Sistemas Distribuidos

Java Sockets y Threads (Conectores e Hilos)

Tabla de Contenidos

1. Introducción

Para introducirnos en tema, primeramente haremos una breve revisión de conceptos para los Protocolos TCP/IP, en relación con la abstracción que proporciona la API sockets para el protocolo TCP, originaria de UNIX BSD 4.x.

1.1 Streams TCP

La mencionada API, proporciona la abstracción de flujo de bytes **(stream),** en el que pueden escribirse y leerse datos. Esta abstracción de stream, oculta las siguientes características de la red:

- **Tamaño de los mensajes:** La aplicación puede elegir la cantidad de datos que se quiere escribir o leer del stream. El conjunto de datos puede ser muy pequeño o muy grande. La implementación del flujo TCP subyacente, decide cuántos datos recoge antes de transmitirlos como uno o más paquetes IP. En el destino, los datos son proporcionados a la aplicación según los va solicitando. Las aplicaciones pueden forzar, si fuera necesario, que los datos sean enviados de forma inmediata.
- **Mensajes perdidos**: El protocolo TCP utiliza un esquema de acuse de recibo de los mensajes. Como un ejemplo de un esquema simple (no utilizado por TCP), el extremo emisor almacena un registro de cada paquete IP enviado y el extremo receptor acusa el recibo de todos los paquetes IP que le llegan. Si el emisor no recibe dicho acuse de recibo dentro de un plazo de tiempo fijado, volverá a transmitir el mensaje. El esquema de desplazamiento de ventana, más sofisticado, reduce drásticamente el número de mensajes de reconocimiento necesarios.
- **Control del flujo:** El protocolo TCP intenta ajustar las velocidades de los procesos que leen y escriben en un stream. Si el escritor es demasiado rápido para el lector, entonces será bloqueado hasta que el lector haya consumido una cantidad suficiente de datos.
- **Duplicación y ordenamiento de los mensajes:** A cada paquete IP se le asocia un identificador, que hace posible que el receptor pueda detectar y rechazar mensajes duplicados, o que pueda reordenar los mensajes que lleguen desordenados.
- **Destinos de los mensajes:** Un par de procesos en comunicación establecen una conexión antes de que puedan comunicarse mediante un stream. Una vez que se ha establecido la comunicación, los procesos simplemente leen o escriben en el stream, sin tener que preocuparse de direcciones Internet, ni de los números de puerto. El establecimiento de la conexión implica una petición de conexión: **connect**, desde el cliente al servidor, seguida de una aceptación: **accept**, desde el servidor al cliente antes de que cualquier comunicación pueda tener lugar. Esto puede suponer una sobrecarga considerable para una única petición y una única respuesta.

El API para la comunicación por streams, supone que en el momento de establecer una conexión entre dos computadoras de la red, una de ellas juega el papel de cliente y la otra juega el papel de servidor, aunque después se comuniquen de igual a igual. El rol de cliente implica la creación de un conector **(socket)**, de tipo stream, sobre cualquier puerto y la posterior petición de conexión con el servidor en su puerto de servicio. El papel del servidor involucra la creación de un conector de escucha ligado al puerto de servicio y a la espera de clientes que soliciten conexiones. El conector mantiene una cola de peticiones de conexión. En el modelo de sockets, cuando un servidor acepta una conexión, crea un nuevo conector para mantener la comunicación con el cliente, mientras que se reserva el conector del puerto de servicio, para escuchar las peticiones de conexión de otros clientes.

El par de conectores, el del cliente y el del servidor, se conectan por un par de streams, uno en cada dirección. Así, cada conector tiene su propio stream de entrada y de salida. Uno de los procesos del par, puede enviar información al otro escribiendo en su stream de salida y el otro, puede obtener la información leyendo de su stream de entrada.

