Trabajo de prácticas

Sistemas y servicios Distribuidos

Curso 2015/2016

Aplicación para el almacenamiento automático remoto de archivos.

Alumnos:

Antonio José Bermejo Giménez – abermejog94@gmail.com
Jairo Peña Iglesias – jairopi91@gmail.com

Profesores:

Esteban Egea López
M. Dolores Cano Baños

1. Índice.

2. Introducción:

En este proyecto hemos desarrollado una aplicación para la sincronización de directorios remotos, con una funcionalidad “similar” a la de servicios ya conocidos como Dropbox o Google Drive.

Esta aplicación se fundamenta en la arquitectura Cliente – Servidor cuya implementación se describe con detalle en el apartado 3.

3. Implementación:

El desarrollo de este proyecto se ha realizado de forma incremental, es decir, implementando la funcionalidad según requisitos. Lo que ha hecho que en numerosas ocasiones hayamos tenido que modificar la forma de ejecución de alguna parte ya implementada anteriormente cuando nos hemos encontrado con algún problema.

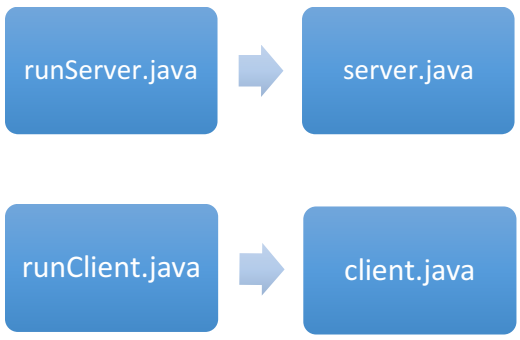
3.0. Ejecución:

Como se ha comentado anteriormente, nos basamos en la **arquitectura Cliente – Servidor**, es por ello que las dos clases principales de nuestra aplicación son:

- Client.java
- Server.java

En un principio nos encontramos con el problema de que necesitábamos una funcionalidad para “arrancar nuestro programa”, como solución hemos implementado dos “Launchers”:

- runServer.java
- runClient.java



Al ejecutar estas clases se pone en funcionamiento tanto nuestro servidor como los múltiples clientes que se conectan a el. Por supuesto, primero debemos ejecutar el Launcher del Servidor para que el cliente tenga un servidor al que conectarse, en caso contrario nos lanzará una excepción con un mensaje describiendo el problema.

La clase runServer se encarga de crear una nueva instancia de Server, atendiendo peticiones en el puerto 5000 (se pasa como parámetro).
En el lado cliente, runClient se encarga de solicitar el nombre de usuario (necesario para la extensión de la aplicación solicitada – Cap.3) a la vez que da la opción de modificar los directorios donde se sitúan las carpetas a sincronizar, tanto en local (Cliente) como en remoto (Servidor). También existe la opción de dejar que se ejecute tomando una ruta por defecto, tan solo pulsando "Enter".

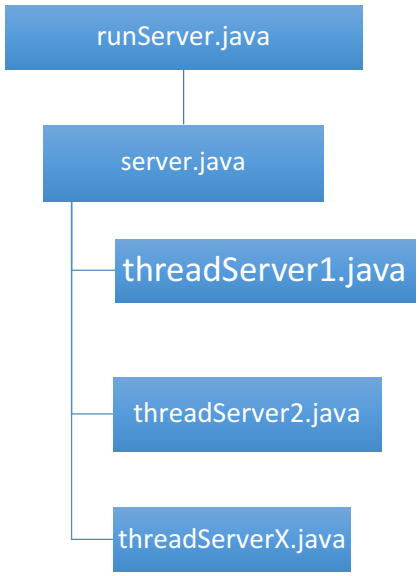
Una vez completado lo anterior, crea una instancia del cliente con los datos establecidos y la ejecuta.

3.1. Servidor (Server.java):

La clase **Server.java** es la encargada de atender las peticiones de los clientes, para ello continuamente espera la llegada de una petición, una vez que un cliente se ha conectado se ejecuta un hilo **threadServer.java**.

```
while (true) {
    accept a connection;
    create a thread to deal with the client;
}
```

El motivo de atender a un cliente en un hilo separado, es para solucionar el problema de concurrencia que se nos presenta, ya que ,es posible, que múltiples clientes se conecten al servidor simultáneamente.



