MEDAC
Instituto Oficial de Formación Profesional

PROGRAMACIÓN DE SERVICIOS Y PROCESOS
TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Los servicios en red. Sockets II

11

ÍNDICE

OBJETIVOS

Conocer cómo realizar un cliente HTTP.

Conocer cómo realizar un cliente FTP.

Conocer cómo realizar un cliente Telnet.

Conocer cómo realizar un cliente SMTP.

Crear aplicaciones con sockets que pueden atender a varios clientes usando hilos.

/ 1. Introducción y contextualización práctica

En esta unidad vamos a continuar con el estudio de los servicios en red, y concretamente con el concepto de socket, aprendiendo cómo podemos crear diferentes tipos de clientes en el lenguaje de programación Java.

Concretamente, vamos a ver cómo podemos crear los siguientes tipos de clientes:

- Cliente HTTP.
- Cliente FTP.
- Cliente Telnet.
- Cliente SMTP.

Además, vamos a crear aplicaciones con sockets que implementen un servidor que pueda atender a varios clientes a la vez, utilizando para ello, hebras.

Escucha el siguiente audio en el que planteamos el caso práctico que iremos resolviendo a lo largo de esta unidad.

Fig. 1. Usando un cliente HTTP.

Audio Intro. "Aplicaciones concurrentes"
https://bit.ly/32YKADy

/ 2. Programación de un cliente HTTP

La implementación del protocolo HTTP o HTTPS se basa en un **proceso sencillo que implica una serie de solicitudes y respuestas** a esas solicitudes **por parte tanto del cliente, como del servidor:**

- En primer lugar, **un cliente establecerá una conexión con un servidor,** enviando para ello un mensaje de solicitud con los datos pertinentes.

- Cuando el **servidor** reciba dicho mensaje, le **responderá con un mensaje** muy similar, conteniendo éste el resultado de la operación solicitada por el cliente.

El **lenguaje de programació**n Java dispone de dos clases que nos van a permitir programar aplicaciones donde tengamos tanto servidores como clientes HTTP. Estas clases son:

- **La clase URL:** Esta clase nos va a permitir representar una dirección de una página web, por ejemplo, https://www.google.es, de forma que el programa pueda realizar operaciones con ella.

  Si quieres saber más sobre esta clase puedes consultar su documentación oficial en:

  https://docs.oracle.com/javase/7/docs/api/java/net/URL.html .

- **La clase URLConnection:** Esta clase es la que nos va a permitir realizar operaciones con la dirección web que hemos creado mediante URL. Podremos lanzar operaciones tipo GET y obtener las respuestas de éstas de una forma muy sencilla.

  Si quieres saber más sobre esta clase puedes consultar su documentación oficial en:

  https://docs.oracle.com/javase/7/docs/api/java/net/URLConnection.html .

Al utilizar estas clases estamos programando un servicio HTTP de alto nivel, esto quiere decir que todas las operaciones que harán posible la comunicación no se visualizarán, y para trabajar con ello, será tan sencillo como programar un servidor o cliente mediante el uso de *sockets*.

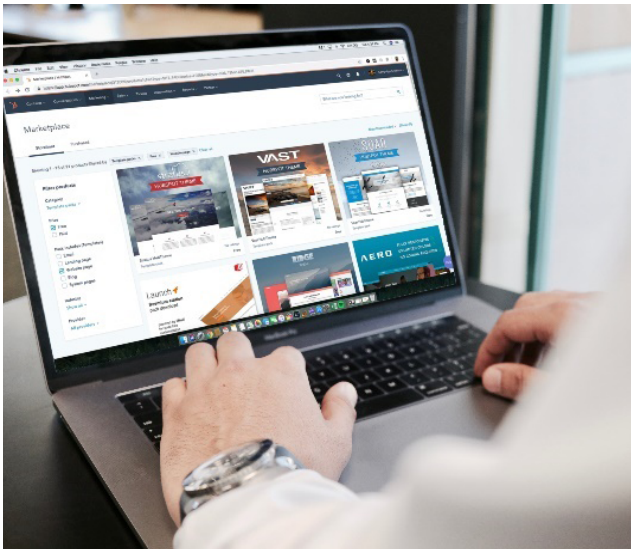

Fig. 2. Realizando peticiones HTTP.

