Universidad Politécnica de Cartagena

Escuela Técnica Superior de Ingeniería de Telecomunicación

PRÁCTICAS DE SISTEMAS Y SERVICIOS DISTRIBUIDOS

Propuesta del Trabajo de Prácticas 2021/2022

Aplicación para el intercambio de archivos peer to peer

Revisión 1.2

Profesores:
Esteban Egea López

Índice.

Índice. 2
1 Consideraciones generales. 3
1.1 Objetivos 3
1.2 Introducción 3
1.2.1 Qué deben hacer los alumnos 3
1.2.2 Cómo está organizada esta propuesta 3
2 Desarrollo de la aplicación 3
2.1 Extensiones 6
2.2 ¿Java RMI o interfaz Sockets? 6
2.3 Preasignación de archivo 6
2.4 Uso de funciones *hash*. 6
2.5 Paralelización de tareas 7
2.6 Parametrización 7
2.7 Ejecución e identificación de pares 7
2.8 Desarrollo de una aplicación alternativa 8
3 Material a entregar y criterios de evaluación 8
3.1 Memoria y Código 8
3.2 Criterios de evaluación 8
3.3 Grupos de una persona 9
4 Notas adicionales para la implementación 9
4.1 Encapsulación 9
4.2 Colecciones y contenedores 10
4.3 Gestión de errores 10
5 Bibliografía 10

1 Consideraciones generales.

1.1 Objetivos

En este trabajo de prácticas los alumnos realizarán por parejas una aplicación para la descarga de archivos *peer-to-peer*. Implementarán una versión simplificada del protocolo *bittorrent*.

1.2 Introducción

La aplicación a desarrollar imitará de manera simplificada el funcionamiento de *bittorrent*. Este protocolo permite la descarga eficiente de archivos entre pares. Durante el desarrollo de un sistema de este tipo aparecen problemas típicos de los sistemas distribuidos como la necesidad de coordinación y ejecución de tareas en paralelo o la necesidad de adaptación de estructuras de datos a sistemas y plataformas (hardware+sistema operativo) heterogéneas y su transporte sobre la red. El uso de Java, con sus capacidades multiplataforma, resuelve algunos de estos problemas y facilita el desarrollo. En particular, elimina el problema de la adaptación del formato de los datos a plataformas distintas. Aun así, permite que el alumno se familiarice con estos conceptos aunque sea de manera simplificada.

La aplicación que se desarrollará, aunque se describe con más detalle en apartados posteriores, funciona a grandes rasgos de la siguiente manera: existe un conjunto de procesos que serán los pares (*peer*) y un proceso servidor que hará de *tracker*. Los pares realizan dos tareas: por una parte tienen archivos que se encargan de compartir (ofertar) con el resto de pares, por otra parte, actúan de cliente en la descarga de un archivo determinado. El *tracker* es un proceso especial que se encarga de coordinar a los pares en la descarga de los archivos. Para compartir un archivo se realizan las siguientes acciones: inicialmente, existe un par que tiene el archivo a compartir completo, este proceso se denomina *seed*. Antes de compartir el archivo, el *seed* dividirá (virtualmente) el archivo en bloques y calculará un *hash* de cada uno de los bloques. El *seed* registrará en el *tracker* su información de acceso (cómo se puede conectar con él) e informará del archivo que posee, pasando la lista de *hash* de los bloques correspondientes al archivo. Cuando un cliente quiera descargar el archivo, se conectará con el *tracker* que le informará de los archivos disponibles y sus correspondientes *seed*, además del resto de pares que están descargando el archivo en ese momento. El conjunto de pares que participa en la descarga compartida de un archivo se denomina *swarm* (enjambre). A partir de ese punto, el cliente conectará con los pares y les solicitará bloques del archivo que quiere descargar. A medida que descarga esos bloques, informará al resto del enjambre de los bloques que posee y estará en condiciones de servir él mismo esos bloques a otros pares.