3.1.1. Hilo servidor (threadServer.java):

La funcionalidad del servidor se ejecuta en el hilo **threadServer.java** al que se le pasa como argumento tanto el socket donde se encuentra el cliente como como la ruta dentro del servidor donde se encuentra el directorio a sincronizar.

Cuando este hilo se ejecuta, se llama al método inicializador:

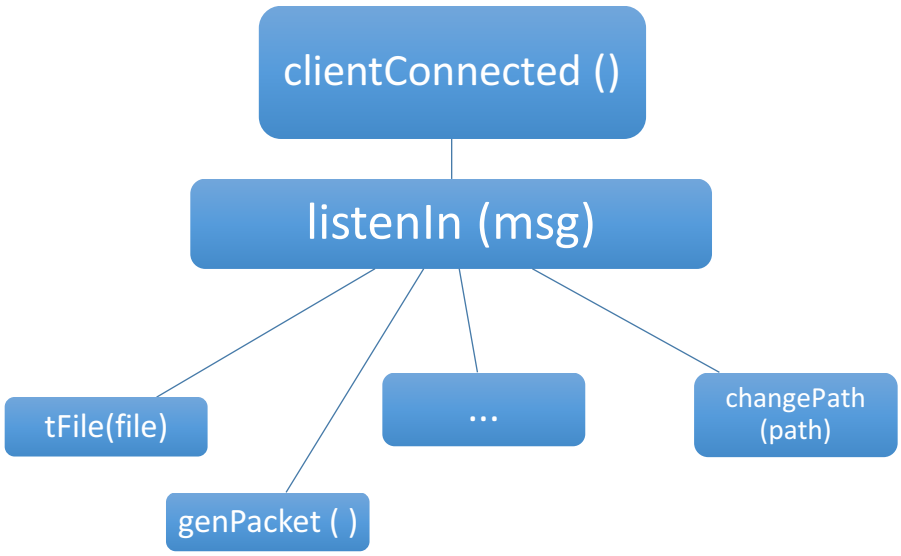
- clientConnected()

Este es el encargado de establecer la entrada/salida con el cliente y demás pasos previos. Una vez hecho esto se pone a la espera de recibir un mensaje por parte del cliente, este mensaje puede contener argumentos. Cuando se recibe se invoca al método

- listenIn(msg)

Este método se ocupa de analizar el mensaje recibido y ejecutar la acción que indique. A continuación se indica para cada mensaje recibido, la acción (método) a ejecutar:

MENSAJE	FUNCIÓN
serverList:	genPacket()
changePath:*Ruta*	changePath(Ruta)
serverTime:	Devuelve la hora del servidor
fileTime:*Archivo*	tFile(Archivo)
uploadFile:*Archivo*	Upload (Archivo)
downloadFile:*Archivo*	Download (Archivo)



- genPacket()

Este método proporciona una extensión a la implementación que se propone, al comenzar este proyecto tan solo enviábamos el nombre de los archivos que hay en el Servidor, pero posteriormente nos dimos cuenta que seria más eficiente enviar pares Archivo:Hash reduciendo así la carga en el servidor al no tener que atender numerosas peticiones para calcular el código Hash de cada archivo.
Por lo tanto, este método devuelve un conjunto de pares de nombres de archivo con su código Hash (función calcularHASH(Archivo).
El servidor, envía por la salida (*ObjectOuputStream*) estos paquetes.

- calcularHASH(Archivo)

Es la función encargada de calcular el HASH del Archivo que se le pasa como parámetro, utilizando para ello el método *digest( )* propio de *Java IO.*

- changePath()

Cuando se llama a este método se establece la ruta donde se encuentra el directorio sobre el que trabaja este cliente (lo establecemos en el *launcher*), lo que además ayuda a la implementación de la extensión propuesta (Cap. 3)

- tFile(Archivo)

Usamos esta función para conocer la hora de la última modificación del Archivo pasado como parámetro.

- download(Archivo)

Este método es el encargado de descargar los archivos que el proceso cliente solicita al servidor, cabe destacar que en nuestra implementación, para la transferencia de archivos, usamos *socketChannel* situados en el puerto 50001. Una vez conectado, se procede al envío del archivo en concreto.

