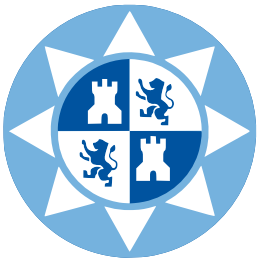

Universidad Politécnica de Cartagena

Universidad Politécnica de Cartagena
Ingeniería Telemática

Conexión Cliente-Servidor mediante sockets en Java

Trabajo de prácticas de sistemas distribuidos

Autores
Álvaro Herández Riquelme
y André Yermak Naumenko

30 de enero de 2025

Índice general

Capítulo 1

Introducción

En este documento se presenta el trabajo realizado en la asignatura de sistemas distribuidos, en el cual se ha implementado un sistema de transferencia de archivos entre un cliente y un servidor mediante sockets en Java. Se ha elegido el uso de sockets para la comunicación entre el cliente y el servidor, ya que lo consideramos una forma más sencilla y eficiente de implementar lo que se pide en estre trabajo.

1.1. Estructura del proyecto

La estructura del proyecto se basa mayoritariamente alrededor de **la máquina de estados** que hemos diseñado para éste, siendo de gran importancia el contexto que tenga el programa en todo momento, ya sea cliente, servidor, o sistema. Con esta implementación, se ha conseguido que en la misma ejecución del programa se pueda pasar de cliente a servidor en un determinado momento si es que el usuario lo desea.

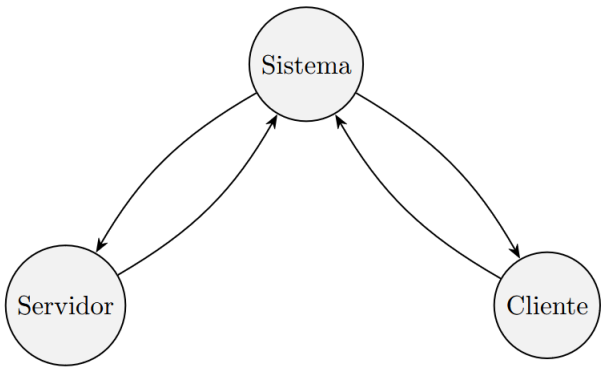

Figura 1.1. Máquina de estados diseñada para el proyecto.

Como podemos apreciar en la figura 1.1, tras arrancar el programa estamos en el contexto de Sistema, éste puede pasar de un estado a otro en cualquier momento, sin permitir que el cliente pase a servidor de forma directa ya que esto conllevaría a graves errores. Por otro lado, la estructura de archivos finalmente quedará organizada de la siguiente manera:

```
├── FileSystem.java
├── Main.java
├── SystemContextHandler.java
├── client
│   ├── ClientContextHandler.java
│   ├── ClientMain.java
│   ├── ClientUtils.java
│   └── SimpleClient.java
├── common
│   ├── CommandMessage.java
│   ├── ConsoleGUI.java
│   ├── Const.java
│   ├── Context.java
│   ├── ContextCommandHandler.java
│   ├── ContextManager.java
│   ├── ContextObserver.java
│   ├── Header.java
│   └── Utils.java
├── META-INF
│   └── MANIFEST.MF
└── server
    ├── ConcurrentServer.java
    ├── SimpleServer.java
    ├── ServerCommandProcess.java
    ├── ServerContextHandler.java
    └── ServerMain.java
```

Cuadro 1.1. Estructura de archivos del proyecto.

Capítulo 2

Guía de uso

En el comprimido se incluirá un archivo .jar y además el código fuente del proyecto. Se puede ejecutar de las dos maneras.

Para poder ejecutar el programa hay que asegurarse de tener la versión de Java 21 instalada y es recomendable para probar el funcionamiento el tener 2 archivos funcionando en carpetas diferentes, ya que el programa requiere de tener un directorio `servidor` independiente al del `cliente`. Se debería hacer una copia del .jar para el cliente, y en otra carpeta distinta, situar el .jar del servidor.

Para ejecutar el programa desde el ejecutable .jar, bastará con darle doble click o si se desea, desde la consola se puede introducir el siguiente comando:

```
»java -jar filetransfer.jar –nogui
```

Siendo –nogui un argumento opcional. Al ejecutar el programa por primera vez, se creará una carpeta FileSystem y en ella contiene otra "storage". Esta carpetas son los directorios raíces remotos y locales respectivamente.