Los alumnos podrán elegir la forma de la implementación: mediante Java RMI o mediante el uso de Sockets directamente. Ambas alternativas tienen sus ventajas e inconvenientes.

Finalmente, se ofrece la posibilidad de que el alumno extienda la funcionalidad requerida, innove o incluso implemente una aplicación totalmente distinta, como se detalla en los siguientes apartados.

1.2.1 Qué deben hacer los alumnos

Los alumnos deben implementar por parejas en Java una aplicación con la funcionalidad que se especifica en este documento. La aplicación debe ser completamente funcional y debe poder ejecutarse y probarse. De hecho, **los alumnos dejarán en el servidor del laboratorio un ejecutable** de la aplicación, que será probado por el profesor.

Se deberá además entregar **una memoria del trabajo** realizado. El contenido de la memoria y los criterios de evaluación también están descritos en esta propuesta.

1.2.2 Cómo está organizada esta propuesta.

En el apartado 2 se describe el funcionamiento de la aplicación solicitada la funcionalidad requerida y se detallan algunas partes de la misma.
En el apartado 3 se especifican los criterios de evaluación y la memoria y el material que se debe entregar.
Finalmente, en el apartado 4 se dan algunas sugerencias sobre la implementación y el uso de librerías.

2 Desarrollo de la aplicación

El objetivo de la aplicación, como se ha dicho, es el proporcionar un servicio de descarga de archivos entre pares. La aplicación se dividirá en dos módulos funcionales: proceso *tracker* y proceso *peer*.

A partir de las definiciones anteriores, la **funcionalidad mínima requerida** (consulte los criterios de evaluación en 3.2) es la siguiente, que se describirá siguiendo los pasos necesarios para el funcionamiento de la aplicación. **Para la funcionalidad mínima solo se requiere que se permita la descarga de UN ÚNICO archivo**. Extender la funcionalidad a **múltiples archivos** se considera una **extensión**.

1) **Todos** los procesos **clientes o pares (peer)** deben ser capaces de actuar como ***seed*** inicial. Para ello deben ser capaces de:

- **a)** **Seleccionar al menos un archivo a compartir** (que se pasará como parámetro al ejecutar el cliente o de otra forma).
- **b)** **Crear un resumen ("torrent") del archivo**: esta acción consiste en dividir el archivo en bloques de 256 KB ó 512 KB y generar un resumen *SHA-1* o *MD5* de cada bloque. Estos resúmenes de cada bloque se almacenarán en una estructura de datos (puede ser otro archivo o en una colección o estructura de datos adecuada en memoria) que representan un resumen del archivo a compartir. Nótese que **no hay que realizar una división real del archivo**, es decir, no es necesario separar el archivo en otros archivos de 256 KB, basta con leer y calcular los *hash* de los bloques y guardar el resultado en la estructura de datos correspondiente.
- **c)** **Registrar en el *tracker* el resumen ("torrent") que se comparte**. De manera que el *tracker* pueda a su vez informar al resto de los pares de los archivos disponibles. Es decir, el tracker enviará a los clientes el "torrent" cuando se conecten por primera vez.
- **d)** **Registrar su propia información de acceso**. Es decir, la dirección IP y el puerto que escucha, si se utilizan Sockets, o el nombre del objeto remoto, si se utiliza RMI.

De este modo, cuando un par actúa como *seed*, primero genera una lista de *hash* del archivo que comparte, después registra esta información en el *tracker* y después queda a la espera de peticiones de bloques del archivo de otros pares.

