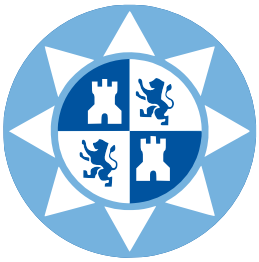

Universidad Politécnica de Cartagena


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 system, el contexto que tiene la máquina antes de poder pasar al contexto de cliente o servidor.

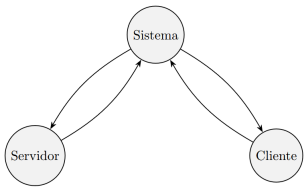


Figura 1.1. Máquina de estados diseñada para el proyecto.

Teniendo en cuenta la maquina de estados (funciona tal tal....), la estructura de archivos finalmente quedará organizada de la siguiente manera:

**FileSystem**

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

# Capítulo 2

# filetransfer

## 2.1. Uso del programa

## 2.2. FileSystem.java

## 2.3. Main.java

## 2.4. SystemContextHandler.java

# Capítulo 3

# common

## 3.1. CloseMessage.java

## 3.2. CommandMessage.java

Comandos disponibles:

- FILE_UPLOAD
- FILE_DOWNLOAD
- DIRECTORY_CREATE
- DIRECTORY_LIST
- DIRECTORY_LOCATION
- FILE_DELETE
- DIRECTORY_OPEN
- CON_CHECK

# Capítulo 4

# client

## 4.1. ClientContextHandler.java

## 4.2. ClientMain.java

## 4.3. ClientUtils.java

## 4.4. SimpleClient.java

# Capítulo 5

# server

## 5.1. 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.

## 5.2. 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.

## 5.3. 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**

## 5.4. 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**

## 5.5. 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.