Práctica 6: Servidores concurrentes en Java

Esta práctica pretende familiarizarte con la programación de servidores concurrentes que emplean los servicios TCP mediante *sockets*. Para ello introduciremos brevemente los *Threads*, o hilos de ejecución, como herramienta básica para conseguir concurrencia en Java. Posteriormente nos centraremos en su aplicación a la programación de diferentes tipos de servidores concurrentes TCP.

Al acabar la práctica deberías ser capaz de:

1) Escribir un programa básico en Java que haga uso de hilos de ejecución con el objetivo de realizar varias tareas de forma concurrente.
2) Explicar la estructura básica de un servidor TCP concurrente, incluyendo la distribución del código del servidor entre los diferentes métodos de la clase *Thread*.
3) Convertir un servidor TCP secuencial sencillo escrito en Java en un servidor concurrente que ofrezca el mismo servicio, pero a varios clientes a la vez.

1. Implementación de la concurrencia

Como ya sabrás, cuando ejecutamos un programa el sistema operativo crea un proceso, le asigna recursos y controla cuando puede hacer uso del procesador. En el caso más sencillo, como hemos visto hasta ahora, los procesos constan de un único hilo de ejecución. En java ese hilo es el que ejecuta el método **`main`**. Sin embargo, si queremos que en nuestro programa varias tareas se ejecuten simultáneamente necesitaremos varios hilos de ejecución que se ejecuten de forma concurrente. A diferencia de los procesos concurrentes que, generalmente, son independientes entre sí, los hilos del mismo proceso comparten datos y espacio de direcciones, lo que facilita la transición entre hilos.

Java ofrece varias posibilidades para crear un hilo de ejecución. La más sencilla es declarar una subclase de la clase **`Thread`**[1]. Cada objeto de esta subclase permite lanzar un nuevo hilo que se ejecutará en paralelo con los ya existentes. La sintaxis para crear una nueva subclase a partir de la clase **`Thread`** es:

```
class Hilo extends Thread
```

La clase **`Thread`** (y por extensión la subclase creada a partir de ella) siempre incluye un método **`run()`** que contiene el código que debe ejecutarse concurrentemente con el resto del programa. También incluye un constructor en el que se puede incluir el código necesario para inicializar el hilo tras crearlo.

Por otro lado, es posible particularizar el comportamiento de cada objeto instanciado de la clase **`Hilo`** mediante atributos propios de la clase. Dichos atributos se

[1] Esta clase pertenece al paquete java.lang.

declaran antes del constructor de la clase, pueden inicializarse con los parámetros de dicho constructor y emplearse en **`run()`**.

Poniendo juntos todos estos detalles, nuestra nueva clase tendrá un aspecto parecido a:

```
class Hilo extends Thread {
  Socket cliente; //atributo de la clase
  public Hilo(Socket s) { // constructor de la clase
    cliente = s; // Código a ejecutar durante la inicialización
 }
 public void run() {
     // Código del hilo
     // Aquí ponemos lo que queremos que ejecute cada hilo
     //En nuestro caso el diálogo con el cliente
     Scanner entrada= new Scanner(cliente.getInputStream());
     ...
 }
}
```

Con relación al método **`run()`**, es importante destacar que no es posible modificar el prototipo de la función, y que por tanto no es posible modificar su cabecera para añadir la propagación (**`Throws`**) de excepciones. Esto implica la necesidad de tratar las excepciones dentro del propio **`run()`**mediante clausulas **`try/catch`**. Tampoco es posible pasar parámetros en la llamada al método.

Después de haber visto cómo se declara una subclase de la clase **`Thread`,** vamos a ver cómo utilizarla. El primer paso es crear un objeto de la subclase nueva, en nuestro ejemplo, la clase **`Hilo`**. Por ejemplo:

```
...
Hilo h=new Hilo ();
...
```

La instrucción anterior crea el hilo, pero no lo arranca. Es simplemente un **`Thread`** vacío que no tiene asignados aún recursos del sistema. Por lo tanto, el paso siguiente es “arrancar” el hilo, es decir, comenzar la ejecución en paralelo del código correspondiente. Esto se hace invocando el método **`start()`**, que creará un contexto para el hilo, asignando al hilo los recursos necesarios y comenzando su ejecución. El método **`start()`** a su vez ejecutará el código incluido en el método **`run()`**de forma concurrente con el resto de **`Threads`** en el sistema. Por lo tanto, para crear un hilo y ponerlo en ejecución necesitamos un fragmento de código similar al siguiente:

```
...
Hilo h=new Hilo ();
h.start();
...
```

Además, como hemos visto antes **`Hilo`** debe declararse como una clase descendiente de la clase **`Thread`** y hay que implementar su método **`run()`**.

