Práctica 5: Servidores TCP secuenciales

Esta práctica pretende introducir al alumno en la programación de servidores que emplean *sockets* TCP. Los servidores, en función del número de clientes que son capaces de atender al mismo tiempo, pueden clasificarse en secuenciales y concurrentes. Un servidor secuencial no atenderá a un nuevo cliente hasta que termine con el anterior. Por lo tanto, diversos clientes serán atendidos uno tras otro, de forma secuencial. Por el contrario, un servidor concurrente puede atender a varios clientes al mismo tiempo. Por este motivo, la programación de servidores concurrentes resulta un poco más compleja y se analizará en una práctica posterior, centrándonos de momento en los servidores secuenciales.

Objetivos de la práctica

Al acabar esta práctica serás capaz de implementar servidores secuenciales TCP sencillos en java. En particular podrás:

- Crear y utilizar objetos de la clase **`ServerSocket`** para poner un servidor a la escucha en un puerto y establecer una conexión TCP con un cliente.
- Diferenciar las características de los *sockets* creados mediante objetos de tipo **`ServerSocket`** o de tipo **`Socket`**.
- Enviar y recibir información de tipo texto a través de un *socket* conectado mediante una conexión TCP.

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 la imprescindible para poder abordar con éxito la práctica. Para ver los detalles completos es aconsejable consultar la página web de Oracle[1] 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* (mediante un objeto de tipo **`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.

[1] http://docs.oracle.com/javase/8/docs/api/index.html

2. **`ServerSocket`** permite a un servidor escuchar en un puerto a la espera de recibir una petición de conexión TCP. Al establecerse ésta, en el programa servidor se crea un objeto de la clase **`Socket`**, que es el que queda conectado con el cliente y que permite 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, 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 petición de conexión TCP. Al recibirla, se crea un nuevo objeto de la clase **`Socket`**, conectado con el cliente, que será el qe se use durante el resto de la comunicación cliente-servidor.

El empleo del método **`accept()`**, o la creación de un objeto **`ServerSocket`** pueden 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`**.

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();
         }
        }
   catch(IOException e) { System.out.println(e); }
   }
  }
```

Como se aprecia en el código anterior, el bloque **`while(true)`** permite que el servidor pueda conectar con un nuevo cliente tras acabar de atender al que tenía en curso. Es decir, si hay varias solicitudes de diversos clientes, el servidor los irá atendiendo uno a uno. Es importante que al terminar de atender a un cliente cerremos el *socket* conectado con él para liberar los recursos asignados por el sistema.

2. Gestión de la entrada/salida

Una vez el servidor 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()`,** que proporciona un flujo de entrada que permite recibir los datos que envía el cliente.
- **`getOutputStream()`**,que proporciona un flujo de salida para enviar datos al cliente.

Como ya vimos en prácticas anteriores, para el intercambio de texto es mejor no manejar estos flujos directamente, sino a través de otras clases como **`Scanner`** o **`Printwriter`**.

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

