Universidad Politécnica de Cartagena

Escuela Técnica Superior de Ingeniería de Telecomunicación

PRÁCTICAS DE SISTEMAS Y SERVICIOS DISTRIBUIDOS

Propuesta del Trabajo de Prácticas 2024/2025

Servidor de transferencia de archivos

Revisión 1.1

Profesores:
Esteban Egea López

Índice.

1 Consideraciones generales.

1.1 Objetivos

En este trabajo de prácticas los alumnos realizarán por parejas una aplicación para la transferencia de archivos y directorios. Los objetivos son: primero, 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. Segundo, que los alumnos se familiaricen con la programación de aplicaciones distribuidas mediante las librerías de Java.

1.2 Introducción

La aplicación a desarrollar imitará de manera simplificada el funcionamiento de un servidor de transferencia de ficheros. Este es uno de los sistemas cliente-servidor fundamentales, que es la base de las aplicaciones que permiten el almacenamiento en la nube y la sincronización de archivos y directorios. Durante el desarrollo de un sistema de este tipo aparecen problemas típicos de los sistemas distribuidos como 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, limita algunos de estos problemas y facilita el desarrollo. En particular 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: el servidor mantendrá una carpeta raíz a la que se podrán subir archivos que luego estarán disponibles para la descarga por el mismo u otro usuario. El servidor permitirá que se creen subdirectorios sobre los que también se podrán subir o descargar archivos. También se permitirá la carga o descarga de una carpeta completa.

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 ser ejecutada y probarse.

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 mantener una copia remota de los archivos depositados en una carpeta local de manera automática. La aplicación se dividirá en dos módulos funcionales: aplicación *cliente* y aplicación *servidor*. Además, contaremos con una *carpeta raíz* que mantendrá el servidor y que contendrá los archivos y directorios que se pueden descargar.
A partir de las definiciones anteriores, la **funcionalidad mínima requerida** (consulte los criterios de evaluación en 3.2) es:

- Pueden subirse y descargarse ficheros de cualquier tipo (no solo texto) y de cualquier tamaño (incluso de varios GB).
- La aplicación cliente permitirá:
    1. **Listar contenidos de carpetas y subcarpetas**. De manera que el usuario pueda examinar el contenido del servidor de archivos, moviéndose por el árbol de directorios.

2. **Descargar o subir archivos** individuales.
3. **Crear y eliminar** carpetas y archivos.
4. **Subir o descargar carpetas completas.** En el caso de descarga, se creará la carpeta correspondiente en el cliente y se descargarán todos los ficheros que contenga esa carpeta en el servidor. O, al contrario, se creará la carpeta en el servidor y se subirán todos los archivos de la carpeta del cliente. En principio, se supondrá que una carpeta no contiene más subcarpetas. Como **funcionalidad opcional** (ver 2.5) se pide que implemente la creación de árboles de directorios completos de manera recursiva.

- La aplicación servidor por su parte, se encargará de complementar la funcionalidad del cliente para lo que:
  1. Permanecerá a la espera de peticiones del cliente.
  2. Proporcionará una lista de archivos que contiene una carpeta dada, que puede ser la raíz o una subcarpeta.
  3. Guardará los archivos que le mande el cliente.
  4. Enviará los archivos que le solicite el cliente.
  5. Eliminará o creará las carpetas o archivos que le solicite el cliente.

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

No es necesario desarrollar una interfaz gráfica para la aplicación, basta con utilizar la consola.

En cualquier caso, **se aconseja que la funcionalidad se implemente de manera incremental**, desarrollando la funcionalidad por pasos (ver 2.7).
Los alumnos deben **ejecutar la aplicación servidor** y **comprobar que funciona correctamente con el cliente 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 cliente** en su propio PC frente a **la aplicación servidor 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 ¿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.
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. Y, en particular, **tanto la descarga como la subida de ficheros deben hacerse en bloques**, mediante sucesivas llamadas. Tanto el cliente como el servidor deben ir guardando el fichero cuando reciban un nuevo bloque de bytes a partir de la última posición guardada. Ver apartado 2.2 para más información.

Por otra parte, la ventaja de usar una implementación clásica, mediante el uso de Sockets es que la descarga y subida de ficheros se implementa de manera más natural, enviando a medida que se lee. Además, 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. Ver apartado 2.2 para más información.

2.2 Subida y descarga de ficheros

Los **archivos** en general, y sobre todo si son de gran tamaño, **se transfieren en bloques de bytes**. Tanto si utiliza RMI como Sockets no es una buena idea cargar un archivo de golpe en memoria, ya que si es muy grande, agotará la memoria disponible.
Mediante sockets es relativamente simple la transferencia: se lee un bloque de bytes de disco y se escribe al socket correspondiente, y así sucesivamente. No es necesario indicar la posición del bloque ya que el protocolo de transporte se encarga de ordenar los mensajes a la llegada.
Mediante RMI igualmente se lee un bloque de disco, pero se invoca una función para guardar/descargar el bloque remotamente. Puesto que estas invocaciones son independientes, en cada llamada se debe indicar la posición en el archivo dónde hay que empezar a escribir los datos del fichero. Además, es necesario indicar cuándo ha terminado la transferencia del archivo, es decir, no hay más bloques para transmitir. Esto implica que, dependiendo de la implementación, es posible que sea necesario invocar funciones antes o después de comenzar la transferencia para inicializar o finalizar las acciones necesarias.