2) El proceso ***tracker*** se encarga de coordinar a los pares. **La IP/puerto del tracker** o bien **un nombre de objeto remoto** es conocido por todos los pares. De esta manera los pares siempre saben cómo comunicarse con el tracker La funcionalidad del tracker es la siguiente:

- a) **Permite el registro del resumen ("Torrent") del archivo** que se está compartiendo, de manera que un *seed* pueda proporcionar dicha información.
- b) **Permite el registro de información de acceso de los pares**. Los pares se registrarán con el *tracker*, proporcionando la información que permite conectarse a ellos: IP/puerto o nombre de objeto remoto.
- c) **Permite el registro de *seed*/descarga**. Los pares registrarán en el *tracker* si están descargando el archivo o hacen de *seed*.
- d) **Proporciona el resumen del archivo a los pares que lo solicitan**.
- e) **Proporciona una lista de pares que están compartiendo el archivo.** Es decir, proporciona una lista con la información de acceso (IP o nombre de objeto remoto) de dichos pares.
- f) **Permite registrar cuando un par deja el enjambre**. De manera que el *tracker* lo elimine de la lista.

Como se puede ver, el *tracker* proporciona funcionalidad que permite la coordinación de los pares. Los pares usarán el *tracker* básicamente para obtener la información del archivo que se descarga (de esto se encargan los archivos *.torrent* en el bittorrent real), y para saber qué pares lo comparten, es decir, unirse o dejar el enjambre.

3) Los **pares (*peers*)** se conectan entre sí y descargan/suben información a los demás. Para ello deberán realizar una serie de acciones.

- **a)** **Registrarse en el *tracker*,** para unirse al enjambre, indicando si descargan o hacen de *seed* y proporcionando su información de acceso (IP o nombre de objeto remoto).
- **b)** **Informar al *tracker* cuando dejen el enjambre**. De manera que el *tracker* pueda mantener actualizada la lista de participantes en el enjambre.
- **c)** **Mantener una lista de otros participantes (vecinos o *neighbors*)**. Esta lista se obtiene inicialmente del *tracker* y se **actualiza cada vez que un par** que no está en la lista le solicita una nueva conexión.
- **d)** **Intercambio de información de bloques**: cuando un par establece una nueva conexión con otro par, se **intercambian la lista de los bloques** de que disponen. Llamaremos a este paso el **intercambio de un mensaje Hello**. **Nota:** puesto que los pares disponen del resumen del archivo (la lista de *hash* de los bloques), basta con indicar mediante un **mapa de bits** los bloques de los que se dispone, es decir, indicar con un 1 ó 0 si se tiene o no cada bloque.
- **e)** **Informan a los otros pares de los bloques que adquieren**. Cuando un par termina de descargar un bloque, **informará** a todos sus vecinos **del bloque que ha descargado**, y los vecinos **actualizarán** la lista de bloques correspondiente a dicho vecino.
- **f)** **Transferencia de bloques**. Los pares descargan y suben bloques a otros pares. La selección de los bloques que se van a descargar se realiza por medio del **Algoritmo de selección de bloques** (ver más adelante). Una vez seleccionados los bloques, el par solicitará la descarga a los pares que tienen el bloque. Sin embargo, un par **solo subirá archivos a un número fijo de pares**. Es decir, sólo permitirá que un **máximo de N pares** (será un parámetro que establecerá a 4 por defecto, pero que puede ser cambiado), se descarguen bloques de él. Este conjunto de pares a los que se les permite la descarga se denomina **conjunto de vecinos activo (*active neighbor set*).** La selección de los pares que forman el conjunto de vecinos activos se realiza por medio del **Algoritmo de selección de pares** (ver más adelante).

g) **Comprobación de la integridad del bloque**. Los clientes deben comparar el *hash* del bloque descargado con el correspondiente *hash* del bloque del resumen del archivo para asegurar que la descarga es correcta, antes de informar al resto de pares de la descarga de un bloque.

Como puede observar, los pares realizan el grueso de las acciones que permiten la descarga compartida de archivos. Una vez que los pares conocen el resumen del archivo que quieren bajar y disponen de una lista de vecinos en el enjambre, proceden al intercambio de bloques con los demás pares, ejecutando **periódicamente** los algoritmos de selección de bloques y pares.

