BitCascade: Sistema de descarga de ficheros P2P

Se trata de un proyecto práctico de desarrollo **en grupos de 2 personas** cuyo plazo de entrega termina el **12 de junio**. Tenga en cuenta que si completa esta práctica puede obtener hasta una calificación de 12 puntos (se podría decir que incluye una parte extra).

Objetivo de la práctica

La práctica consiste en desarrollar un sistema de descarga de ficheros de gran volumen de forma simultánea por un número considerable de usuarios utilizando un esquema de tipo P2P. Salvando las distancias, se trata de una funcionalidad similar a la de BitTorrent, aunque muy simplificada, y de ahí el nombre del proyecto: BitCascade.

En cuanto a la tecnología de comunicación usada en la práctica, se ha elegido Java RMI (si no está familiarizado con el uso de esta tecnología puede consultar esta guía sobre la programación en Java RMI). Para desarrollar esta práctica en un equipo solo se requiere tener instalado Java. Para desarrollar esta práctica en un equipo solo se requiere tener instalado Java.

Para afrontar el trabajo de manera progresiva, se propone un desarrollo incremental en cuatro fases, que, además, están descompuestas en una serie de pasos. Por cada fase, se indicará qué funcionalidad desarrollar como parte de la misma y qué pruebas concretas realizar para verificar el comportamiento correcto del código desarrollado.

Arquitectura del sistema

Aunque BitTorrent ha sido estudiado en detalle en la asignatura, en esta sección se realiza un breve recordatorio de la funcionalidad de este sistema identificando, en primer lugar, los tres tipos de nodos presentes en el mismo:

- El nodo que hace público un fichero para su descarga: `Seed` o `Publisher`.
- Los nodos que descargan un fichero: `Leeches` o `Downloaders`. Colaborarán entre ellos para agilizar la descarga de un fichero usando una estrategia P2P: un `leech` descargará los bloques de un fichero tanto del `seed` que lo publicó como de otros `leeches`.

- El `tracker`, que almacena la metainformación asociada a los ficheros que se están descargando. Por cada fichero, entre otros datos como el nombre del fichero, su tamaño y el tamaño del bloque (en terminología de BitTorrent se usa el término pieza y habitualmente se utilizan tamaños entre 32KiB y 16 MiB), guarda el punto de contacto con su `seed` y con los `leeches` que lo están descargando.

A continuación, se repasa someramente su modo de operación:

- El `publisher` contacta con el `tracker` proporcionándole la metainformación del fichero, que incluye sus datos de contacto.
- El `downloader` contacta con el `tracker` solicitándole la metainformación del fichero que pretende descargar. Además, informa al `tracker` de cuál es su información de contacto. Por tanto, en la metainformación del fichero se incluye cómo contactar con el `publisher` y con `downloaders` que están en proceso de descarga del fichero.
- Un `downloader` usa la metainformación del fichero recibida del `tracker` para descargar bloques tanto del `publisher` como de otros `downloaders` previos.

Organización del software del sistema

Antes de pasar a presentar cada una de las fases, se especifica en esta sección qué distintos componentes hay en este sistema. En primer lugar, hay que resaltar que la práctica está diseñada para no permitir la definición de nuevas clases, estando todas ya presentes, aunque prácticamente vacías en la mayoría de los casos, en el material de apoyo. El software de la práctica está organizado en tres directorios, de forma similar a los ejemplos de la guía:

- `common` (paquete `interfaces`): donde están definidas todas la interfaces del sistema, que ya están completamente programadas y no se deben modificar:
  - `Tracker.java`: la interfaz del `tracker`.
  - `Seed.java`: la interfaz del `publisher`.
  - `Leech.java`: la interfaz de un `downloader`.
  - `FileInfo.java`: la clase que se usa en la interacción entre el `tracker` y los `downloaders` que contiene la información asociada a un fichero.

  Al compilar los ficheros contenidos en este directorio se genera un JAR (`common.jar`) con la funcionalidad requerida por todos los nodos del

sistema, existiendo un enlace simbólico en los otros dos directorios que se describen a continuación para tener acceso al mismo.