Cuando una aplicación cierra un conector, indica que no va a escribir nada más en su stream de salida. Cualquier dato que se encuentre en su buffer de salida, será enviado al otro extremo del stream y puesto en la cola del conector destino con una indicación de que el stream ha sido roto. El proceso en el destino puede leer los datos en la cola, pero cualquier lectura después de que la cola

esté vacía resultará en una indicación de final de stream. Cuando un proceso finaliza su ejecución o falla, todos sus conectores se cierran y cualquier proceso que intente comunicarse con él descubrirá que la conexión se ha roto.

Los siguientes aspectos, relacionados con la comunicación de streams, son importantes:

- **Concordancia de ítems de datos:** Los dos procesos que se comunican necesitan estar de acuerdo en el tipo de datos transmitidos por el stream. Por ejemplo, si un proceso escribe un int seguido de un double, el proceso receptor debe interpretarlo como un int seguido de un double. Cuando un par de procesos no coopera correctamente en el uso del stream, el proceso lector puede encontrarse con problemas cuando interprete los datos o puede bloquearse, debido a que se encuentre una cantidad insuficiente de datos en el stream. Veremos entonces más adelante, la necesidad de efectuar el aplanado (***Marshalling***) de los datos previo a su envío, como así mismo el proceso inverso del aplanado (***Unmarshalling***) en la recepción.
- **Bloqueo:** Los datos escritos en un stream se almacenan en un buffer en el conector destino. Cuando un proceso intenta leer datos de un canal de entrada, o bien extraerá los datos de la cola o bien se bloqueará hasta que existan datos disponibles. El proceso que escribe los datos en el stream, puede llegar a ser bloqueado por el mecanismo de control de stream de TCP, si el conector del otro lado intenta almacenar en la cola de entrada más información de la permitida.
- **Hilos (Threads):** Cuando un servidor acepta una conexión, puede crear un nuevo hilo de ejecución para comunicarse con el nuevo cliente. La ventaja de utilizar un hilo separado para cada cliente, es que el servidor puede bloquearse a la espera de entradas sin afectar a los otros clientes. Más adelante se brindará un concepto más acabado de esta capacidad de procesamiento multitarea de los sistemas operativos actuales

Comportamiento ante fallo

Para satisfacer la propiedad de **integridad** de una comunicación fiable, como se enunció anteriormente, TCP utiliza una suma de comprobación para detectar y rechazar los paquetes corruptos y un número de secuencia para detectar y eliminar los paquetes duplicados. Con respecto a la propiedad de **validez**, los streams TCP utilizan timeouts y retransmisión de los paquetes perdidos, por lo tanto, los mensajes tienen garantizada su entrega cuando alguno de los paquetes originales se haya perdido.

Ahora bien, si la pérdida de paquetes sobrepasa un cierto límite, o la red que conecta un par de procesos en comunicación está severamente congestionada, el software TCP responsable de enviar los mensajes no recibirá acuses de recibo de los paquetes enviados y después de un tiempo declarará cerrada la conexión.

Cuando una conexión está rota, se notificará al proceso que la utiliza siempre que intente leer o escribir. Esto tiene los siguientes efectos:

a) Los procesos que utilizan la conexión no distinguen entre un fallo en la red y un fallo en el proceso que está en el otro extremo de la conexión.

b) Los procesos que se comunican no pueden saber inmediatamente si sus mensajes recientes han sido recibidos o no.

Utilización de TCP

Muchos de los servicios utilizados se ejecutan sobre conexiones TCP, con números de puerto reservados. Entre ellos se encuentran los siguientes:

HTTP: El protocolo de transferencia de hipertexto se utiliza en comunicación entre un navegador y un servidor web.

FTP: El protocolo de transferencia de archivos permite leer los directorios de una computadora remota y transferir archivos entre las computadoras de una conexión.

Telnet: La herramienta Telnet proporciona acceso a un terminal en una computadora remota.

SMTP: El protocolo simple de transferencia de correo se utiliza para enviar correo electrónico entre computadoras.