```
import java.util.Scanner;
import java.io.*;
import java.net.*;
...
Scanner recibe=new Scanner(cliente.getInputStream());
String s = recibe.nextLine();
...
PrintWriter envia=new PrintWriter(cliente.getOutputStream());
envia.printf("mensaje a enviar\r\n");
envia.flush();
```

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

Ejercicio 1:

Escribe un servidor de eco que escuche en el puerto 7777. El servidor debe ejecutar el siguiente bucle:

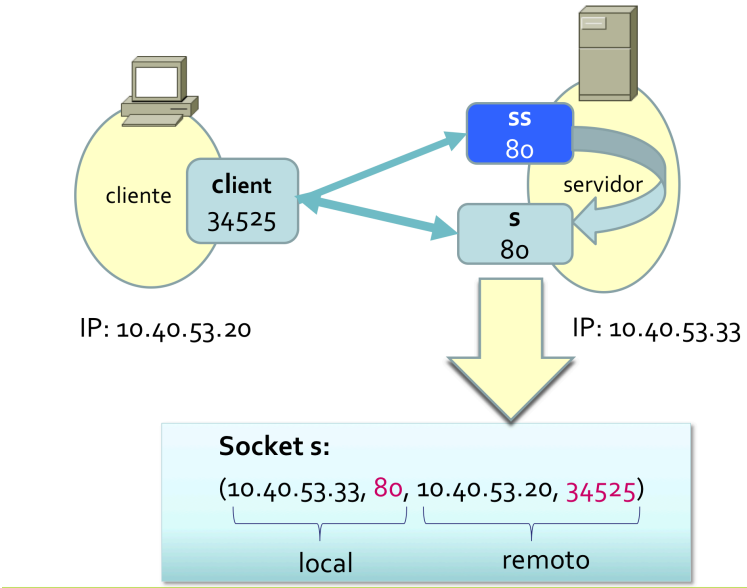
1. Esperar hasta establecer conexión con un cliente.
2. Devolver al cliente (a través del *socket*) la primera línea de texto que el cliente le envíe.
3. Imprimir en la pantalla del servidor el mensaje: "Se ha conectado un cliente al servidor"
4. Cerrar el *socket* conectado al cliente.

Puedes comprobar el funcionamiento de tu servidor con la orden "`nc localhost 7777`", ejecutada desde un terminal o consola de tu ordenador. Recuerda que el cliente

debe enviar una línea de texto al servidor (que tendrás que teclear tú) y obtener como respuesta la misma línea.

3. Diferencias entre los dos tipos de *sockets* utilizados

Existe una diferencia importante entre los dos tipos de *sockets* que estamos utilizando en la práctica. Mientras el *socket* inicial, generado en java mediante un objeto de tipo **`ServerSocket`**, es un *socket* no conectado, el *socket* que se genera como resultado del método **`accept()`** (en java un objeto de tipo **`Socket`**) está asociado a un cliente mediante una conexión TCP. Ambos *sockets* utilizan la misma dirección IP local y el mismo puerto local. Sin embargo, el *socket* conectado al cliente tendrá asociados también los valores remotos de la dirección IP y puerto del cliente.



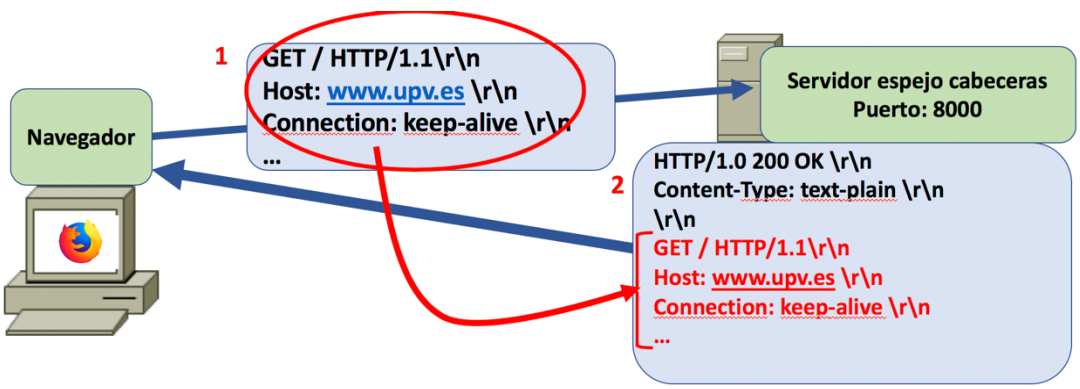
Recordemos que la clase **`Socket`** nos proporciona, entre otros, los métodos **`getLocalAddress()`**, **`getLocalPort()`**, **`getInetAddress()`** y **`getPort()`** que nos permiten obtener esos 4 valores asociados al **`Socket`** y que identifican a la conexión TCP que tiene establecida. Por el contrario, la clase **`ServerSocket`** nos ofrece únicamente dos de esos cuatro métodos, los asociados a los valores locales: **`getInetAddress()`**, **`getLocalPort()`**. Observa que el método **`getInetAddress()`** tiene significado distinto según se trate de objetos **`Socket`** o **`ServerSocket`**. Mientras que el método **`getInetAddress`** de la clase **`Socket`** devuelve la dirección remota con la que está conectado el **`Socket`**, el mismo método en la clase **`ServerSocket`** devuelve la dirección IP local de este **`ServerSocket`**.

Ejercicio 2:

Modifica el servidor del ejercicio 1 para que visualice en pantalla todos los valores de dirección IP y puerto utilizados por los *sockets* del programa, indicando si corresponden al objeto de tipo **`ServerSocket`** o **`Socket`**. Comprueba la diferencia en los valores mostrados al utilizar distintos nombres para conectarte al servidor. Por ejemplo, puedes probar la orden "`nc localhost 7777`" y la orden "`nc nombre_de_tu_ordenador 7777`", siendo nombre_de_tu_ordenador el nombre de dominio de tu puesto de trabajo. Este nombre dependerá del laboratorio en el que te encuentres, en el laboratorio de Redes será de la forma rdcXX.redes.upv.es. También podrías utilizar diferentes direcciones IP asociadas a tu ordenador.

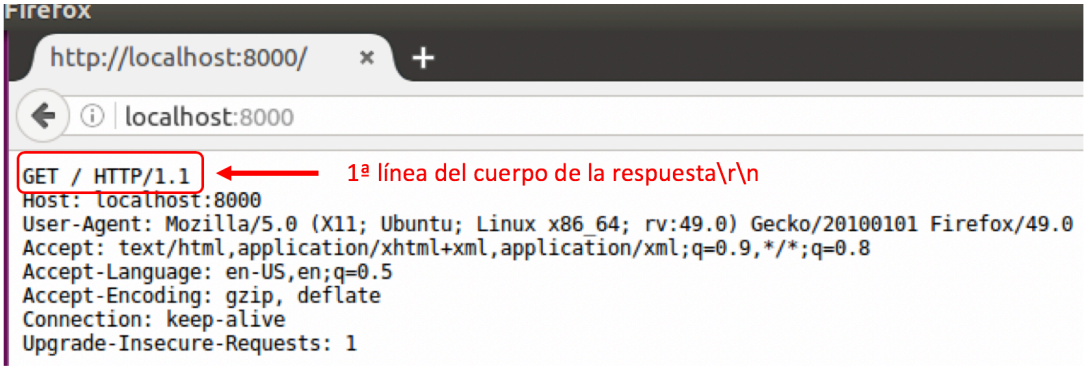