- `tracker_node` (paquete `tracker`): contiene la funcionalidad del `tracker` en el fichero `TrackerSrv.java`.
- `peer_node` (paquete `peers`): contiene tres clases:
  - `Publisher.java`: la implementación del `publisher`.
  - `DownloaderImpl.java`: la implementación del `downloader`.
  - `Downloader.java`: programa que usa los servicios de `DownloaderImpl` para descargarse un fichero. Se trata de un programa para hacer pruebas que no es necesario modificar.

Recapitulando, solo es necesario modificar tres clases durante el desarrollo de la práctica: `TrackerSrv.java`, `Publisher.java` y `DownloaderImpl.java`. Además de las diversas clases, en los distintos directorios se incluyen *scripts* para facilitar la compilación de las clases y la ejecución de los programas.

Para facilitar el desarrollo de la práctica, se proporciona también como material de apoyo los dos primeros ejemplos de la guía de Java RMI y un ejemplo de acceso a un fichero usando la clase `RandomAccessFile` de Java.

Ejecución de la práctica

Aunque para toda la gestión del ciclo de desarrollo del código de la práctica se puede usar el IDE que se considere oportuno, para aquellos que prefieran no utilizar una herramienta de este tipo, se proporcionan una serie de scripts que permiten realizar toda la labor requerida. En esta sección, se explica cómo trabajar con estos scripts.

Como primer paso, se debería descargar y desempaquetar el código de la práctica:

```
wget https://laurel.datsi.fi.upm.es/~ssoo/SD.dir/practicas/BitCascade.tgz
tar xvf BitCascade.tgz
cd DATSI/SD/BitCascade.2023
```

Para compilar la práctica, existe un script denominado `./compile.sh` en cada uno de los 3 directorios de la práctica. También se dispone del script `./compile_all.sh` en el directorio raíz de la práctica que va invocando los *scripts* de compilación de cada directorio.

En cuanto a la ejecución, se plantea un ejemplo con tres nodos, suponiendo que se ejecutan en la misma máquina (se pueden usar los distintos nodos de `triqui` para hacer una prueba en distintas

máquinas: `triqui1.fi.upm.es`, `triqui2.fi.upm.es`, `triqui3.fi.upm.es` y `triqui4.fi.upm.es`). En primer lugar, arrancamos el `registry` y el `tracker`:

```
cd tracker_node
./start_rmiregistry 23456 &
./execute_tracker.sh 23456 mi_tracker # nombre del tracker para depurar
```

A continuación, arrancamos el `publisher` especificando un nombre que, por convenio, corresponderá a un subdirectorio de `peer_node/bin` donde está almacenado el fichero que se quiere publicar:

```
cd peer_node
mkdir bin/mi_publisher
cp /etc/passwd bin/mi_publisher/Fichero # fichero publicado
./execute_publisher.sh localhost 23456 mi_publisher Fichero 512 # en un
sistema real el tamaño de cada bloque sería mucho mayor (alrededor del
megabyte).
```

Por último, ejecutamos un `downloader` especificando un nombre que, por convenio, corresponderá a un subdirectorio de `peer_node/bin` donde quedará almacenado el fichero que se quiere publicar (no hay que crear el directorio ya que lo hace el propio `downloader`):

```
cd peer_node
./execute_downloader.sh localhost 23456 down1 Fichero
# en peer_node/bin/down1/Fichero debe quedar una copia del fichero
```

Se podrían ir activando más `downloaders` especificando diferentes nombres.

Fase 1: Interacción con el Tracker (3 puntos)

Durante esta fase el `publisher` publica la información de un fichero en el `tracker` y un `downloader` obtiene esa información encapsulada en la clase `FileInfo`. Las distintas operaciones que hay que realizar en esta fase están etiquetadas con `TODO 1` en los ficheros correspondientes.

Para realizar esta funcionalidad puede tomar como base el cliente y el servidor del servicio de eco explicado en la guía, que se incluye también en el material de apoyo.