Audio 1. "Procesamiento de peticiones HTTP"
https://bit.ly/2HgRf4r

/ 3. Programación de un cliente FTP

Las siglas del protocolo FTP significan *File Transfer Protocol,* o Protocolo de Transferencia de Ficheros, y es el protocolo que debemos usar siempre que deseemos t**ransferir ficheros entre un servidor y un cliente** o viceversa.

Los pasos para crear un cliente FTP son los siguientes:

- Realizar una conexión al servidor.
- Comprobar que la conexión que se ha realizado con éxito.
- Validar el usuario FTP que se ha conectado. En caso de que el usuario no sea válido deberemos abortar la conexión y enviar un mensaje de error.
- Realizar las operaciones pertinentes con el servidor.
- Desconectar del servidor una vez terminemos de realizar las operaciones requeridas.

No hay que olvidar que todo el proceso de conexiones y realización de operaciones puede generar excepciones, concretamente puede lanzar las **excepciones** *SocketException e IOException*.

El lenguaje de programación Java no dispone de clases específicas para el uso del protocolo FTP, pero la fundación Apache creó la API *org.apache.commons.net.ftp* para trabajar con clientes y servidores FTP. Esta **API** tiene las siguientes clases que **nos permiten operar** de una manera sencilla con el protocolo FTP:

- **Clase FTP:** Esta es la clase que nos va a proporcionar todas las funcionalidades básicas para poder realizar un cliente FTP básico. Esta clase hereda de *SocketClient*.
- **Clase *FTPReplay*:** Esta es la clase que nos va a permitir operar con los valores devueltos por las consultas FTP del servidor.
- **Clase *FTPClient*:** Esta clase es la encargada de dar soporte a las funcionalidades del cliente FTP. Hereda de *SocketClient*.
- **Clase *FTPClientConfig*:** Esta clase nos va a permitir realizar las configuraciones oportunas de los clientes FTP de una forma sencilla.
- **Clase *FTPSClient*:** Esta clase nos va a permitir utilizar el protocolo FTPS, que es la versión segura del protocolo FTP. Esta clase utiliza el protocolo SSL y hereda de *FTPClient*.

Fig. 3. Ficheros.

/ 4. Caso práctico 1: "Seguridad ante todo"

Planteamiento: Pilar y José acaban de recibir por parte de su profesor un nuevo ejercicio. Este consiste en que deben diseñar el funcionamiento de un servidor FTP, que debe ser capaz de controlar la seguridad del sistema y en caso de que todo esté correcto devolver un valor aleatorio al cliente.

Pilar y José se ponen manos a la obra.

Nudo: ¿Cuáles crees que pueden ser los errores de seguridad mínimos que deben controlarse desde el servidor antes de realizar la operación solicitada por algún cliente?

Desenlace: La seguridad en los servidores es algo totalmente necesario, ya que puede exponerse código o documentos confidenciales. No solo con los servidores FTP tenemos que hacer una serie de comprobaciones básicas, sino que con cualquier tipo de servicio deberíamos de tener unas mínimas comprobaciones de seguridad.

