeerMessage` del proyecto *NanoFiles*

El código de partida proporcionado al alumnado contiene un paquete llamado `es.um.redes.nanoFiles.tcp.message`, en el cual se encuentra la clase `PeerMessage.java`. Dicha clase está concebida para modelar los mensajes intercambiados con un peer servidor de ficheros, y se apoya en las constantes definidas en la clase `PeerMessageOps.java`, como veremos a continuación.

Atributos

Cada uno de sus atributos de instancia corresponde con el valor de algún campo, para todos los formatos de mensajes intercambiados entre un *peer* servidor de ficheros y otro *peer* actuando como cliente (receptor) de ficheros. Por ejemplo, la clase ya declara un atributo `opcode` que contendrá el tipo de mensaje; los posibles valores que tomará se deberán definir simbólicamente en la clase `PeerMessageOps.java`. A partir del valor del `opcode`, se podrá deducir tanto el significado del mensaje como su formato exacto (los campos que aparecen a continuación del `opcode`).

En el desarrollo del proyecto *nanoFiles*, el alumnado deberá ampliar los atributos de esta clase para modelar los campos de los diferentes tipos de mensajes diseñados en sus protocolos. Igualmente, se deberán añadir los correspondientes métodos *getter* y *setter* para obtener/establecer sus valores, con las comprobaciones adecuadas referidas a la existencia de dicho campo según el tipo del mensaje del que se trate.

Envío de mensajes

Siguiendo un esquema similar al que vimos en el boletín de mensajes textuales (clase `DirMessage`), la clase `PeerMessage` contiene un método de instancia `writeMessageToOutputStream` (asimilable al método `DirMessage.toString`), cuya implementación los alumnos deberán ampliar. Este método se aplicará sobre un objeto `PeerMessage` para enviar un mensaje binario multiformato por un `DataOutputStream` cuyos campos contengan los datos adecuados, obtenidos a partir de los atributos del objeto. En este método se utilizarán los métodos `writeLong`, `write(byte[])`, etc. de `DataOutputStream`.

Recepción de mensajes

De manera complementaria, la clase `PeerMessage` implementa un método de clase `readMessageFromInputStream`, utilizado para construir y devolver un objeto `PeerMessage` cuyos atributos habrán sido previamente establecidos a partir de los campos de un mensaje leído de un `DataInputStream` (este método sería asimilable a `DirMessage.fromString`). En este caso, se utilizarán los métodos `readLong`, `readFully(byte[])`, etc. de `DataInputStream`.

En particular, para leer un array de bytes completo a partir de un `DataInputStream` (p.ej., leer datos de un fichero enviados por el servidor) se recomienda el uso de `readFully` frente al uso de `read(byte[])`, ya que `readFully` garantiza que a su retorno se han leído del *stream* tantos bytes como el tamaño del array, o bien lanza una excepción en caso de que no sea posible leer dicha cantidad de datos.

Operaciones con ficheros

Dado que el objeto del protocolo de comunicación entre *peers* de *NanoFiles* es la transferencia de ficheros, es necesario programar la lectura y escritura de información en ficheros. La información contenida en dichos ficheros será trasferida entre pares usando los mensajes del protocolo diseñado, en los campos de datos correspondientes. En las siguientes secciones se describe cómo llevar a cabo operaciones con ficheros y las alternativas que encontramos.

Lectura o escritura completa de un fichero

Una primera opción a la hora de trabajar con ficheros es disponer de todo el contenido del fichero en memoria para poder trabajar con él y, por tanto, limitar las operaciones de entrada y salida a simplemente leer el fichero completo o escribirlo completamente.

Dado que estamos interesados en poder leer y escribir ficheros que pueden ser tanto de texto como binarios, vamos a usar la clase `FileInputStream`. Usando esta clase, abrir un fichero con el fin de leer su contenido lo haríamos de la siguiente forma:

```
File f = new File("file.tgz");
DataInputStream dis = new DataInputStream(new FileInputStream(f));
```

Leer todo el contenido de un fichero, una vez abierto, en un array de bytes se puede hacer también de forma sencilla, una vez que sabemos el tamaño del fichero. No obstante, esto no es recomendable si se trata de ficheros de tamaño grande, debido a las posibles limitaciones impuestas por la memoria disponible. Para saber el tamaño del fichero podemos hacer uso del método `length` la clase `File`. Una vez sabemos su tamaño podemos leer tantos bytes como tenga el fichero haciendo uso del método `readFully(byte[])` de la clase `DataInputStream`. Este método lee tantos bytes como longitud tenga el array de bytes que se le pasa como parámetro. Para acabar, debemos cerrar el stream (el fichero).

```
long filelength = f.length();
byte data[] = new byte[(int) filelength];
dis.readFully(data);
dis.close();
```

A la hora de escribir en un fichero, haremos uso de la clase `FileOutputStream`, y su método `write(byte[])`, que escribe tantos bytes como longitud tenga el array de bytes que se le pasa como parámetro.

```
File f = new File("file.tgz");
if (!f.exists()) {
    f.createNewFile();
    FileOutputStream fos = new FileOutputStream(f);
```

```
            fos.write(data);
            fos.close();
        }
```

Leer un fragmento de un fichero

La clase `RandomAccessFile` en Java se puede usar para leer un fragmento específico de un fichero. Es útil cuando necesitas leer o escribir en una posición específica de un archivo, en lugar de procesarlo secuencialmente desde el principio hasta el final. Esto puede ser especialmente útil para archivos grandes o para situaciones en las que solo necesitas acceder a una (pequeña) parte de un archivo. Esto será útil a la hora de implementar la descarga de un fichero desde varios servidores.