2. Conectores (Sockets)

Las formas de comunicación UDP y TCP, utilizan la abstracción de sockets, que proporciona los puntos extremos de la comunicación entre procesos. Los sockets se originan en UNIX BSD aunque están presentes en la mayoría de las versiones de UNIX, Linux, Windows y Macintosh OS. La comunicación entre procesos consiste en la transmisión de un mensaje entre un conector de un proceso y un conector de otro proceso, según se muestra en la Figura 1.

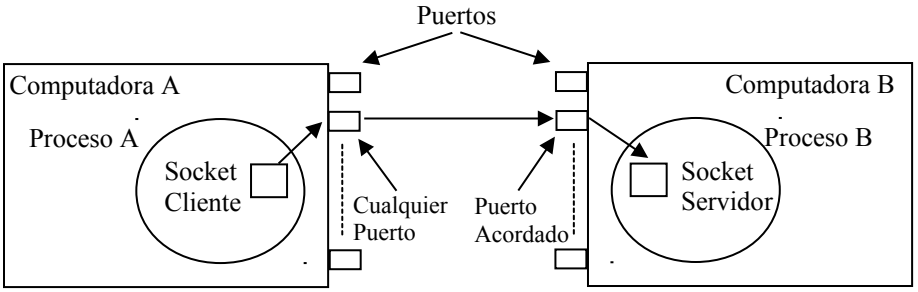

Figura 1. Sockets y puertos.

El conector de cada proceso debe estar asociado a un puerto local y a la dirección Internet de la computadora donde se ejecuta. Los mensajes enviados a una dirección de Internet y a un número de puerto concretos, sólo pueden ser recibidos por el proceso cuyo conector esté asociado con esa dirección y con ese puerto. Los procesos pueden utilizar un mismo conector tanto para enviar como para recibir mensajes.

La biblioteca/definición de sockets permite un gran número de puertos posibles (2^{16}), que pueden ser usados por los procesos locales para comunicación de mensajes. Por su parte, cada proceso puede utilizar varios puertos para recibir mensajes, pero un proceso no puede compartir puertos con otros procesos de la misma computadora. No obstante, cualquier cantidad de procesos pueden enviar mensajes a un mismo puerto. Cada conector se asocia con un protocolo concreto, que puede ser UDP o TCP.

2.1. API de Java para las direcciones Internet

Como los paquetes IP que subyacen a TCP y UDP se envian a direcciones Internet (Dirección IP) , Java proporciona una clase, **InetAddress**, que representa las direcciones Internet. Los usuarios de esta clase se refieren a las computadoras por sus nombres de host en el Servicio de Nombres de Dominio (Domain Name Service, DNS). Por ejemplo, se pueden crear instancias de InetAddress que contienen direcciones de Internet invocando a un método estático de InetAddress con un nombre DNS de host como argumento. El método utiliza DNS para conseguir la correspondiente dirección Internet. Por ejemplo, para conseguir un objeto que represente la dirección Internet de un host cuyo nombre DNS es alfa.unp.edu.ar, se debería utilizar:

InetAddress unaComputadora = InetAddress.getByName(“alfa.unp.edu.ar”);

Como se verá en el ejemplo dado más adelante, se consigna explícitamente la dirección IP o el nombre de red local de la computadora destino (en este caso la computadora del archivo **Hosts,** que en caso de Windows, otorga las equivalencias de Nombre – Dirección IP dentro de una LAN.

2.2. API Java para los Streams TCP

Clases ServerSocket y Socket

ServerSocket: Esta clase está diseñada para ser utilizada por un servidor, con el fin de crear un conector que, en el puerto de servidor, escucha las peticiones de conexión de los clientes. Su método: **accept** toma una petición **connect** de la cola, o si la cola está vacía, se bloquea hasta que llega una petición. El resultado de ejecutar accept es una instancia de Socket: un conector que da acceso a streams para comunicarse con el cliente.