Paso 1: Alta del `tracker` en el `registry`

En este primer paso de esta fase, el `tracker`, en su función `main`, debe localizar el `registry`, cuyo puerto ha recibido como argumento junto con su propio nombre que se usa para depuración, y dar de alta en el `registry` una instancia de esa misma clase (`TrackerSrv`) que ya se ha creado previamente.

Para comprobar que la funcionalidad es correcta, debe incorporar en `Publisher.java` la operación de `lookup` del `registry` que debe obtener

correctamente la referencia al `tracker` lo que se validará porque se imprime el nombre del `tracker`.

Pruebas

Arrancamos el `tracker`:

```
cd tracker_node
./start_rmiregistry 23456 &
./execute_tracker.sh 23456 mi_tracker # nombre del tracker para depurar
```

Y el `publisher` que realmente no está publicando nada todavía sino verificando que se ha dado de alta bien el `tracker` (el error de fichero ya publicado aparece porque no está completada la funcionalidad):

```
./execute_publisher.sh localhost 23456 mi_publisher cualquier_cosa 512
el nombre del nodo del tracker es: mi_tracker
Fichero ya publicado
```

Paso 2: Publicación en el `tracker` de un fichero por un `Seed`

En este paso, hay que implementar en el `tracker` el método `announceFile` que permite a un `publisher` realizar la publicación de un fichero. Este método debe crear un objeto de la clase `FileInfo` con todos los datos del fichero e insertarlo en una estructura de tipo mapa que debe gestionar el `tracker`, devolviendo un error si el fichero ya estaba publicado.

Por otro lado, en el `publisher` debe instanciarse un objeto de la clase `Publisher`, cuyo constructor ya realiza varias operaciones como obtener el número de bloques que ocupa el fichero que deja disponible en el método `getNumBlocks`, e invocar al método `announceFile` del `tracker`.

Pruebas

Creamos el fichero a exportar:

```
cd peer_node
mkdir bin/mi_publisher
cp /etc/passwd bin/mi_publisher/Fichero # fichero publicado
```

Arrancamos el `tracker` y el `publisher` en dos ventanas diferentes debiendo aparecer las respectivas salidas que se muestran:

```
cd tracker_node
./start_rmiregistry 23456 &
./execute_tracker.sh 23456 mi_tracker
mi_publisher ha publicado Fichero
cd peer_node
./execute_publisher.sh localhost 23456 mi_publisher Fichero 512
el nombre del nodo del tracker es: mi_tracker
Dando servicio...
```

Compruebe además que si se arranca un segundo `publisher` en otra ventana sale el mensaje indicando que el fichero ya está publicado:

```
cd peer_node
./execute_publisher.sh localhost 23456 mi_publisher2 Fichero 512
el nombre del nodo del tracker es: mi_tracker
Fichero ya publicado
```

Paso 3: Obtención de la información de un fichero por un `Leech`

Hay que implementar el método `lookupFile` del `tracker` que obtiene del mapa la información asociada a un fichero.

En el método `init` de `DownloaderImpl`, como ya hicimos en el `publisher`, hay que realizar la operación de `lookup` del `registry` que debe obtener correctamente la referencia al `tracker` lo que se validará porque se imprime el nombre del `tracker`.

Asimismo, hay que llamar al método `lookupFile` para obtener la metainformación del fichero e instanciar un objeto de la clase `DownloaderImpl`.