Si se quiere arrancar un cliente basta con introducir –client en la entrada de la consola y para el servidor –server. Comandos disponibles por parte del cliente:

- **ls** [**-l (optional)**]: Listar archivos del servidor.
- **scp -u / -d** [**file**]: Subir o un archivo al servidor o descargarlo del directorio del servidor.
- **mkdir** [**directory**]: Crear una carpeta en el servidor.
- **pwd**: Mostrar el directorio actual del servidor.
- **rm** [**file**] [**-r (optional)**]: Eliminar un archivo del servidor.
- **rn <current><new>**: renombrar un archivo del servidor.
- **cd** [**directory**]: Cambiar de directorio en el servidor.
- **checkCon** [**message (optional)**]**:** Comprobar la conexión con el servidor.
- **echo** [**message**]: Enviar un mensaje al servidor.
- –**close**: Salir del programa.

- **help / ?**: Mostrar la ayuda.

Por último, si se desea usar otros valores por defecto, el archivo config.yml permite que se pueda inicializar el puerto, número máximo de conexiones y el host remoto, de tal forma que no haga falta introducirla manualmente por cada ejecución.

- **port**: Puerto del servidor.(Int)
- **max-connections**: Número máximo de conexiones que el servidor puede aceptar.(Int)
- **client-default-server-ip**: Dirección IP a la que el cliente se conectará.(String)

Si se desea, hay un servidor de prueba escuchando peticiones en la dirección IP: 84.126.4.212:4999 para poder comprobar el funcionamiento remoto.

Capítulo 3

Filetransfer

3.1. Main.java

Main inicializa FileSystem.

3.2. FileSystem.java

FileSystem es la encargada de inicializarlo todo, la interfaz gráfica (si es que el usuario no ha puesto como argumento de entrada `-nogui` o le ha dado doble click al .jar), las carpetas de gestión de archivos y el config.yml, que permite al usuario dejar argumentos de carga preestablecidos, de tal forma que no tenga que escribirlos cada vez que quiera iniciar el programa, los argumentos que se pueden guardar son: Puerto de servidor, número máximo de conexiones que un servidor puede aceptar y dirección IP a la que el cliente se conectará. Y por último inicializa la lectura por consola.

3.3. SystemContextHandler.java

SystemContextHandler es el primer contexto que instancia el usuario al arrancar el programa. Para ver los comandos disponibles hay que escribir en consola alguno de los siguientes comandos: `?`, `-help` o `-h`, que devolverá por pantalla los comandos disponibles que son en el contexto del sistema `-client` y `-server` con sus argumentos opcionales.

Capítulo 4

common

La carpeta common es la que contiene las clases que son compartidas por los 3 contextos. En ella podemos encontrar clases como los serializadores de objetos que son, el Header, CloseMessage,TextMessage y CommandMessage. También podemos encontrar las interfaces y constantes para gestionar los estados del programa entre otros.

4.1. CommandMessage.java

Es la clase encargada de crear todos los comandos y que además puedan esos comandos puedan construirse con argumentos y payload en el objeto desde la subclase Builder Comandos disponibles:

- `FILE_UPLOAD`
- `FILE_DOWNLOAD`
- `FILE_RENAME`
- `DIRECTORY_CREATE`
- `DIRECTORY_LIST`
- `DIRECTORY_LOCATION`
- `FILE_DELETE`
- `DIRECTORY_OPEN`
- `ECHO_FILE`
- `CON_CHECK`

4.2. ConsoleGUI.java

Se ha creado una interfaz gráfica opcional (si el usuario al arrancar el programa desde la consola con el argumento –nogui no se iniciará la consola) se encarga de redirigir simplemente todos los mensajes en consola al Textarea, la implementación ha sido inspirada y adaptada de Stackoverflow. [1]

4.3. Const.java

En FileSystem, hay muchos valores que son constantes y se recurren muy amenudo, por lo que se ha decidido crear una clase estática que contenga todas las constantes que se usan en el programa.

4.4. Context.java

Es un enum que contiene los estados posibles del programa.

4.5. ContextCommandHandler.java

Una interfaz para todos los contextos que manejan comandos.

4.6. ContextManager.java

Es la clase encargada de gestionar los contextos, y poder alternar entre ellos sin problemas.

4.7. ContextObserver.java

Una interfaz necesaria principalmente para la consola gráfica la cual necesita ser notificada de los cambios de contexto.

4.8. Header.java

Es la clase que le da cabecera a todos los objetos serializables, en nuestro caso se ha quedado como vestigio por usar serialización automática.

4.9. Utils.java

Esta clase estaba destinada a tener métodos útiles, pero finalmente se ha empleado simplemente para obtener las direcciones IP tanto pública como privada del servidor.