4) **Algoritmo de selección de bloques (*piece selection algorithm*)**. Se pueden implementar distintas estrategias. Y, de hecho, en la práctica se implementan diferentes "políticas". Tendrá que implementar la política denominada **más-raro-primero (*rarest-first*)**: se trata de seleccionar el bloque menos disponible primero. Es decir, se examina el número de pares que tiene cada bloque y se seleccionan aquellos menos disponibles. Esta estrategia se complementa con otra denominada **primer-bloque-aleatorio (*random first piece*):** consiste en seleccionar los primeros bloques aleatoriamente, hasta que se disponga de al menos **X bloques (parámetro configurable**). Una vez se disponga de los primeros X bloques, se pasa a la política *rarest-first*.
5) **Algoritmo de selección de pares (*choke algorithm*).** Existen varios algoritmos usados en la práctica. Se deja a su elección la forma de selección de pares. En la práctica, la selección se hace por algún mecanismo de reciprocidad, es decir, se prioriza a los pares que comparten más (por ejemplo, tienen una tasa de subida mayor).

Los alumnos deben decidir e implementar el conjunto de clases necesarias para desarrollar la funcionalidad anterior. De hecho, los alumnos deben decidir si se implementa mediante:

- **Alternativa R:** Llamada a procedimiento remoto (RMI), mediante Java RMI.
- **Alternativa S:** Aplicación cliente servidor clásica usando la interfaz Socket de Java.

En cualquier caso, se aconseja que la funcionalidad se implemente de manera incremental, desarrollando la funcionalidad por pasos.

Ejemplo de ejecución

Para aclarar el funcionamiento de la aplicación, se describe una posible ejecución del mismo.

1. Inicialmente se pone en marcha el Tracker, a la espera de que se registre un seed. La IP/nombre del Tracker es conocida.
2. Se pone en marcha el peer1 (P1) que actúa como seed. Genera el resumen del archivo a compartir y se registra en el Tracker. En este momento el Tracker conoce que el archivo lo comparte P1.
3. Al cabo de un tiempo se pone en marcha el peer3 (P3). Se conecta con el Tracker, que le pasa el resumen, la IP/nombre del seed y la lista de pares que comparten (vacía). En este momento el Tracker conoce que el archivo lo comparte también P3 y actualiza su lista.
4. P3 se conecta con el seed (P1) con un mensaje Hello, pasándole la lista de bloques que tiene (ninguno) y recibiendo la lista de bloques de P1 (todos, ya que es seed). En este momento P1 sabe que P3 también está compartiendo, por lo que lo añade a la lista de pares.
5. P3 le solicita a P1 alguno de los bloques del archivo.
6. Se pone en marcha peer2 (P2). Se conecta con el Tracker, que le pasa el resumen, la IP/nombre del seed y la lista de pares que comparten (P3).
7. P2 se conecta tanto con el seed (P1) como con P3, pasándole la lista de bloques que tiene (ninguno) y recibiendo la lista de bloques de P1 y P3, es decir, haciendo el intercambio de mensajes Hello. En este momento tanto P1 como P2 saben que P2 también está compartiendo, por lo que lo añaden a la lista de pares.
8. P2 solicita tanto a P1 como a P3 bloques del archivo. Por su parte, P3 solicitará bloques tanto a P1 como a P2, a medida que vaya actualizando la lista de bloques disponibles. De esta forma, todos los pares descargan de los demás.

Los alumnos deben **ejecutar la aplicación en su cuenta del servidor del laboratorio** (labit601.upct.es) y **comprobar que funciona correctamente con los pares ejecutándose en un ordenador remoto** (es decir, cualquier otro PC, bien del laboratorio, su propio PC, etc.). Como parte de la evaluación del trabajo, **el profesor probará la aplicación en el laboratorio,** en su cuenta.