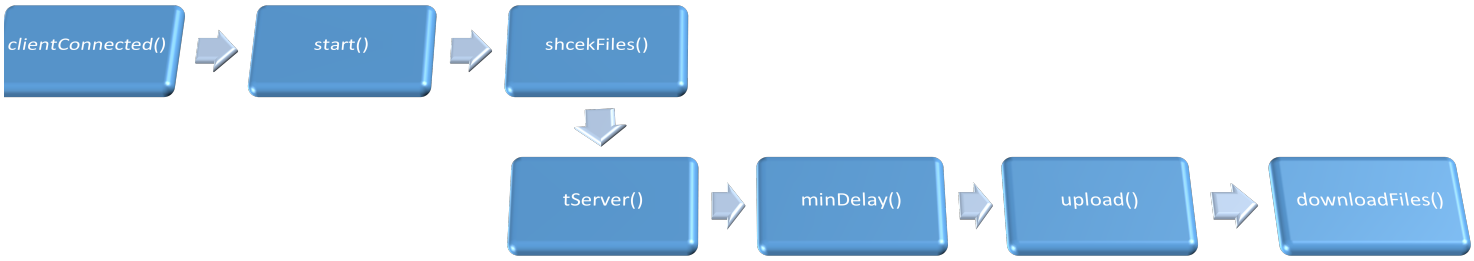
- upload(Archivo)

Complementario al anterior, su función es guardar los archivos que el cliente le envía (Otra vez a través del puerto 50001)

3.2. Cliente (client.java):

Esta es la parte de la implementación que más compleja nos ha resultado.
Cuando instanciamos un nuevo objeto de esta clase con el launcher, o dicho de otra forma, ponemos en ejecución un nuevo cliente, este establece el directorio LOCAL a sincronizar, y hace un listado de todos los archivos que hay en el. Y al igual que en el servidor, para que nos resulte más cómodo a la hora de comprobar si un archivo a sido modificado, hacemos un *HashMap* de Archivo:HASH. Para calcular el hash, al igual que en el servidor, usamos la función *calcularHash( )*.

A continuación procedemos a establecer la conexión con el socket del hilo servidor que nos atiende. SI esta conexión es correcta ejecutamos el método: clientConnected() y comienza el flujo de ejecución del programa.



- clientConnected()

Este método se encarga de establecer los flujos de entrada/salida de la parte cliente. En este punto también enviamos la instrucción *changePath:* al servidor, para establecer el directorio de trabajo que el usuario a designado. Lo siguiente es llamar al método:

- start()

En esta función enviamos el mensaje *serverList:* a través del socket con la intención de obtener el listado de archivos en nuestra carpeta del servidor acompañad del hash de estos archivos.

Hemos optado por hacer que lo siguiente sea mostrar por pantalla el contenido de las carpetas Local y Remota, una vez establecido esto llamamos a:

- checkFiles()

Este es el encargado de comprobar los archivos del cliente y del servidor, comprobar que archivos faltan, cuales se han añadido y cuales han sido modificados.

Para conocer cuales de estos archivos han sido modificados comparamos los códigos HASH del Archivo tanto en cliente con servidor, y nos quedamos con el que sea más reciente.

Es en este punto donde ejecutamos el **Algoritmo de Cristian** para sincronizar, para ello pedimos la hora del cliente y la del servidor con con:

- tServer()

Mandamos un mensaje al servidor y este nos responde con su hora.

También necesitamos conocer el retardo de red:

- minDelay()

Este método calcula el minimo RTT (Round Trip Time) es decir, el tiempo mínimo que tarda un mensaje en ir al servidor y volver. En nuestro caso hemos calculado el retardo de la petición tServer 10 veces.

Con esto conseguimos conocer la diferencia temporal entre cliente y servidor.

A partir de estos datos se crean dos listas: una con los archivos a descargar y otra con los archivos a subir al servidor, las cuales mostramos por pantalla.

Lo siguiente será subir/descargar los archivos según la lista en que se encuentren.