Para una mínima seguridad en el sistema podríamos realizar las siguientes comprobaciones una vez nos llegue una petición de un cliente:

- **Comprobar que el cliente es un cliente válido del sistema.** Para esto deberemos mantener el listado de clientes que tenga el sistema en una pequeña base de datos o, en su defecto, en un fichero.
- **Comprobar que la operación que nos pide el cliente es una operación válida en el sistema.** Para esto basta con saber qué operaciones hemos diseñado en nuestro sistema y cuáles son sus parámetros.
- **Comprobar que el cliente tiene permisos para realizar dicha operación.** De igual forma que deberemos tener un registro de clientes válidos en el sistema, le podemos agregar una serie de datos indicando qué operaciones pueden realizar.

```
// Comprobamos que el cliente es válido
c_valido = comprobarCliente(nombre)

// Comprobamos que la operación solicitada
// es válida
op_valida = comprobarOperacion(operacion)

// Comprobamos que el cliente tiene permiso para
// realizar dicha operación
permisos = comprobarPermisos(nombre, operacion)

si c_valido = verdad Y op_valida = verdad Y permisos = verdad
    // Realizamos la operación solicitada
sino
    // Devolvemos un mensaje de error
```

Fig. 4. Pasos a seguir.

/ 5. Programación de un cliente Telnet

El protocolo Telnet, cuyo nombre significa *TELecomunication NETwork*, nos va a permitir **acceder a otros equipos conectados a nuestra red,** pudiendo realizar así una administración de forma remota, como si estuviéramos sentados frente al equipo que estamos administrando remotamente.

Gracias a este protocolo, se hace muy simple la administración de equipos que no tengan pantalla ni teclado, es decir, que sean simplemente un servidor que se arranca y lanza todas sus tareas y servicios automáticamente.

El protocolo Telnet está basado en:

- Tiene el esquema básico del protocolo cliente/servidor.
- El puerto que utiliza es el 23.
- Funciona mediante comandos en modo texto.

Aunque sea un protocolo que nos ayudará y simplificará la administración de equipos, **no ofrece mucha seguridad,** y es por esta misma razón por la que apenas se utiliza en las grandes empresas. Esto se debe a que toda la información se transmite en texto plano, incluidos los datos y contraseñas de los usuarios, cuando toda la información debería intercambiarse cifrada.

La biblioteca que Apache nos ofrece en su paquete *org.apache.commons.net.telnet* para poder trabajar con este protocolo es:

- **Clase *TelnetClient*:** Esta es la clase que nos va a permitir implementar un terminal para usar el protocolo Telnet. Esta clase hereda de la clase SocketClient.

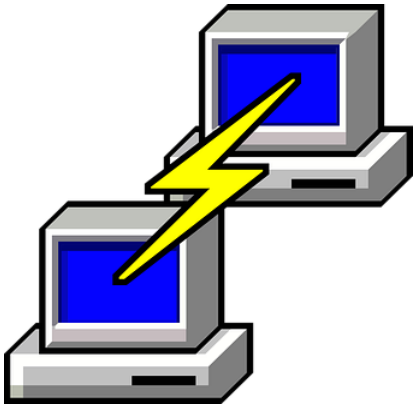

Fig. 5. Icono de la administración mediante Telnet.

Entre los métodos más útiles de esta clase tenemos:

- **El método SocketClient.connect(),** que nos permitirá realizar una conexión al servidor.
- **Los métodos TelnetClient.getInputStream() y TelnetClient.getOutputStream(),** que nos permitirán obtener los flujos de comunicación.
- **El método TelnetClient.disconnect(),** que nos permitirá desconectar.

/ 6. Programación de un cliente SMTP

El protocolo SMTP es el que se utiliza para **enviar y recibir correos electrónicos.**

Para poder implementar un cliente SMTP vamos a utilizar la API *javax.mail* que nos proporciona el lenguaje de programación Java.

Dentro de esta API tenemos las siguientes clases que nos van a permitir realizar una gestión de correos electrónicos de una manera fácil y sencilla:

- **Clase Session:** Esta clase representa **una sesión de usuario para correo electrónico.** Aquí vamos a tener agrupada toda la configuración por defecto que utiliza la API *javax.mail* para la gestión de correos electrónicos. Mediante el método getDefaultInstance() podremos obtener una sesión por defecto, con toda su configuración correspondiente.