En los siguientes apartados se describe con más detalle la funcionalidad y se discuten posibles problemas además de ofrecer consejos para la implementación. **Lea con detenimiento las siguientes secciones antes de empezar la implementación.**

2.1 Extensiones

En el apartado anterior se ha descrito la funcionalidad *mínima* del programa. Para obtener la máxima calificación es necesario realizar alguna extensión del mismo. Puede decidir cómo extender su programa libremente, aunque aquí se le sugieren algunas posibilidades.

- La funcionalidad mínima solo considera la descarga de un único archivo. Puede extender su programa para que permita la descarga de **múltiples archivos**.
- Aunque debe probar en cualquier caso su programa para asegurarse que la funcionalidad mínima está correctamente implementada, puede considerar la realización de **pruebas sistemáticas y exposición de resultados**. Para ello, puede comparar por ejemplo si se obtiene alguna mejora respecto a la descarga directa o cómo influyen los parámetros en el rendimiento. El entorno de funcionamiento real de un sistema P2P (red extensa) no se parece al entorno en el que realizará sus pruebas (único PC o red local). Para obtener resultados coherentes necesitará seguramente emular el entorno real. Para ello, puede utilizar los programas *tc* y *netem* en Linux. Vea https://wiki.linuxfoundation.org/networking/netem.
- Implementación de diferentes **Algoritmos de selección de bloques y pares**. En particular, en bittorrent se utilizan unos algoritmos de selección de pares determinados que puede probar.
- Desarrollo de una **interfaz gráfica.**

2.2 ¿Java RMI o interfaz Sockets?

Ambas alternativas tienen sus ventajas e inconvenientes. El uso de una u otra alternativa se tratará por igual a la hora de calificar.
La ventaja de usar Java RMI es que las cuestiones relativas al transporte de datos sobre la red, es decir, a la serialización de los objetos, se realizan automáticamente. Es decir, el paso de objetos del cliente al servidor se realiza invocando funciones con los parámetros adecuados y devolviendo el tipo de datos correspondientes por parte de la función en el servidor. Además, la concurrencia se gestiona automáticamente. Es decir, una interfaz remota puede atender múltiples peticiones simultáneas sin necesidad de que el programador haga nada especial.
Las desventajas son que la configuración es más compleja y presenta problemas si se utiliza en redes de área extensa o a través de firewalls.

Por otra parte, la ventaja de usar una implementación clásica, mediante el uso de Sockets, es que no es necesaria una configuración especial y los firewalls son menos problemáticos.
Las desventajas son que es necesario definir algún tipo de interfaz entre el cliente y el servidor, declarando posiblemente tipos de mensajes que se van a intercambiar y serializar y deserializar correctamente los datos. Y, sobre todo, es necesario, implementar correctamente la concurrencia y paralelización de tareas. Ver apartado 2.5 para más información.

2.3 Preasignación de archivo

La transferencia de archivos entre pares se realiza por bloques de manera no consecutiva, es decir, recibirá los bloques de los pares de forma no ordenada. Esto requiere que se reserve espacio en disco para el fichero completo. Aunque hay otras opciones (uso de *sparse files* en los SO que lo soportan), la forma tradicional de hacerlo consiste en una reserva de espacio. Es decir, una vez se determina el tamaño del fichero completo, se escribe un fichero de ceros en disco del tamaño requerido, con un nombre temporal. Posteriormente, se va escribiendo, **mediante acceso aleatorio al fichero**, cada uno de los bloques reales a medida que se reciben. Cuando se dispone del fichero completo se renombra el archivo.

2.4 Uso de funciones *hash*.

En la aplicación se realiza un *hash* (resumen o *digest*) de los bloques del archivo al crear el resumen del archivo y cada vez que se descarga un bloque para comprobar la integridad. Los *hash* son funciones matemáticas, que se utilizan sobre todo en criptografía, que dada una cadena de bytes, generan otra cadena de longitud fija, llamada *hash*, con la propiedad de que si la cadena original cambia, al generar un nuevo *hash* este también será distinto. Hay varios algoritmos que realizan esta función como MD5 o SHA1.