NOTA: *Cuando en el constructor de la clase* ***`ServerSocket`*** *no se especifica, la dirección IP del servidor con la que queremos enlazar nuestra aplicación, el método* ***`getInetAddress()`*** *proporciona como respuesta la dirección IP 0.0.0.0/0.0.0.0. Esto significa que nuestro servidor atenderá a cualquier petición de conexión que llegue a cualquier interfaz o adaptador de red que tenga nuestro equipo. Si al crear el* ***`ServerSocket`*** *hubiéramos especificado una IP concreta de todas las que disponga nuestro equipo, este método nos devolvería esa IP (por ejemplo la IP del interfaz Ethernet, o la del interfaz WiFi, la IP local asociada a localhost, …; pero en estos casos nuestro servidor sólo atendería a peticiones que llegaran a esa interfaz particular, pero no al resto de interfaces que tiene).*

Un ejercicio que puede resultarnos de utilidad es programar un servidor que devuelva la información de control que ha enviado un navegador al conectarse con él. Es decir, un servidor que devuelva al navegador la línea del método inicial (GET /…) y el conjunto de cabeceras (Host: …) que le había enviado el navegador al realizar su petición. De esta forma, nuestro servidor permitirá que veamos la línea inicial y las cabeceras que envían distintos navegadores. Observa que este servidor, aunque nos permitirá ver la petición HTTP que ha realizado un navegador, NO es un servidor web propiamente dicho. Es más bien una especie de "servidor de eco (o espejo) del



protocolo HTTP".

Supongamos que hemos puesto nuestro servidor en ejecución escuchando, por ejemplo, en el puerto 8000. Podemos probarlo localmente con cualquier navegador poniendo, `localhost:8000` en la barra de direcciones del navegador. El resultado esperado es el que se muestra en la figura siguiente, donde podemos ver el conjunto de líneas que forman la petición HTTP enviada previamente por nuestro navegador al servidor. Es decir, la línea de petición con el método GET y las cabeceras que Firefox ha enviado cuando le hemos indicado que acceda a nuestro servidor.



Ejercicio 3:

Escribe un servidor espejo de cabeceras HTTP como se ha descrito en los párrafos anteriores. El servidor escuchará en el puerto 8000. Tras recibir la petición del **navegador**, el servidor devolverá al cliente una "página web" conteniendo la línea de petición y las cabeceras que éste le había enviado en su petición. Es decir, la información de control que ha enviado el cliente es el contenido que el servidor devolverá como cuerpo de su mensaje.

Por lo tanto, la información que debe enviar el servidor al cliente empezará con las siguientes líneas (sólo la columna de respuestas, no los comentarios):

Respuestas	**Comentarios**
`HTTP/1.0 200 OK \r\n`	1ª línea, línea de estado
`Content-Type: text/plain \r\n`	2ª línea, cabecera de respuesta
`\r\n`	Línea en blanco para separar
1ª línea del cuerpo de la respuesta`\r\n`	En el cuerpo de la respuesta se copiarán todas y cada una de las líneas que componen la petición original del navegador
2ª línea del cuerpo de la respuesta`\r\n`	
3ª línea del cuerpo de la respuesta`\r\n`	
…	

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

La forma más sencilla de reproducir este comportamiento es enviar primero las tres líneas que aparecen en azul (línea de estado HTTP, cabecera y fin de cabeceras) y después realizar un bucle donde el servidor va leyendo cada línea de texto que le envía el cliente y se la devuelve. Piensa detenidamente cuál deber ser la condición de salida del bucle (crucial para que el programa funcione).

Para comprobar el funcionamiento de tu servidor, abre un navegador e indica como URL http://localhost:8000.

¿Por qué el navegador no muestra las dos primeras líneas de la respuesta del servidor?

Elimina la línea en blanco en la respuesta que envía el servidor. ¿Qué efecto tiene el cambio? ¿Por qué es necesario incluir esta línea en blanco?