Pruebas

Arrancamos el `tracker` y el `publisher` en dos ventanas diferentes debiendo aparecer las respectivas salidas que se muestran:

```
cd tracker_node
./start_rmiregistry 23456 &
./execute_tracker.sh 23456 mi_tracker
mi_publisher ha publicado Fichero
cd peer_node
./execute_publisher.sh localhost 23456 mi_publisher Fichero 512
el nombre del nodo del tracker es: mi_tracker
Dando servicio...
```

A continuación, ejecutamos un `downloader` que debe imprimir la información del fichero:

```
cd peer_node
./execute_downloader.sh localhost 23456 mi_downloader Fichero
el nombre del nodo del tracker es: mi_tracker
        tamaño de bloque: 512
        número de bloques: 6
        Seed: mi_publisher
```

Pruebe además a arrancar un segundo `downloader` en otra ventana con un fichero no publicado.

```
./execute_downloader.sh localhost 23456 mi_downloader2 Fichero_noexiste
el nombre del nodo del tracker es: mi_tracker
Fichero no publicado
```

Fase 2: Descarga del fichero del Seed (3 puntos)

Esta fase realiza la descarga de un fichero desde un `seed` a un `leech`:

- En el `publisher` hay que implementar el método remoto `read` que lee usando un `RandomAccessFile` el bloque solicitado de un fichero y lo retorna como resultado del método. El fichero se debe abrir para lectura en el constructor de la clase. Nótese que normalmente el fichero no ocupará un número entero de bloques por lo que habrá que asegurarse de que la última lectura solo devuelve los datos realmente leídos. Téngase en cuenta que el descriptor de un fichero no es serializable por lo que habrá que definirlo con el calificativo `transient` para indicar que ese campo no se debe intentar serializar.
- En el método `downloadBlock` de `DownloaderImpl` hay que solicitar el bloque al `seed` y escribirlo en el fichero. El fichero se debe abrir para escritura en el constructor de la clase vaciando su contenido por si existiera previamente (`setLength(0)`).

Se proporciona como material de apoyo para esta parte un ejemplo de lectura de un bloque de un fichero.

Pruebas

Arrancamos el `tracker`, el `publisher` y un `downloader` en tres ventanas diferentes debiendo aparecer las respectivas salidas que se muestran:

```
cd tracker_node
./start_rmiregistry 23456 &
./execute_tracker.sh 23456 mi_tracker
mi_publisher ha publicado Fichero
cd peer_node
./execute_publisher.sh localhost 23456 mi_publisher Fichero 512
el nombre del nodo del tracker es: mi_tracker
Dando servicio...
publisher read 0
publisher read 1
publisher read 2
publisher read 3
publisher read 4
publisher read 5
cd peer_node
./execute_downloader.sh localhost 23456 mi_downloader Fichero
el nombre del nodo del tracker es: mi_tracker
        tamaño de bloque: 512
        número de bloques: 6
        Seed: mi_publisher
Pulse return para leer el siguiente bloque
Pulse return para leer el siguiente bloque
Pulse return para leer el siguiente bloque
Pulse return para leer el siguiente bloque
Pulse return para leer el siguiente bloque
Pulse return para leer el siguiente bloque
Pulse return para leer el siguiente bloque
```

Se debe comprobar que la descarga ha sido correcta:

```
diff bin/mi_publisher/Fichero bin/mi_downloader/Fichero
```

Fase 3: Descarga del fichero de Leeches y del Seed usando información estática (3 puntos)

En esta fase, un `downloader` irá descargando alternativamente bloques del fichero desde los `leeches` previos y desde el `seed`. Suponiendo tres `leeches`, descargará del primero, después del segundo, continuando por el tercero, seguido del `seed`, y otra vez del primer `leech`. Por tanto, el `downloader` ejerce también el papel de servidor teniendo que convertirlo en un objeto remoto. Téngase en cuenta que en esta fase la estrategia usada presenta bastantes limitaciones porque vamos a descargar de los `leeches` previos solo los bloques que sabemos que estos ya han descargado en el momento que se activa este `leech`. La siguiente fase supera estas limitaciones.

Paso 1: Convertir Downloader en objeto remoto y añadirlo a FileInfo

El primer paso es cambiar la definición de esta clase para que sea de tipo remota.

Asimismo, en el método `init` de `DownloaderImpl` se debe invocar el método `addLeech` del `tracker`, que hay que implementar en esta fase.