Hay varias opciones:

- Java proporciona librerías para realizar *hash*. La clase MessageDigest permite realizar un *hash* de un conjunto de bytes. Para realizar el *digest* de un fichero hay que ir leyendo bloques de bytes y actualizar el digest mediante *update()*. El *hash* se genera al final cuando se invoca *digest()*. Esta función resetea el estado, con lo que solo debe invocarla una vez por bloque, cuando disponga del bloque completo.
- Alternativamente, el sistema operativo Linux proporciona programas como *md5sum* u *openssl dgst* que realizan esta función. Puede ejecutarlas desde su programa en Java y obtener el resultado. No se recomienda esta opción en esta aplicación por la sobrecarga que introduce.

2.5 Paralelización de tareas

Dadas las características y el funcionamiento de la aplicación, es imprescindible que ejecute diferentes tareas en *threads* separados. Debe decidir la implementación, pero tenga en cuenta las siguientes sugerencias:

- Un par actúa simultáneamente de cliente y de servidor. Se puede, por tanto, separar la ejecución de la funcionalidad de cliente y servidor en *threads* diferentes. Además, el servidor deberá ser concurrente, por lo que el *thread* de servidor creará a su vez nuevos *threads* para atender las peticiones entrantes. Si utiliza RMI, tenga en cuenta que parte de la concurrencia ya se gestiona automáticamente. Por ejemplo, si declara la funcionalidad del servidor como una interfaz remota, el objeto remoto automáticamente gestiona las peticiones concurrentes.
- Una de las ventajas de los *threads* es que pueden acceder a zonas de memoria común directamente. Es decir, el *thread* servidor puede actualizar las variables del *thread* cliente. Como ejemplo concreto: el *thread* servidor recibe un mensaje de un par que le informa que ha descargado determinado bloque y actualiza el bitmap correspondiente a dicho par. Este bitmap posteriormente será consultado por el *thread* cliente cuando determine los bloques que va a descargar mediante el algoritmo de selección de bloques.
- En relación a lo anterior, en ciertos casos es necesario evitar la concurrencia de escritura, es decir, que múltiples *threads* intenten modificar una misma variable. Java proporciona funciones para gestionar la concurrencia de manera relativamente sencilla.
- Tenga en cuenta que en ocasiones su programa no puede/debe hacer nada hasta que se dé una condición determinada. En estos casos lo conveniente es hacer que el *thread* vaya a dormir para evitar desperdiciar ciclos de CPU. Además, note que dispone de la clase *ScheduledThreadPoolExecutor* que permite ejecutar tareas (*threads*) de manera periódica, algo que le puede ser muy útil para controlar el flujo general de su programa.

2.6 Parametrización

En una aplicación como esta hay un gran número de opciones que deberían poder configurarse para ajustar el funcionamiento y el rendimiento de la aplicación. Permita que se pueda configurar los parámetros más relevantes del funcionamiento de los pares y el *tracker*.

2.7 Ejecución e identificación de pares

En un sistema real, los pares se ejecutan habitualmente en diferentes equipos y se identifican mediante la IP. En su caso, si solo dispone de un equipo para probar, deberá ejecutar cada par como un proceso Java diferente y, puesto que todos tienen la misma IP, deberá identificarlos mediante el par IP/puerto, o, si utiliza RMI, debe asignar un nombre distinto a cada objeto remoto que represente un par.