  Si quieres más información sobre esta clase puedes visitar su documentación oficial en:

  https://docs.oracle.com/javaee/7/api/javax/jms/Session.html .

  https://docs.oracle.com/javaee/6/api/javax/mail/Session.html .

- **Clase Message:** Esta clase representa un **mensaje de correo electrónico.** Podremos configurar el remitente mediante el método setFrom(), el asunto del correo electrónico, mediante el método setSubject(), y el texto del mismo, mediante el método setText().

  Si quieres más información sobre esta clase puedes visitar su documentación oficial en:

  https://javaee.github.io/javamail/docs/api/javax/mail/Message.html .

- **Clase Transport:** Esta clase es la que representa **el envío de los correos electrónicos.** Hereda de la clase *Service*, que es una clase que proporciona las funcionalidades comunes a todos los servicios de mensajería.

  Si quieres más información sobre esta clase puedes visitar su documentación oficial en:

  https://docs.oracle.com/javaee/7/api/javax/mail/Transport.html .

Fig. 6. SMTP, protocolo para el e-mail.

Mediante las clases anteriores podemos crear fácilmente una aplicación que envíe correos electrónicos mediante nuestra cuenta de Gmail, por ejemplo.

/ 7. Sockets e hilos I

Hasta el momento, y sobre todo en el tema anterior, hemos implementado una serie de servidores y clientes utilizando los **protocolos TCP y UDP**, pero estos tenían **una gran deficiencia,** y es que los servidores únicamente podían **atender a un cliente a la vez,** teniendo los demás clientes que querían solicitar un servicio a nuestro servidor, esperar a que el cliente que estuviera en acción terminara su proceso, formando así una cola de espera de clientes.

Esto es algo contraproducente para programar cualquier servicio, siendo lo ideal que cada servidor pueda atender a muchos clientes al mismo tiempo, haciendo así una implementación concurrente de los servicios.

Si recordamos de unidades anteriores, **la concurrencia la conseguíamos utilizando hebras o hilos,** haciendo así que se lanzase **una hebra por cada tarea concurrente** que quisiéramos realizar, dando la sensación de que se estaban ejecutando todas al mismo tiempo.

A la hora de programar servicios con ***sockets*** también vamos a hacer uso de las hebras, haciendo que cada servidor pueda atender a varios clientes a la vez.

Cuando el servidor detecte que un cliente le ha hecho una petición éste deberá aceptarla, procesarla y crear una hebra que sea capaz de atender la petición del cliente.

De esta forma, cada cliente será atendido por una hebra diferente y el servidor podrá volver a escuchar una petición nueva de otro cliente, inmediatamente después de lanzar la hebra que atenderá al primer cliente.

De esto debemos deducir que las hebras se ejecutarán en la parte del servidor, ya que es este el que atiende la petición del cliente, de forma que el cliente no será consciente de si lo está atendiendo el propio servidor o una hebra en el mismo.

```
// Crear socket
while (Boolean.TRUE) {
    // 1. Aceptar una solicitud de cliente
    // 2. Código
    // 3. Crear una hebra independiente
        // en el servidor para procesar
        // la solicitud
}
```

Fig. 7. Boceto de utilización de hilos con sockets.

Vídeo 1. "Monitorización de tiempos de respuesta"
https://bit.ly/3nHkHAb

/ 8. Caso práctico 2: "DNI concurrentes"

Planteamiento: Pilar y José acaban de recibir un nuevo encargo por parte de su profesor. Este ejercicio es uno que ya tuvieron que resolver anteriormente. Trata sobre calcular la letra del DNI de una persona conociendo únicamente su número.

En esta ocasión, deberán implementar el servicio de forma concurrente, para ello, tendrán que realizar un servidor TCP que una vez que acepte la solicitud con el número del DNI del cliente, lance una hebra que atienda dicha solicitud.

Nudo: ¿Cómo crees que se puede solucionar este problema? ¿Qué parámetros habrá que pasarle a la hebra del servidor?