Pruebas

Arrancamos el `tracker`, el `publisher` y dos `downloader` en cuatro ventanas diferentes debiendo aparecer las respectivas salidas que se muestran a continuación donde se puede apreciar que en el `FileInfo` del segundo `downloader` ya aparece el primer `downloader` (nótese que en esta prueba no vamos a solicitar descargar bloques):

```
cd tracker_node
./start_rmiregistry 23456 &
./execute_tracker.sh 23456 mi_tracker
mi_publisher ha publicado Fichero
cd peer_node
./execute_publisher.sh localhost 23456 mi_publisher Fichero 512
el nombre del nodo del tracker es: mi_tracker
Dando servicio...
cd peer_node
./execute_downloader.sh localhost 23456 mi_downloader Fichero
el nombre del nodo del tracker es: mi_tracker
        tamaño de bloque: 512
        número de bloques: 6
        Seed: mi_publisher
Pulse return para leer el siguiente bloque
cd peer_node
./execute_downloader.sh localhost 23456 mi_downloader2 Fichero
```

```
el nombre del nodo del tracker es: mi_tracker
        tamaño de bloque: 512
        número de bloques: 6
        Seed: mi_publisher
        Leech: mi_downloader
Pulse return para leer el siguiente bloque
```

Paso 2: Descarga de bloques de nodos alternos

En primer lugar, hay que implementar el método `read` de `DownloaderImpl`, que será básicamente igual que el de `Publisher`.

Para poder descargar de distintos nodos se requiere conocer cuál es el último bloque descargado por los `downloaders` previos en el momento en el que se activa este `downloader`. Se recomienda usar un `ArrayList` para almacenar objetos de la clase `LeechInfo` correspondientes a esos `downloaders` previos.

En este paso, en el constructor de `DownloaderImpl` se debe rellenar esa lista llamando a `getLastBlockNumber` por cada `leech` contenido en `FileInfo`.

En `downloadBlock`, hay que seleccionar de qué *peer* se realiza la descarga usando la siguiente estrategia que busca un cierto grado de equidad:

- Se itera en el `ArrayList` buscando el primer `leech` que tenga descargado el bloque solicitado. Se recomienda iterar directamente usando el método `get` con la posición numérica en el *array*.
- Si hay un `leech` que lo cumple, se descarga de ese nodo. La próxima llamada a este método `downloadBlock` comenzará a iterar justo por el siguiente de la lista.
- En caso de que se alcance el final de la lista se descargará del `seed`.

Pruebas

Arrancamos el mismo escenario que en la prueba anterior pero en este caso con tres `downloaders` y, una vez arrancados el `tracker` y el `publisher`, con la siguiente secuencia de ejecución (recuerde que el fichero de prueba tiene 6 bloques):

- arranca `downloader 1`: por el momento no lee nada.
- arranca `downloader 2`: por el momento no lee nada.
- `downloader 1` lee cinco bloques: todos del `seed`.
- `downloader 2` lee cuatro bloques: todos del `seed`.
- arranca `downloader 3`.
- `downloader 3` lee primer bloque: debe hacerlo de `downloader 1`.
- `downloader 3` lee segundo bloque: debe hacerlo de `downloader 2`.

- `downloader 3` lee tercer bloque: debe hacerlo de `seed`.
- `downloader 3` lee cuarto bloque: debe hacerlo de `downloader 1`.
- `downloader 3` lee quinto bloque: debe hacerlo de `seed`.
- `downloader 3` lee sexto bloque: debe hacerlo de `seed`.

Fase 4: Descarga del fichero de Leeches y del Seed usando información dinámica (3 puntos)

En esta fase un `downloader` va a ser informado dinámicamente por los `downloaders` previos según estos vayan descargándose nuevos bloques siguiendo este modo de operación:

- Como ya ocurría en la fase previa, un nuevo `downloader` contacta con cada uno de los previos, pero no solo para conocer cuál es su último bloque descargado (`getLastBlockNumber`), sino también lo hace (`newLeech`) para solicitar ser notificado cada que se descargue uno nuevo.
- Cuando un `leech` completa una descarga (`downloadBlock`), lo notifica a los `leeches` posteriores mediante `notifyBlock` que actualiza asíncronamente el número del último bloque descargado en la entrada correspondiente del `ArrayList` del `leech` anterior.