Socket: Esta clase es utilizada por el par de procesos de una conexión. El cliente utiliza un constructor para crear un conector, especificando el nombre del host o su dirección IP y el puerto del servidor. Este constructor no sólo crea el conector asociado con el puerto local, sino que también se conecta con la computadora remota y el puerto indicados. Puede lanzar (*throw*) una excepción UnknownHostException si el nombre de host no es correcto, o una excepción IOException, si se da un error de entrada/salida.

La clase Socket proporciona los métodos getInputStream y getOutputStream para acceder a los dos streams asociados con un conector. El tipo de datos devueltos por esos métodos son InputStream y OutputStream, respectivamente (clases abstractas que definen métodos para leer y escribir bytes). Los valores de retorno se pueden utilizar como argumentos de constructores para gestionar adecuadamente los streams de entrada y de salida del programa.

Nuestro ejemplo más adelante, utiliza DataInputStream y DataOutputStream, los cuales permiten utilizar representaciones binarias de los tipos primitivos de datos para ser leídos y escritos de una forma independiente de la máquina.

Clases DataInputStream y DataOutputStream

Como se ha dejado trascender, en Java, la manera de transferir información a las entradas y salidas es a base de flujos de bytes o streams. Un stream es la conexión entre un proceso y la fuente o el destino de los datos. La información se traslada en serie (un dato a continuación de otro) a través de esa conexión. Esto permite una forma general de representar un variado tipo de comunicaciones. Por ejemplo, si se quiere imprimir algo en la pantalla, se lo hace a través de un stream que conecta el proceso con el dispositivo de salida. Esto es, se da al stream la orden de escribir y éste lo traslada hacia el monitor. El concepto es tan general, como para representar la lectura y escritura en archivos de un disco o la comunicación de computadoras a través de una red.

El Package **java.io** contiene las clases necesarias para la comunicación de un proceso con el exterior. Dentro de ese package existen dos familias de jerarquías distintas para la entrada y salida de datos. La diferencia principal es que una de ellas opera con bytes, mientras que la otra lo hace con caracteres (Los caracteres en Java se representan con dos bytes pues siguen la codificación Unicode).

La entrada y salida de datos del proceso se puede hacer con clases derivadas de **InputStream** (lectura) y **OutputStream** (escritura) estas clases tienen los métodos básicos **read() y write()**. En la figura 2, se representan las clases que, entre otras, derivan de las mencionadas.

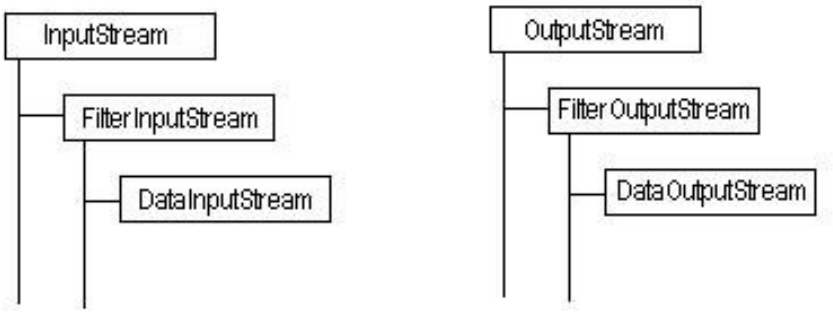

Figura 2. Herencia de Clases en Java.

Las clases **DataInputStream** y **DataOutputStream** que derivan de InputStream y OutputStream respectivamente, si bien aparecieron con la versión Java 1.0, tienen recomendado su uso en situaciones especiales ante el uso de determinadas tareas como ser la serialización y compresión de datos. Estas clases se utilizan para leer y escribir datos en los formatos propios de Java, que los hacen independientes de las plataformas donde corran esos programas. Debido a ello, se las utiliza especialmente para transmisiones de datos entre computadoras. Estas clases poseen los métodos básicos como read() o write() para lectura y escritura de bytes, como también poseen los métodos **readUTF()** y **writeUTF()** que poseen un uso más específico, como se indica en la próxima sección.