Desenlace: Para poder realizar este ejercicio primero tenemos que tener muy claro que la hebra que lancemos se ejecutará en el servidor, y gracias a esto podremos pasar los datos que necesitemos del cliente que a la misma, ya que el servidor en sí los tiene.

Para resolver este problema deberemos crear una clase que sea una hebra, pudiendo hacerlo de los dos métodos posibles, heredando de *Thread* o implementando *Runnable*. A esta clase le tendremos que implementar un método que proporcione el servicio deseado, siendo en este caso una función que calcule y devuelva la letra del DNI.

Dentro de los parámetros de la clase le vamos a pasar el número del DNI del cliente y además el flujo de salida del cliente, para que de esta forma podamos enviar la letra al cliente una vez calculada.

Si hemos seguido estos pasos podremos atender a tantos clientes como necesitemos de forma concurrente, aunque siempre es muy recomendable establecer un tope máximo de clientes, dependiendo esto del servicio que queramos dar y de las prestaciones de nuestro servidor.

```
1  // Aceptamos el cliente
2  cliente = esperarCliente()
3
4  // Obtenemos el flujo de entrada y el número
5  entrada = cliente.obtenerFlujoEntrada()
6  numerodni = entrada.obtenerMensaje()
7
8  // Obtenemos el flujo de salida del cliente
9  salida = cliente.obtenerFlujoSalida()
10
11 // Creamos el objeto de la hebra y la lanzamos
12 hebraservidor = Hebra(numerodni, salida)
13 hebraservidor.lanzar()
```

Fig. 8. Pseudocódigo del servidor concurrente.

/ 9. Sockets e hilos II

Vamos a ver cómo podemos crear un programa cliente/servidor concurrente con TCP mediante hilos.

Para esto deberemos crear las siguientes clases:

- **Clase Servidor:** Esta clase será la que va a representar el servidor de nuestra aplicación.
- **Clase ServidorHebra:** Esta clase será la que va a representar una hebra que se lanzará en el servidor y que será la encargada realmente de dar servicio al cliente conectado, haciendo posible la concurrencia.
- **Clase Cliente:** Esta clase será la que va a representar un cliente de nuestra aplicación. Se podrán ejecutar tantos clientes como se quiera.

Dentro de la clase servidor deberemos crear un **método *main*,** en el que crearemos un *ServerSocket* con el que esperaremos que se conecte un cliente.

Una vez conectado el cliente a nuestro servidor, **deberemos crear un objeto** de la clase *ServidorHebra*, al que le pasaremos como mínimo, los flujos de entrada y salida del cliente, para poder así poder establecer una comunicación con él. Después de crear y lanzar la hebra, nuestro servidor volverá a escuchar para que se conecte otro cliente al que dar servicio.

Dentro de la clase cliente deberemos crear un *main*, en el que creamos un Socket que conectaremos al servidor. Una vez hecho esto actuaremos exactamente igual que vimos en unidades anteriores, comunicándonos con el servidor normalmente.

Dentro de la clase *ServidorHebra* deberemos implementar el método run y el método que ejecutará la hebra, implementando en dicho método toda la funcionalidad que debe dar el servidor normalmente, es decir, realizando una comunicación normal con el cliente, tanto para recibir como para enviarle información.

Será dentro de la clase *ServidorHebra* donde implementemos todas las funciones que necesitemos para poder dar los servicios requeridos a los clientes.