Capítulo 5

Client

5.1. ClientMain.java

El **ClientMain** es la primera instancia que se crea al hacer el cliente. En el momento que en la consola se elige el contexto con –client, se crea una instancia principal que gestiona el puerto y el host que se le envía, e intenta iniciar una nueva conexión. En caso de que exista un servidor escuchando en el host y puerto seleccionado, se creará un **socket**, una clase de **SimpleClient** que hará las operaciones del cliente, y se le pasará esta instancia de **SimpleClient** al ContextHandler, que hemos visto que manejará el puente entre los **tres contextos posibles.** Como se ha comentado, las operaciones principales pasan a ser parte del **simpleClient**

5.2. SimpleClient.java

Al iniciar un SimpleClient, hay operaciones iniciales que se deben hacer para mantener un contexto y un estado del programa. Una vez creada la instancia, se ejecuta la función **setState** que cambia el estado al enum de estado de conexión **CONNECTED** si la conexión ha sido correcta. Esto ayudará a comunicarse con el servidor y que se pueda saber su estado de conexión. A su vez, al inicio igualmente, se comprueba si la carpeta **Storage** está creada, ya que será la raíz principal de los directorios a compartir o descargar. Se llama a la función **EnsureDirectoryStructure** y si no está creada la creará mediante la clase Files del paquete java.nio.

Una vez se ha inicializado y creados los sockets, se procede al recibimiento y envío de objetos. Para ello, Se tienen un **ObjectOutputStream y ObjectInputStream** que vienen de sus respectivos streams, que servirán para el envío y recepción de objetos.

Para el recibimiento de mensajes, se maneja con dos métodos: **recieveMessages()** y **handleCommandResponse()**. Principalmente en el bucle, se llama desde un hilo a **recieveMessages**, cuya finalidad será con el objeto **ObjectInputStream** creado, se llama a su método **readObject**, que nos devolverá el objeto cuando se nos envíe desde el otro extremo, que en este caso será el servidor.

De forma similar a la que hacemos en clase, al recibir el objeto se hará un **casting** que compara de qué tipo es el objeto que previamente ha enviado el servidor. Se comparará si es un objeto de tipo **CommandMessage** o un **String**. No se contempla un tipo distinto

de objeto. El **String** vendrá de parte del servidor si éste nos quiere enviar un mensaje, por lo que simplemente imprimirá por pantalla lo que se recibe.

En caso de que se haya recibido un **CommandMessage**, se llamará a **handleCommandResponse**, como el servidor sabemos que sólo nos enviará un commandmessage en caso de que sea un archivo, se desempaquetará el payload de bytes que está en el comando y se escribirá en la carpeta seleccionada. No se contempla que el servidor use el CommandMessage de otra forma, pues para eso se usarán Strings.

5.3. ClientContextHandler.java

El clientContextHandler es el encargado de manejar los comandos que enviará el cliente, y de enviarlos al servidor. Contiene todos los posibles comandos que puede enviar el usuario para interactuar con la aplicación. No serán sólo los de conexión, sino los de ayuda y cerrar.

Según el comando que se reciba, tendrá una validación de argumentos para garantizar el correcto funcionamiento y evitar posibles errores antes de llegar al servidor. Cada posible comando tiene su propia función validadora. Por ejemplo: el comando de listar archivos **ls** tendrá su función validadora **validateLsArgs**, que comprobará tanto la cantidad de argumentos que se le pasa, como el contenido que incluye.

Si el comando pasa la validación, se construirá el objeto CommandMessage correspondiente gracias a su función **builder**. Será el objeto que se envíe finalmente al servidor.

5.4. ClientUtils.java

ClientUtils contiene todas las validaciones que se comentan anteriormente. No será una clase que se cree con su constructor y atributos, sino que se utilizarán las funciones de validación que contiene.

Capítulo 6

Server

6.1. ServerMain.java

Al inicio del contexto en el servidor, se crea una instancia de ServerMain, que actúa como el punto de entrada principal para iniciar un servidor de transferencia de archivos. La clase extiende ContextManager, por lo que hereda funcionalidades relacionadas con la gestión del contexto. Al instanciarse, ServerMain recibe un puerto como parámetro y utiliza métodos de la clase Utils (getPublicIP() y getPrivateIP()) para obtener tanto la dirección IP pública como la privada del servidor. Estas direcciones IP se almacenan en variables de instancia y se muestran por consola cuando el servidor se inicia.