- upload(lista de subida)

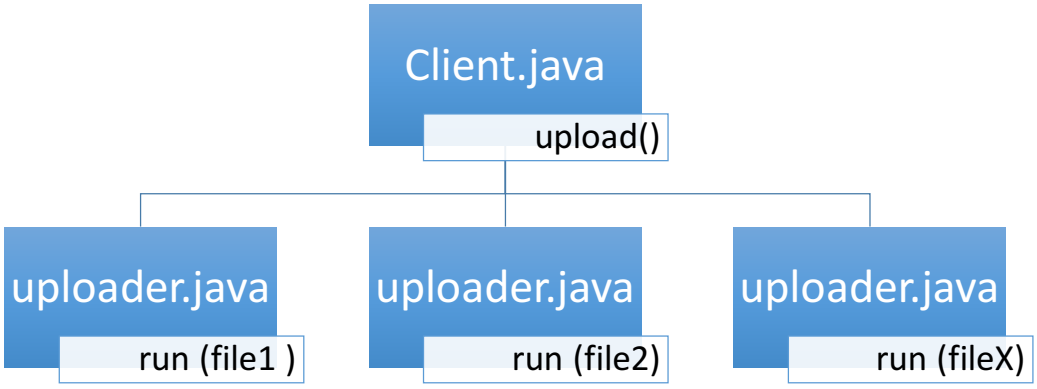
Tal y como se pide en la especificación es necesario que tanto la subida como desde la bajada de archivos en el cliente se realice en paralelo, para ello hemos desarrollado las cases uploader y downloader que implementan los hilos a ejecutar.

Esta implementación también se podría haber realizado en el servidor y aumentar su eficiencia, pero por simplificar lo dejamos así.

Este método se encarga de crear un hilo “Uploader” por cada archivo que hay que subir al servidor, una vez hecha la lista de hilos, se ejecutan todos a la vez para que la subida sea en parelelo.

3.2.1. Hilo de subida (Uploader.java)

Estos hilos son los encargados de, en primer lugar, mandar la instrucción “uploadFile:” al servidor, el cual se pondrá a la espera de recibir el archivo.
Tal y como se ha dicho anteriormente, para la transferencia de archivos usamos el puerto 50001, por lo que creamos un socket con ese puerto.
Utilizamos el SocketChannel de *Java NIO* para transferir el archivo.



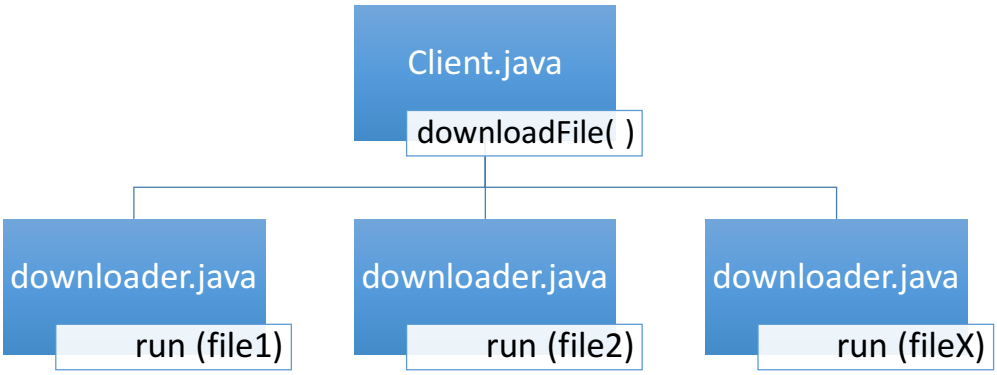
Una vez se han subido todos los archivos se procede a la descarga de los archivos que se solicitan al servidor:

- downloadFiles(lista de bajada)

Al igual que en el caso de la subida, es necesario conseguir que los archivos se descarguen en paralelo por lo que creamos tantos hilos Downloader como archivos a descargar tengamos.

3.2.2. Hilo de bajada (downloader.java)

Estos hilos son los encargados de descargar los archivos que hemos solicitado al servidor hasta nuestra carpeta local, para ello igual que en la subida, hacemos uso de los SoketChannels, cada vez que un archivo se termina de guardar, el hilo finaliza enviando un mensaje de confirmación.