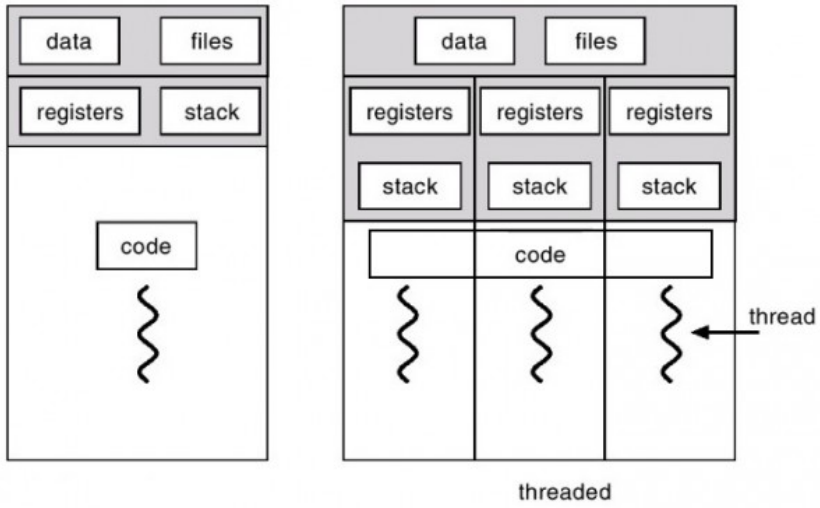

Fig. 9. Servicio sin hilos y con hilos. NO ORIGINAL. Fuente: aqui

Vídeo 2. "Ejemplo de sockets con hilos paso a paso"
https://bit.ly/330ruwJ

/ 10. Resumen y resolución del caso práctico de la unidad

A lo largo de esta unidad hemos visto cómo podemos crear diferentes tipos de clientes en el lenguaje de programación Java. Concretamente, hemos estudiado los siguientes tipos de clientes:

- Cliente HTTP.
- Cliente FTP.
- Cliente Telnet.
- Cliente SMTP.

De todos estos tipos de clientes hemos visto qué clases y bibliotecas deberemos utilizar para poder crear cada uno de ellos, todas ellas ya integradas dentro de la JDK de Java. Además de las clases hemos trabajado con algunos de los métodos que ofrecen, y que implementan las operaciones más utilizadas.

Por último, hemos aprendido a crear aplicaciones son Sockets que implementen un servidor que pueda atender a varios clientes a la vez, utilizando para ello hebras. En la parte de recursos del tema, podrás encontrar el código generado correspondiente a los vídeos 1 y 2.

Resolución del caso práctico de la unidad

Cuando tenemos que mejorar una aplicación lo primero que debemos hacer, sea cual sea la mejora que se necesite implementar en la aplicación, es ver cómo actúa en todas las situaciones que deseemos mejorar.

En el caso que concierne a Pilar y José, deben hacer que la aplicación sea capaz de trabajar con más de un usuario al mismo tiempo, por lo que debemos ver qué ocurre cuando una vez un usuario esté conectado intentemos conectar otro más al sistema.

Fig. 10. Probando el sistema.

En el caso de esta aplicación, el problema está en que no está hecha para soportar más de un usuario conectado al mismo tiempo, así que la solución más sencilla es implementar un sistema de hebras que permita una ejecución concurrente de usuarios, tal y como hemos visto en esta unidad.

/ 11. Bibliografía

Gómez, O. (2016). Programación de Servicios y Procesos Versión 2.1. https://oscarmaestre.github.io/servicios/

Sánchez, J. M., & Campos, A. S. (2014). Programación de servicios y procesos. Alianza Editorial.

Colaboradores de Wikipedia. (2020, 17 julio). Telnet. Wikipedia, la enciclopedia libre. https://es.wikipedia.org/wiki/Telnet

Java — Cómo usar Java.net.URLConnection para disparar y manejar solicitudes HTTP. (2010, 8 mayo). it-swarm.dev. https://www.it-swarm.dev/es/java/como-usar-java.net.urlconnection-para-disparar-y-manejar-solicitudes-http/970069550/

Procesamiento URL de Java. (s. f.). http://www.w3big.com/ Recuperado 29 de agosto de 2020, de: http://www.w3big.com/es/java/java-url-processing.html

Reading *from and Writing to a URLConnection (The JavaTM Tutorials > Custom Networking > Working with URLs)*. (s. f.). Oracle. Recuperado 29 de agosto de 2020, de: https://docs.oracle.com/javase/tutorial/networking/urls/readingWriting.html