El método start:

Es el encargado de iniciar el servidor. Primero, imprime un mensaje indicando que el servidor ha comenzado a escuchar en el puerto especificado, junto con las direcciones IP pública y privada. Luego, crea una instancia de ConcurrentServer, pasando el puerto y la propia instancia de ServerMain (que actúa como ContextManager) como argumentos. Finalmente, llama al método run() de ConcurrentServer, lo que inicia el servidor concurrente y comienza a aceptar conexiones de clientes. Este diseño permite que el servidor maneje múltiples conexiones de manera eficiente, utilizando un pool de hilos para gestionar cada cliente de forma independiente.

6.2. ConcurrentServer.java

El concurrentserver, cuyo nombre viene dado por las prácticas aunque se haya modificado, manejará múltiples conexiones de clientes utilizando un pool de hilos. Cuando el servidor se inicia, escucha en un puerto específico y declara un `ExecutorService` con un número fijo de hilos definido por `Const.MAX_THREADS`. El servidor entra en un bucle infinito donde espera conexiones de clientes. Cada vez que un cliente se conecta, se crea obtiene un `Socket` que representa la conexión con ese cliente y muestra la dirección del cliente conectado. Para manejar la conexión, el servidor crea un nuevo hilo en el pool, donde se instancia un objeto `SimpleServer` pasándole el socket del cliente y se llama al método 'run()'. Este método es el gestionará la comunicación con el cliente. De esta forma el servidor atiende a varios clientes de manera simultánea mediante los hilos.

6.3. SimpleServer.java

Una vez hecho el hilo y formado el socket de conexión, se necesita una forma de comunicación entre los procesos. Cada conexión del servidor con un cliente distinto tendrá una instancia independiente de **SimpleServer**, donde se implementa el manejo de objetos tanto de **input** como de **output**.

Recepción del mensaje serializado

Al iniciar la instancia de SimpleServer, también se creará un **ObjectInputStream**, leído del **InputStream**, que deserializa y crea el objeto de lo enviado anteriormente mediante OutputStream. En este caso, como es el cliente el único que le envía al servidor, haremos un **casting** del objeto recibido, para comprobar que efectivamente es un **CommandMessage**, el único objeto que debe recibir el servidor de parte del cliente. Una vez comprobado si pertenece a la clase correcta, si el mensaje pidió una descarga de archivo se busca en el **Path correspondiente** si el archivo existe, y si existe se empaquetan los bytes y se envía la respuesta del archivo pedido.

En caso de que el objeto del servidor sea un **CommandMessage** y no sea una petición de descarga, se llamará al método **processCommand** de la clase **ServerCommandProcess** añadiendo de argumentos el comando y el path, que procesará una respuesta según el comando que haya sido. Una vez recibida la respuesta que le debe pasar se envía mediante el **ObjectOutputStream**

6.4. ServerCommandProcess.java

El ServerCommandProcess, es el encargado de procesar los objetos **CommandMessage** que recibe el servidor, y actuar en consecuencia. Cada comando requiere una función propia, y al optar por hacerlo con un enfoque distinto al de clase, donde cada comando podría ser una clase distinta y que cada función se ejecute según el objeto, se ha hecho un **hashmap** que contiene las funciones que se ejecutarán según el comando que se reciba.

Todas las funciones que se ejecutan en el hashmap, reciben un objeto de tipo **commandMessage**, que viene de **SimpleServer**. Ya habiendo pasado las validaciones tanto de tipado (el casting), como de argumentos (En la parte del cliente). Cada función del hashmap también recibe un objeto de tipo **Path**, del paquete de java.nio, que será el directorio en el que se ejecutará el comando, el cual se procesa al inicio de la clase. Este path es una combinación del basepath, donde se encuentra el storage, y el clientpath, que será el directorio en el el servidor entiende que se encuentra el cliente. Se tiene en cuenta que el cliente no pueda salir del directorio base.

Cada función opera de forma distinta, según el comando, aunque la mayoría son operaciones que nos permite ejecutar el paquete **java.nio**

6.5. ServerContextHandler.java

El serverContextHandler hace función similar que el clientContextHandler, pero en menor medida, ya que el dispositivo que actúa como servidor no tiene que enviar muchos comandos, por lo que tiene comandos basicos como –close o –help, para cerrar u obtener ayuda de los

posibles comandos del servidor. No está pensado para interactuar mucho, su finalidad es recibir peticiones del cliente y procesarlas.

Bibliografía

[1] MadProgrammer. «How to set output stream to TextArea». URL: https://stackoverflow.com/questions/12945537/how-to-set-output-stream-to-textarea/12945678#12945678 (vid. pág. 6).