Uso de RMI con diferentes máquinas
El registro de **RMI no permite que un programa registre un objeto como servidor en una máquina remota**. Es decir, el registro se tiene que estar ejecutando en la misma máquina que el programa que se quiere registrar como servidor. Por tanto, los pares tienen que registrar su interfaz en su propia máquina, aunque pueden buscar los objetos remotos de otros pares en cualquier máquina. Esto implica que, si se usa RMI con diferentes máquinas, la identificación de los pares requiere el uso de IP/Nombre de par en registro siempre.
Ejemplo: para registrar su interfaz los pares siempre tienen que usar Naming.rebind(“rmi://localhost/NombrePar”);
Para buscar otros pares tienen que usar la IP de la máquina donde se está ejecutando el otro par, por ejemplo, Naming.lookup(“rmi://192.168.6.3/NombrePar”);

2.8 Desarrollo de una aplicación alternativa

Como se ha dicho, los objetivos de este trabajo son que los alumnos practiquen distintas técnicas utilizadas y en sistemas distribuidos y se enfrenten a los retos de diseño e implementación que presentan estos sistemas en la realidad.
Si algún grupo está interesado en desarrollar un sistema distribuido diferente al que se propone aquí, pueden solicitar al profesor que lo considere como trabajo. Para ello, deben enviar un correo electrónico a Esteban Egea indicando la aplicación o sistema que quieren desarrollar para que lo apruebe antes de comenzar el desarrollo.

3 Material a entregar y criterios de evaluación

Los alumnos deberán entregar una **memoria** del trabajo, además del **código** fuente de la aplicación, mediante el aula virtual.
En la memoria deberá indicar las instrucciones para poner en ejecución el programa.
Para incentivar el desarrollo progresivo del trabajo, se realizarán **entregas parciales**.

- **Primera entrega**: **creación del "Torrent".** Se solicitará la implementación de la funcionalidad 1.A) y 1.B). **(15% de la puntuación)**
- **Segunda entrega**: **implementación del Tracker**. Se solicitará la implementación de la funcionalidad 2.A) a 2.F) además de 1.C) y 1.D). **(25% de la puntuación)**
- **Entrega final**: el trabajo completo. **(60% de la puntuación)**

Las entregas parciales deben incluir una memoria de la implementación. De esta forma la memoria final también se va realizando progresivamente.

Las fechas límite de las entregas se publicarán en el Aula Virtual.
No se admitirá ningún trabajo en fechas posteriores a las fechas límite. Es posible que en algún caso los profesores necesiten reunirse con los alumnos de un grupo para evaluar su trabajo. Para esos casos **se publicará una lista de los grupos que deben entrevistarse con los profesores para explicar su trabajo.**

3.1 Memoria y Código

La memoria debe contener al menos lo siguiente:
- Nombre completo de los componentes del grupo y correo electrónico.
- Índice
- **Instrucciones para ejecutar su aplicación**. Esto es imprescindible para que el profesor pueda evaluar su trabajo.
- Descripción de la implementación.
- Lista de clases que se han desarrollado, clasificadas según si implementan la funcionalidad del cliente y la del servidor.
- Descripción de la funcionalidad de cada clase (¿qué hace?).
- Documentación de cada clase: atributos y métodos que la componen y descripción de la función que realiza cada método.
- Pruebas realizadas y resultados obtenidos.

No se limite a comentar el código. La memoria debe explicar desde un punto de vista de alto nivel, arquitectónico, la implementación de la aplicación. Debe centrarse en describir los problemas encontrados y la solución implementada. Debe describir brevemente la función de las clases desarrolladas y el bloque o entidad al que pertenece, por ejemplo, - *…el tracker se ha implementado mediante las clases X e Y. La clase X se encarga de* … No debe describir funcionalidad que no haya desarrollado el grupo, es decir, no copie y pegue la documentación de las clases de las librerías de Java que utilice. Puede incluir un diagrama de flujo de los programas principales o de las principales funciones. Puede además incluir cualquier otro diagrama (de clases, de relaciones, etc.) que considere necesario para la explicación.

3.2 Criterios de evaluación

La puntuación total del trabajo se ha repartido de la siguiente forma

Funcionalidad	Puntuación
Implementación y ejecución de la funcionalidad mínima requerida.	7
Implementación de extensiones.	2
Calidad de la memoria	1

Para la evaluación se tendrá en cuenta al menos los siguientes aspectos:

- Que se hayan implementado **correctamente** la funcionalidad mínima.
- Se valorará la claridad, rigor y concisión en las explicaciones aportadas en la memoria.
- Se valorará la claridad y orden en el código y la correcta separación en clases y métodos de la funcionalidad requerida.
- La capacidad de realizar o atender tareas concurrentemente de la aplicación. Esto en sí es parte de la propia funcionalidad que se requiere.

