Práctica 5: Servidores concurrentes en Java

Esta práctica pretende familiarizar al alumno 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 diferentes tipos de servidores concurrentes.

Al acabar la práctica el alumno debería ser capaz de:

1) Escribir un programa básico en Java que haga uso de *threads* con el objetivo de realizar varias tareas de forma concurrente.
2) Conocer 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 ofreciera el mismo servicio, pero a varios clientes a la vez.

1. Implementación de la concurrencia

Java ofrece varias posibilidades para crear un hilo de ejecución. La más sencilla es declarar una subclase de la clase **`Thread`**. 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 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:

Boletín realizado por Alberto Bonastre, Elvira Baydal, y Federico Silla. Cualquier errata o sugerencia, por favor hacédsela llegar a los autores ({bonastre, elvira, fsilla}@disca.upv.es).

```
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
     Scanner entrada= new Scanner(cliente.getInputStream());
     ...
 }
}
```

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**. A continuación es necesario "arrancar" el hilo, es decir, comenzar la ejecución en paralelo del código correspondiente. Esto se hace invocando el método **start()**, que a su vez ejecutará el código incluido en el método **run()** de forma concurrente con el resto de *Threads* en el sistema. Esto se puede ver en el siguiente fragmento de código:

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

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 pasarle parámetros en la llamada al método.

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 100 milisegundos mediante la orden **sleep(100)**.

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 del servidor, 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 mientras el hilo principal del servidor queda a la espera de recibir las peticiones de nuevos clientes.

Suponiendo que disponemos de una clase derivada de *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
```

Ejercicio 2:

Construye un servidor concurrente de **Echo** TCP. El hilo principal esperará clientes en el puerto 7777. Cada vez que llegue un cliente TCP se lanzará un nuevo hilo de servicio el cual se le pasará como parámetro el socket conectado al cliente. Este hilo devolverá

indefinidamente al cliente las líneas que éste le envía a través del *socket*, hasta que la cadena recibida sea **FIN**. En ese momento, el servidor cerrará la conexión con el cliente y finalizará la ejecución del hilo.

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 multiservicio y los 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 *string*, como ***time***, que devuelve la hora del sistema en segundos desde el 1 de enero de 1900, 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.

Algunos clientes también requieren concurrencia, especialmente cuando no se sigue un protocolo de aplicación estricto. 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.

Ejercicio 3:

Escribe un programa cliente de chat en java que se conecte al servidor y puerto que indicará el profesor. El programa requerirá al menos un hilo adicional al que ejecuta el procedimiento `main`. Al disponer de dos hilos, 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.