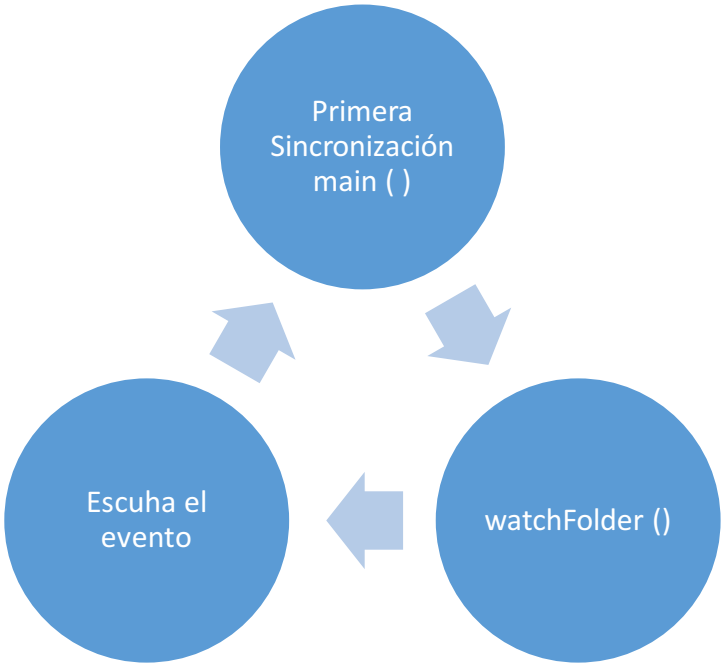
4. Opcional:

La parte opcional propuesta consiste en dotar a nuestra aplicación con la capacidad de monitorizar constantemente la carpeta del cliente y en el caso de que ocurra algún evento (Creación, borrado o modificación) actualizar las carpetas cliente y servidor.

Esta funcionalidad se ejecuta cuando el proceso de primera sincronización de archivos finaliza, llamando al método:

- watchFolder()

En un principio, nuestra idea era la de ejecutar el programa cliente cada cierto intervalo de tiempo para comprobar si algo había cambiado, pero dedicando un poco de tiempo a investigar en internet, descubrimos que la monitorización de un directorio en las ultimas versiones de Java se había vuelto algo sencillo gracias a la API **WatchService.**



Gracias a esto es posible estar al tanto de cualquier alteración que se realice en un directorio después de su ejecución.

Nuestra implementación es simple, cada vez que este servicio detecta un cambio (Creación, modificación, eliminación) vuelve a ejecutar el método principal del cliente, actualizándose de esta forma todo el sistema.

5. Extensión de la aplicación:

Aparte de la función de mandar la lista de ficheros en el servidor junto con sus hash para evitar tener que procesar demasiadas peticiones descrito anteriormente.
Llegamos a una conclusión durante el desarrollo de la aplicación, y es que había un problema que abordar: ¿Cómo distinguimos dentro del servidor la carpeta que corresponde a cada cliente? La solución más efectiva que se nos ocurrió es la siguiente:

Cuando un cliente en su ordenador lanza su ejecutable clientRun.java, se le solicita que introduzca su nombre de usuario.

La ruta que se le pasa al servidor para comenzar la ejecución incluye al final "_NOMBRE" (tanto a la default como a una nueva definida) de esta forma el servidor solo accede a la carpeta de ese usuario, y si no existe la crea. Todo esto de forma transparente para el cliente.

Esta función carece de seguridad alguna, cualquiera podría poner el nombre de usuario que quisiera y descargar los archivos del usuario suplantado.
La solución consistiría en conseguir seguridad extra creando una base de datos SQL con usuarios/contraseñas y realizando la consulta a esta. De forma que al iniciar el servicio tuviéramos que introducir la contraseña y así autenticar al propietario.

6. Test aplicación.

A lo largo de todo el proceso de desarrollo que hemos seguido hemos ido realizando pruebas del funcionamiento de la aplicación, para conseguir depurarlo correctamente hemos ido introduciendo funciones de escritura de texto en consola cada vez que el evento que nos interesada ocurría. Finalmente hemos puesto a prueba a la funcionalidad general, solucionando multitud de problemas que nos han ido surgiendo. Algunas de ellas han sido:

- Ejecutar servidor y cliente en el mismo ordenador, carpetas default, carpetas determinadas, archivos diferentes. Otras veces con los archivos iguales para que no hiciera nada y otras con una mezcla de ambos.
- Hemos probado con archivos de varios formatos (.txt, .pdf, . exe, etc…) para comprobar que el intercambio se hiciera correctamente independientemente del tamaño.
- Verificamos que la función para monitorizar la carpeta local se ejecuta correctamente (aunque con un poco de retraso algunas veces).
- Hemos intentado hacer funcionar el servidor y el cliente en dos maquinas diferentes.