Para familiarizarte con el funcionamiento de los hilos, vas a realizar un ejercicio que te muestre cómo funcionan los hilos. **Atención: el ejercicio NO implementa un cliente ni un servidor. Por lo tanto, NO empleará objetos de tipo `Socket`.**

Ejercicio 1:

Construye un programa Java que, mediante una clase que herede de **`Thread`**, lance tres hilos de ejecución. Cada uno de ellos mostrará por pantalla diez veces un texto, que se le pasará como parámetro en el constructor. Entre impresiones a pantalla, cada hilo deberá esperar 1 segundo mediante la orden **`sleep(1000)`**.

Como hemos visto en este ejercicio y también en el ejemplo de la clase **`Hilo`**, una posibilidad para pasar un objeto al nuevo hilo que arrancamos es hacerlo a través de un parámetro definido en el constructor de la clase. Esta idea nos resultará muy útil a la hora de elaborar nuestro servidor concurrente.

2. Servidores Concurrentes

La característica principal de un servidor concurrente es su capacidad de atender a varios clientes simultáneamente. Esto se puede implementar – no es la única forma – manteniendo un hilo de servicio para cada uno de los clientes que han conectado con el servidor.

El diseño básico de un servidor concurrente, por lo tanto, podrá utilizar varios hilos. En el hilo principal, el servidor permanecerá a la espera de que se conecte algún cliente. Cuando se reciba una petición de conexión TCP, el servidor aceptará la conexión y creará un nuevo hilo para atenderla. Por lo tanto, en este punto el programa servidor dispondrá de varios hilos. El que está atendiendo al cliente y el hilo principal del servidor que permanece a la espera de recibir las peticiones de nuevos clientes. Cuando el cliente finalice la conexión, el hilo que lo estaba atendiendo también termina. Ten en cuenta que el servidor puede conectarse con un nuevo cliente antes de finalizar con el cliente anterior, por lo que en un instante dado el servidor puede estar utilizando dos, tres, cuatro o más hilos.

Suponiendo que disponemos de una clase derivada de la clase *Thread* denominada **`Servicio(Socket s)`**, que atiende al cliente a través del socket **`s`** que se le pasa como parámetro, el hilo principal tendrá un aspecto parecido al siguiente:

```
public class ServidorConcurrente {
   public static void main(String argv[]) throws
                          UnknownHostException, IOException {
       int puerto=8000; //puerto bien conocido del servidor
       ServerSocket servidor=new ServerSocket(puerto);
       while (true) {
         Socket cliente=servidor.accept(); //Espero un cliente
          // Para atender la petición se crea un objeto Servicio
          // Se ejecuta el constructor sobre el socket "cliente"
         Servicio Cl=new Servicio(cliente);
          // Y se arranca el hilo para atender al cliente en paralelo
         Cl.start();
         } //Fin While
     } // Fin Main
} //Fin ServidorConcurrente
```

La clase **`Servicio`** se definiría como hemos visto en el apartado anterior, extendiendo la clase **`Thread`**. En el método **`run()`** de la clase **`Servicio`** pondríamos las instrucciones necesarias para realizar el diálogo con el cliente y prestarle el servicio deseado.

Ejercicio 2:

Construye un servidor concurrente de **eco** TCP. El servidor debe devolver al cliente cada línea de texto que el cliente le envíe, hasta que la cadena recibida sea "FIN". Tras devolver el "FIN" recibido, el servidor cerrará la conexión con el cliente.