En consecuencia, hacen faltan dos listas:

- Una asociada a los `leeches` anteriores que ya se implementó en la fase previa como un `ArrayList` de objetos de la clase `LeechInfo`. El cambio que se produce en esta fase es que se actualiza asíncronamente mediante el método `notifyBlock` el campo que guarda el último bloque descargado. Para facilitar esa actualización evitando tener que iterar sobre el `ArratList`, se recomienda crear en el constructor un mapa sobre esa lista usando como clave la referencia al `leech`, que, suponiendo que hemos llamado `leechList` al `ArrayList`, se podría hacer con esta línea de código:
- `import java.util.Map;`
- `import java.util.stream.Collectors;`
- `import java.util.function.Function;`
- ```
  leechMap =
  leechList.stream().collect(Collectors.toMap(LeechInfo::getLeech,
  Function.identity()));
  ```

  Téngase en cuenta que este `ArrayList` se crea en el constructor y ya no se modifica, no presentando problemas de acceso concurrente.
  ```

- Una lista adicional de los `leeches` posteriores que serán notificados cada vez que este `leech` complete una descarga (solo haría falta guardar en la lista la referencia a cada `leech`). En el método `newLeech` se añadiría el `leech` invocante a la lista de este `leech`. Por su parte, esa invocación se produciría en el constructor de la clase. Al completar la descarga de un bloque se iteraría sobre esa lista para realizar las notificaciones. En este caso, se trata de una lista dinámica que se actualiza concurrentemente por lo que se recomienda el uso de la clase `ConcurrentLinkedQueue`.

Recapitulando, en esta fase habría que hacer las siguientes modificaciones en `DownloaderImpl`:

- En el constructor, se debe crear un mapa sobre el `ArrayList` para simplificar el código de `notifyBlock`. Asimismo, hay que llamar al método `newLeech` de los `leeches` previos.
- Al completar la descarga de un bloque (`downloadBlock`) se debe invocar el método `notifyBlock` de todos los `leeches` posteriores.
- En `newLeech` hay que añadir el `leech` a la lista concurrente.
- En `notifyBlock` se debe actualizar el campo correspondiente al último bloque en la entrada correspondiente del `ArrayList`.

Pruebas

Arrancamos el mismo escenario que en la prueba anterior pero en este caso con dos `downloaders` y, una vez arrancados el `tracker` y el `publisher`, con la siguiente secuencia de ejecución (recuerde que el fichero de prueba tiene 6 bloques):

- arranca `downloader 1` lee primer bloque: debe hacerlo de `seed`.
- `downloader 2` lee primer bloque: debe hacerlo de `downloader 1`.
- `downloader 2` lee segundo bloque: debe hacerlo de `seed`.
- `downloader 1` lee el resto de los bloques: todos del `seed`.
- `downloader 2`: a partir de este punto lee de forma alternada un bloque de `downloader 1` y otro del `seed`.

Material de apoyo de la práctica

El material de apoyo de la práctica se encuentra en este enlace.

Al descomprimir el material de apoyo se crea el entorno de desarrollo de la práctica, que reside en el directorio: `$HOME/DATSI/SD/BitCascade.2023/`.

Entrega de la práctica

Se realizará en la máquina `triqui`, usando el mandato:
`entrega.sd BitCascade.2023`

Este mandato recogerá los siguientes ficheros:

- `autores` Fichero con los datos de los autores:
- `DNI APELLIDOS NOMBRE MATRÍCULA`
- `memoria.txt` Memoria de la práctica. En ella se pueden comentar los aspectos del desarrollo de su práctica que considere más relevantes. Asimismo, puede exponer los comentarios personales que considere oportuno. No es necesario rellenarla en caso de que no tenga nada que reseñar.
- `tracker_node/src/tracker/TrackerSrv.java`
- `peer_node/src/peers/Publisher.java`
- `peer_node/src/peers/DownloaderImpl.java`