El siguiente ejemplo muestra cómo abrir un archivo para lectura, posicionarse en un punto específico del archivo y leer un fragmento de datos:

```
        // Abrir el archivo con permiso de solo lectura
        RandomAccessFile archivo = new RandomAccessFile("ejemplo.txt", "r");
        // Mover el puntero al inicio del fragmento que queremos leer
        archivo.seek(posicionInicial);
        // Array de bytes para almacenar los datos leídos
        byte[] bytes = new byte[cantidadDeBytes];
        // Leer los datos en el array de bytes
        archivo.readFully(bytes);
        // Hacer algo con los datos leídos en `bytes`
        // Cerrar el archivo
        archivo.close();
```

Este código lee un fragmento de `cantidadDeBytes` bytes del archivo `ejemplo.txt`, empezando desde la posición `posicionInicial`. El método `seek` permite posicionar el "puntero" de lectura/escritura en cualquier posición dentro del archivo antes de realizar operaciones de lectura o escritura. Recuerda que, si utilizas `readFully`, debes asegurarte previamente de que no tratas de leer una cantidad de bytes mayor a los que contiene el fichero a partir del desplazamiento donde está posicionado el puntero de lectura/escritura. Ten presente que el puntero se mueve automáticamente tras realizar una lectura o una escritura en el fichero.

Ejercicios propuestos

A la hora de diseñar e implementar el protocolo de transferencia de ficheros, has de tener en cuenta la funcionalidad prevista para el servidor de ficheros, según los requisitos de la práctica que encontrarás en el documento con la especificación de la práctica de *NanoFiles* (disponible en el Aula Virtual).

La funcionalidad mínima obligatoria que debe soportar el protocolo de transferencia de ficheros se limita a ser capaz de solicitar la descarga de un fichero al servidor, quien ha de responder a dicha solicitud con los datos requeridos. Para cumplir con el requisito mínimo de que la descarga de un fichero disponible en múltiples servidores debe involucrar a todos ellos, es necesario que **el protocolo diseñado tenga la capacidad de solicitar a un servidor la descarga de determinados fragmentos o *chunks* de un fichero**, en lugar de todo su contenido. En lo que a esto respecta, el alumnado tiene libertad a la hora de tomar las decisiones acerca de cómo dividir los ficheros en *chunks*, si estos son de tamaño fijo o variable, etc.

En tu diseño has de tener presente también que el canal subyacente (TCP) es confiable y garantiza la entrega en orden y sin errores; por tanto, no es necesario incluir mensajes de confirmación. No obstante, con el fin de que el código del receptor del fichero pueda comprobar su integridad al terminar la descarga (es decir, comprobar que el contenido del fichero descargado es idéntico al fichero original), **es altamente recomendable que el protocolo permita al servidor comunicar al cliente el *hash* del fichero enviado**. Esto permitirá detectar posibles errores de programación en el código de lectura/envío/recepción/escritura del fichero descargado: el cliente podrá comparar el *hash* proporcionado por el servidor con el *hash* calculado a partir del contenido del fichero descargado.

Teniendo en cuenta lo anterior, a continuación se proponen una serie de ejercicios para avanzar en el diseño e implementación de la práctica.

1. Haciendo uso de un documento de texto plano, **diseña** los mensajes necesarios para la funcionalidad básica de solicitud de descarga de chunks de un fichero por parte de un cliente, así como de las posibles respuestas por parte del servidor.

    1. Debes definir los campos que contiene el mensaje de petición, empezando por la operación que lleva a cabo.
    2. Define el formato del mensaje (o mensajes) de respuesta del servidor. Además de una respuesta que contenga los datos del fichero, contempla la posibilidad de que la descarga no se pueda realizar, en caso de que el servidor no sea capaz de identificar el fichero solicitado. Por ejemplo, si el protocolo diseñado opta por enviar un subcadena del nombre para localizar el fichero a descargar, puede que dicha subcadena sea ambigua o que no haya ningún fichero concordante.

2. Implementa los mensajes de tu protocolo de acuerdo con el diseño llevado a cabo en los puntos anteriores. Para ello, sigue los *TODO's* que encontrarás en el código de la clase `PeerMessage`.

3. Prueba el código que has programado en `PeerMessage` para modelar tus mensajes utilizando la clase `PeerMessageTest.java`, cuyo código tienes disponible en el Aula Virtual.

    1. Coloca el fichero de la clase dentro del paquete `nanoFiles.tcp.message`, en el mismo directorio que la clase `PeerMessage`. `PeerMessageTest.java` contiene un método `main` y por

tanto puedes ejecutarla de forma independiente al programa `NanoFiles` para probar tu código de `PeerMessage`.

2. Modifica/amplía el código del `main` de la clase de pruebas siguiendo los *TODO's*. Empieza creando un objeto `PeerMessage`, de uno de los tipos de mensajes definidos en tu protocolo (con los datos que sean necesarios).
3. Escribe el mensaje en un fichero usando `writeMessageToOutputStream`, según la codificación binaria multiformato que has diseñado.
4. Lee el mensaje del fichero creado en el paso anterior, usando esta vez `PeerMessage.readMessageFromInputStream`, de forma que construyas un nuevo mensaje `PeerMessage` a partir de los datos del fichero.
5. Comprueba que ambos objetos `PeerMessage` (mensajes) tienen valores idénticos, comparando los atributos relevantes (campos) que contiene el mensaje, según su tipo.
6. Repite todos los pasos anteriores para cada uno de los tipos de mensajes definidos en tu protocolo de transferencia de ficheros entre pares. Puedes ir añadiendo el código a la clase de pruebas, de forma que finalmente hayas *testeado* que eres capaz de escribir a un fichero y posteriormente leer de fichero todos los tipos de mensajes de manera correcta.