Práctica 3: Servidores TCP secuenciales

Esta práctica pretende familiarizarnos con la programación de servidores que emplean *sockets* TCP. Además, estos servidores van a ser secuenciales, en contraposición a los servidores concurrentes. Un servidor secuencial no atiende a un nuevo cliente hasta que termina con el anterior. Diversos clientes son atendidos uno tras otro, de forma secuencial. Por contra, un servidor concurrente puede atender varios clientes al mismo tiempo.

1. Clases para crear servidores TCP

En esta práctica vamos a transferir mensajes del nivel de aplicación usando el protocolo de nivel de transporte TCP. Por ello, en este apartado veremos las clases básicas para comunicar un cliente y un servidor mediante *sockets* TCP. La información que proporcionamos sobre las clases java que hay que emplear y sus métodos es bastante limitada. Para ver los detalles completos es aconsejable consultar la página web de Oracle o bien los numerosos manuales y tutoriales que se pueden encontrar en Internet.

Como mínimo necesitaremos dos clases pertenecientes al paquete **`java.net.*`**, las clases **`Socket`** y **`ServerSocket`**:

1. **`Socket`** permite establecer una conexión entre un cliente y un servidor TCP. El cliente crea un socket e inicia la conexión con el servidor. Una vez conectado al otro extremo utiliza este socket para el envío y la recepción de información.
2. **`ServerSocket`** permite a un servidor escuchar en un puerto a la espera de una conexión TCP. Al establecerse ésta, en el programa servidor se crea un objeto de la clase **`Socket`** que se emplea para el intercambio de información posterior entre ambos extremos.

Los servidores que programemos esperarán las conexiones de los clientes escuchando en un puerto previamente determinado (es el comportamiento típico en un servidor). Como nuestros servidores no tienen privilegios de administrador del sistema, el puerto con el que trabajen deberá ser uno mayor que el 1023.

Para esperar la conexión de un cliente hay que invocar el método **`accept()`** de la clase **`ServerSocket`**. Este método bloquea la ejecución del programa a la espera de una conexión. Al recibir una petición de conexión, **`accept()`** crea un nuevo objeto de la clase **`Socket`**, que se usará durante el resto de la comunicación con el cliente. El nuevo socket creado hereda algunas características del socket original (el objeto de tipo

ServerSocket) como la dirección IP y el número de puerto local. La diferencia principal entre ambos tipos de sockets es que el nuevo socket está conectado con el cliente y, por tanto, se identifica mediante 4 parámetros: la dirección IP del servidor, el puerto del servidor, la dirección IP del cliente y el número de puerto del cliente. Recordemos que al igual que ocurre en el socket del cliente, que también utiliza en java un objeto del tipo **Socket**, el valor de estos 4 parámetros puede obtenerse mediante los métodos de la clase **Socket**:

- **getLocalPort()**: devuelve el puerto local al que el socket está conectado. También es un método de la clase **ServerSocket**.
- **getLocalAddress()**: devuelve la dirección IP local a la que el socket está conectado. También es un método de la clase **ServerSocket**.
- **getPort()**: devuelve el puerto remoto al que el socket está conectado.
- **getInetAddress()**: devuelve la dirección IP remota a la que el socket está conectado.

El empleo del método **accept()** puede generar una excepción **IOException**, por lo que, o bien habrá que capturarla mediante una cláusula **try**, tal y como se hizo prácticas anteriores, o bien puede lanzarse (**throws**) al programa o función que llamó al método que genera la excepción. De esta manera no será necesario programar una cláusula **try**. En nuestro caso, dado que es la máquina virtual Java quien llamó al método **main**, esta excepción se lanzará hacia ella.

El esqueleto de un servidor TCP secuencial podría ser el siguiente:

```
import java.net.*;
import java.io.*;
class ServidorTCP {
public static void main(String args[]) {
   try{
         ServerSocket ss=new ServerSocket(puerto);
         while(true){
               Socket s=ss.accept(); // espera un cliente
               //código para dar servicio al cliente
               s.close();
         } //while
       } //try
   catch(IOException e) { System.out.println(e); }
 } //main
} // class
```

2. Gestión de la entrada/salida

Una vez el servidor se ha conectado con un cliente, con el fin de manejar la transferencia de información a través del *socket* recién creado, la clase **`Socket`** dispone de dos métodos: **`getInputStream()`** y **`getOutputStream()`**, que proporcionan un flujo de entrada y uno de salida, respectivamente. Como ya vimos en prácticas anteriores, es mejor no manejar estos flujos directamente, sino envolverlos por otras clases más cómodas de utilizar como **`Scanner`** o **`Printwriter`**.

A continuación se muestra, de forma breve, un ejemplo de uso de estas clases y algunos de sus métodos:

```
import java.util.Scanner;
...
Scanner recibe=new Scanner(cliente.getInputStream());
entrada.nextLine();
...
PrintWriter envia=new
                    PrintWriter(cliente.getOutputStream());
envia.printf("mensaje a enviar");
envia.flush();
```

donde **`cliente`** es un objeto de la clase **`Socket`** conectado con un cliente.

Ejercicio 1:

Escribe un servidor de un único uso (atiende a un cliente y termina) que escuche en el puerto 7777. Tras aceptar la conexión del cliente, el servidor imprime en pantalla el mensaje: "Se ha conectado un cliente al servidor" y termina. (Puede probarse con la orden "nc localhost 7777").

Ejercicio 2:

Siguiendo el modelo anterior, escribe un servidor que escuche en el puerto 7777 y devuelva al cliente la hora del día. Para el cálculo de la hora puedes emplear la clase Calendar del paquete java.util:

```
Calendar now = Calendar.getInstance();
int h = now.get(Calendar.HOUR_OF_DAY);
int m = now.get(Calendar.MINUTE);
int s = now.get(Calendar.SECOND);
```

¿Qué ocurre si intentas ejecutar el cliente dos veces sin volver a ejecutar el servidor? ¿Crees que es el comportamiento habitual de un servidor?

Ejercicio 3:

Modifica el servidor del ejercicio 2 para que atienda a los clientes de forma ininterrumpida.

¿Te has acordado de cerrar el socket que conecta con el cliente?

Ejercicio 4:

Escribe un servidor espejo de cabeceras HTTP. Tras recibir la petición del navegador, el servidor devolverá al cliente una página web conteniendo las cabeceras que éste le había enviado en su petición. Es decir, la información de control que ha enviado el cliente es usada por el servidor como datos para su página web. Por lo tanto, la respuesta del servidor tendrá la forma:

Línea de estado: "HTTP/1.0 200 OK \r\n"

"Content-Type: text/plain \r\n"

LíNEA EN BLANCO

Cuerpo: aquí deben ir las cabeceras recibidas desde el navegador

Después de enviar la respuesta, el servidor cerrará la conexión y esperará la llegada de un nuevo cliente.

¿Por qué hay que incluir una línea en blanco? Cuando tu servidor funcione correctamente puedes comprobar lo que ocurre si eliminas la línea en blanco en la respuesta al navegador.