2.3 Manejo de ficheros y directorios: Java NIO.

Dadas las características de la aplicación es obvio que deberá leer directorios, leer y escribir ficheros, etc. Puede utilizar la funcionalidad clásica de entrada y salida de Java (java.io). Sin embargo, las últimas versiones de Java incluyen las nuevas librerías para el manejo de ficheros (java.nio) que son más eficientes y más simples en algunos casos.
Se recomienda que lea con detenimiento el siguiente tutorial y examine los ejemplos relacionados con este trabajo:
https://docs.oracle.com/javase/tutorial/essential/io/fileio.html

- En particular, tradicionalmente los ficheros en Java se han descrito mediante la clase File, que es independiente de la plataforma, lo que hacía de sufrieran limitaciones. Ahora se introduce la clase Path, que es dependiente de la plataforma, pero mucho más flexible.
- Observe que, por este motivo, Path no es serializable, mientras que File sí que lo es.
- Recuerde que tiene funciones para convertir de Path a File.
- Observe que dispone de funciones en Files (no confundir con File) para leer un directorio, crear carpetas, etc.

2.4 Uso de hilos (threads).

En aplicaciones de este tipo, es muy recomendable que las operaciones de transferencia se hagan en hilos separados. De esta manera, la descarga se realiza en paralelo y la aplicación sigue respondiendo a los comandos del usuario. Es particularmente útil para la descarga/subida masiva (carpetas o múltiples ficheros).
Los threads deben utilizarse si se implementa con Sockets para que el servidor sea concurrente, pero pueden utilizarse adicionalmente al atender una petición: un thread que está atendiendo una petición puede ejecutar nuevos threads para realizar las cargas o descargas.
Así mismo, los threads pueden usarse con programas RMI: un cliente RMI puede ejecutar un thread separado para invocar funciones de carga o descarga del servidor.
En cualquier caso, **los threads deben implementarse con ExecutorService.**

2.5 Carga/descarga de árboles de directorios (Opcional)

Cuando se carga o descarga una carpeta completa, ésta puede contener otras subcarpetas. Como funcionalidad adicional se pide que se carguen/descarguen las subcarpetas que pueda contener una carpeta. Esta funcionalidad puede implementarse de manera más sencilla usando recursión.

2.6 Monitorización de carpeta (Opcional)

El paso de una transferencia de ficheros a una aplicación de sincronización consiste en que el cliente monitorice el estado de una carpeta local continuamente, de manera que, cuando se modifica, borra o añade un archivo, se ejecuta una acción automáticamente. Con Java NIO es relativamente sencillo monitorizar un directorio. Se le pide que extienda la aplicación para que monitorice continuamente una carpeta local y actualice la carpeta en el servidor según los cambios que se produzcan.

2.7 Implementación incremental

Se recomienda que los alumnos implementen la funcionalidad de manera incremental, añadiendo funcionalidad por bloques y probándola antes de pasar al siguiente bloque. A continuación, se describe una posible secuencia de implementación:

- **Funcionalidad relacionada con el manejo de archivos**. Desarrollar clases que se encarguen de leer los archivos en un directorio y los directorios que contiene. Crear y eliminar carpetas. Comprobar su correcto funcionamiento. Esta funcionalidad es independiente del desarrollo del cliente y servidor, por tanto, se puede comenzar a desarrollar una vez se ha impartido la práctica de entrada/salida.
- **Desarrollo de cliente y servidor.** Una vez se ha decidido si la implementación ser realiza mediante Sockets o Java RMI, se procede a desarrollar el cliente y servidor. De nuevo, se implementa la funcionalidad paso a paso, por ejemplo, descarga de un archivo simple, probándola antes de continuar añadiendo nueva funcionalidad. Finalmente, se implementa el envío de los archivos en sí.
- **Pruebas**. Se realizan pruebas para comprobar el funcionamiento global de la aplicación y su rendimiento. Se debe comprobar si la aplicación funciona correctamente cuando se ejecutan cliente y servidor en PCs diferentes.
- **Implementación de funcionalidad adicional y extensiones**. Una vez se dispone de la aplicación básica funcionando correctamente, se implementa la funcionalidad adicional.

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.

Se avisará con suficiente antelación de la fecha límite de entrega.
No se admitirá ningún trabajo en fechas posteriores a la fecha 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 servidor 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.
El código se adjuntará comprimido en un ZIP mediante el correspondiente entregable en el aula virtual.
El código **debe poder ser ejecutado y funcionar en el servidor** y se debe indicar en la memoria como se ejecutan tanto la aplicación cliente y la aplicación servidor, incluidos todos los parámetros necesarios. 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.	6.5
Opcional (monitorización de carpeta y carga/descarga de árboles)	1.5
Extensiones, pruebas y diseño de la aplicación	1
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.

También se valora que se evalúe el rendimiento de la aplicación, 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**.

3.2 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, tal y como se hace en la práctica 4. De esta forma puede usar los *ObjectInputStream* y *ObjectOutputStream* que le permiten serializar objetos sobre un Socket.

4.2 Colecciones y contenedores

Posiblemente 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/

En particular, recuerde que dispone de *ArrayList* y, cuando trabaja con ficheros, que tienen un nombre único, puede ser muy útil utilizar tablas *hash* que asocian un elemento del tipo que sea a una clave única de otro tipo. Dispone de *HashMap* o *HashSet* por ejemplo.

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.