Implementación:

Para poder atender a varios clientes a la vez, el servidor debe emplear varios hilos. El hilo principal, que ejecuta el procedimiento **`main`**, esperará clientes en el puerto 7777. Cada vez que se establezca conexión con un cliente TCP se lanzará un nuevo hilo de servicio, al cual se le pasará como parámetro el socket conectado al cliente. Utilizando este socket, el nuevo hilo se encargará de realizar el diálogo con el cliente, hasta que la cadena recibida sea **FIN**. En ese momento, el servidor cerrará la conexión con el cliente y finalizará la ejecución del hilo auxiliar.

Puedes probar tu servidor ejecutando la orden **`nc localhost 7777`**

NOTA: el método **`equalsIgnoreCase()`** de la clase **`String`** te permite comparar dos objetos de la clase **`String`** sin tener en cuenta mayúsculas y minúculas.

3. Otros usos de la concurrencia

La posibilidad de lanzar hilos dentro de un mismo código facilita la implementación de otros tipos de servidores, como son los servidores multiservicio y los servidores multiprotocolo.

Un servidor multiservicio es capaz de aceptar clientes a través de diversos puertos, ofreciendo diferentes servicios por cada uno de ellos. Por ejemplo, un servidor multiservicio puede proporcionar tanto los servicios de ***daytime***, que devuelve la hora del sistema como un **`String`**; como ***time***, que devuelve la hora del sistema en segundos desde el 1 de enero de 1900. Estos servicios se ofrecerían a través de dos puertos distintos (los puertos estándar son 13 y 37, respectivamente. Puedes probarlos en time.nist.gov). Una variante consiste en los servidores multiprotocolo, que ofrecen el mismo servicio sobre dos protocolos de transporte, generalmente TCP y UDP. Un ejemplo típico de multiprotocolo es un servidor de eco ofreciendo el servicio sobre TCP y sobre UDP.

Algunos **clientes** también requieren concurrencia para su implementación. Por ejemplo, supongamos un cliente de chat. Tras la conexión del usuario a un servidor de chat, el programa cliente debe estar pendiente a la vez de dos posibilidades. Por un lado, el cliente debe recoger el texto introducido por el usuario por teclado y enviarlo al servidor. Por otro lado, el cliente también debe esperar los mensajes de otros usuarios que le envía el servidor de chat y visualizarlos en pantalla. La solución más sencilla consiste en implementar ambas funcionalidades en hilos concurrentes, que permitan atender ambas situaciones. Ten en cuenta que las operaciones de lectura son llamadas bloqueantes. Por lo tanto, cuando ejecutas una sentencia que lee del **`System.in`** o que lee de un socket la ejecución del programa se detiene en esa línea hasta que se reciban datos para leer. Por ello la lectura de la entrada de teclado y de la entrada de socket no pueden estar en un mismo hilo de ejecución, porque se bloquearán mutuamente.

Ejercicio 3:

Escribe un programa **cliente** de chat en java que se conecte al servidor y puerto **TCP** que indicará el profesor. El programa requerirá al menos un hilo adicional al que ejecuta el procedimiento **`main`**. Al disponer de dos hilos de ejecución, uno de ellos se utilizará para leer del teclado los datos que el usuario introduce y enviarlos a través del socket, y el otro hilo para leer los datos que llegan a través del socket y visualizarlos en pantalla. El cliente estará en ejecución hasta que el usuario teclee la orden "quit" que se transmitirá al servidor antes de terminar el programa cliente.

Nota: Recuerda que la clase **`Scanner`** tiene disponible el método **`hasNext()`** que devolverá **`false`** cuando detecte el cierre del objeto **`Socket`**. Este método puede ayudarte a salir de forma elegante del bucle que lee las líneas que envía el servidor.

Ayuda para la implementación:

Se ha supuesto que ya existe un socket conectado con el servidor. . Es conveniente que uno de los 2 hilos sea el del procedimiento **`main`**, aunque no es imprescindible. El otro hilo habrá que crearlo.

Hilo A

…

Crear **`Scanner`** asociado al flujo de entrada del **`Socket`**.

Mientras seguir {

Leer del **`Socket`**.

Imprimir en pantalla.

}

Hilo B

…

Crear **`PrintWriter`** asociado al flujo de salida del socket.

Definir **`Scanner`** asociado al teclado.

Mientras ¡quit {

Leer línea del teclado.

Enviar línea leída al servidor. (Recuerda hacer el **`flush`**).

}

Cerrar el **`Socket`**.