Se anima a los alumnos a que extiendan la funcionalidad requerida de la manera que consideren conveniente, así como, que gestionen adecuadamente los errores que se puedan producir, en particular, los específicos de los sistemas distribuidos.
Las extensiones se valorarán por su **originalidad y su calidad.**

También se valora que se evalúe el rendimiento de la aplicación, independientemente de que se realice como extensión, es decir, que realice una serie de pruebas sistemáticas para determinar el correcto funcionamiento o detectar fallos de la aplicación.

Aunque se admite que pueda haber cierta similitud en la implementación de la funcionalidad requerida, **se penalizará cualquier implementación o memoria que se determine que ha sido copiada de otros compañeros en este curso o anteriores**.

3.3 Grupos de una persona

Al inicio de esta propuesta se indica que el trabajo debe realizarse en parejas, pero en ocasiones las circunstancias impiden formar o mantener un grupo y se acaba con grupos con una sola persona. En caso de no ser posible la formación de una pareja, se tendrá en cuenta a la hora de evaluar el mayor esfuerzo realizado.

4 Notas adicionales para la implementación

4.1 Encapsulación

Tanto si realiza el programa mediante Sockets como RMI, y especialmente en este caso, procure encapsular los datos en objetos cuando sea necesario. En RMI es fundamental ya que en ocasiones querrá que una función remota devuelva más de un resultado. Por ejemplo, suponga que quiere que su función devuelva un *long* y un array de bytes, *byte[]*. Lo recomendable es crear una clase *DataChunk* por ejemplo, que incluya ambos tipos como atributos, de manera que la llamada a su función devuelva un objeto de este tipo. Recuerde además que esa clase debe implementar la interfaz *Serializable* para poder ser serializada.
En el caso de usar Sockets puede definir tipos de mensaje mediante clases, lo que facilitará el transporte de red. De esta forma puede usar los *ObjectInputStream* y *ObjectOutputStream* que le permiten serializar objetos sobre un Socket.

Procure agrupar la funcionalidad relacionada en clases. Por ejemplo, puede implementar una clase *Neighbor* que incluya toda la información de los pares: IP/puerto o nombre de objeto remoto, su mapa de bits de los bloques que tienen, si le está descargando, la tasa de descarga si fuera necesaria, etc. Además, incluya otras clases como propiedades cuando sea necesario. Por ejemplo, puede agrupar la funcionalidad del mapa de bits de bloques es un clase determinada.

4.2 Colecciones y contenedores

Necesitará utilizar listas, vectores u otro tipo de contenedores. Java proporciona una librería muy rica de lo colecciones y contenedores genéricos, que además incluyen funciones de utilidad para buscar, ordenar o comparar elementos. Se recomienda que consulte:

https://docs.oracle.com/javase/tutorial/collections/

4.3 Gestión de errores

Aunque la gestión de errores es importante en un programa real, no será un foco de atención en este trabajo a no ser que explícitamente se realice como extensión. Es decir, se admitirá que simplemente se relancen excepciones o se capturen y simplemente se muestre el error por pantalla.
En caso de desarrollar una gestión de errores como extensión, debe indicarlo y describirlo en la memoria. La gestión de errores puede incluir mecanismos como, por ejemplo, el uso de reintentos ante un error de conexión o transferencia.

5 Bibliografía

[1] Bruce Eckel, Thinking in Java, 4th Edition, Prentice Hall, 2006.
[2] http://web.cs.ucla.edu/classes/cs217/05BitTorrent.pdf
[3] https://www.kth.se/social/upload/516cfabef2765401a0704d28/6-BitTorrent.pdf