2.3 Representación Externa de Datos y Aplanado (Marshalling)

La información almacenada dentro de los procesos se representa mediante estructuras de datos, mientras que la información transportada en los mensajes consiste en secuencias de bytes. Independientemente de la forma de comunicación utilizada, las estructuras de datos deben ser "*aplanadas*" (convertidas a una secuencia de bytes antes de su transmisión y reconstruidas en el destino). Los tipos de datos primitivos transmitidos en los mensajes pueden tener valores de muchos tipos distintos, y no todas las computadoras almacenan los valores primitivos, tales como los bytes correspondientes a enteros, en el mismo orden. También es diferente en diferentes arquitecturas la representación de los números en punto flotante. Otro problema es el conjunto de códigos utilizado para representar los caracteres: por ejemplo, UNIX utiliza la codificación ASCII con un byte por carácter, mientras que el estándar Unicode permite representar caracteres particulares de la mayoría de los lenguajes y utiliza dos bytes por carácter. Como es conocido, también existen dos variantes en la ordenación de bytes que corresponden a enteros: la llamada big-endian, en la que el byte más significativo va el primero, y la llamada little-endian, en la que va último.

Hay que hacer notar, sin embargo, que a pesar del "aplanamiento", los bytes no son alterados durante la transmisión. Para soportar RMI o RPC, cualquier tipo de dato que pueda ser pasado como un argumento o devuelto como resultado debe ser capaz de ser aplanado y cada uno de los tipos de datos primitivos representados es una representación de datos acordada. Al estándar acordado para la representación de estructuras de datos y valores primitivos se denomina **representación externa de datos.**

El aplanado (*marshalling*) consiste en tomar una colección de ítems de datos y ensamblarlos de un modo adecuado para la transmisión en un mensaje. El proceso inverso del aplanado (*unmarshalling*) es el proceso de desensamblado en el destino para recuperar la colección de datos original.

Por lo tanto, aplanar consiste en traducir las estructuras de datos y los valores primitivos en una representación externa de datos. Similarmente, el proceso inverso del aplanado consiste en generar los valores originales desde la representación de datos externa y reconstruir las estructuras de datos.

Serialización de Objetos Java

En Java, se denomina serialización de objetos Java a aquello que está relacionada con el aplanado y la representación externa de datos de cualquier objeto simple o un árbol de objetos que tienen que ser transmitidos en un mensaje o almacenados en disco. Es de uso exclusivo de Java.

En ambos casos, el aplanado y su proceso inverso son llevados a cabo por una capa de **middleware** sin ninguna participación del programador de la aplicación. Debido a que el aplanado requiere la consideración de todos los detalles de bajo nivel de la representación de los componentes primitivos de los objetos compuestos, el proceso es propenso a fallar cuando lo hace el programador de aplicaciones. La eficiencia es otro criterio que resulta deseable en el diseño de los procedimientos de aplanado generados de forma automática.

Aunque estamos interesados en el uso de aplanado para los argumentos y para los resultados de los cliente/servidor, es viable decir que tienen un uso más general en la conversión de las estructuras de datos u objetos, en una forma adecuada para la transmisión en mensajes o para su almacenamiento en archivos.

En particular, los métodos readUTF() y writeUTF() convierten el formato aplanado de Java UTF-8 a un string y viceversa.

2.4. Aplicación ejemplo Cliente / Servidor con Streams TCP

Aunque de manera sencilla, se presenta a continuación un ejemplo *completo* de una aplicación cliente/servidor que uitiliza Java sockets TCP para las comunicaciones y los métodos readUTF() y writeUDF() para la transmisión de strings.

Código del Cliente TCP

Se presenta el programa de un cliente TCP que realiza una conexión a un servidor, envía una petición

y recibe una respuesta. El listado muestra el programa de un cliente donde se le da como argumento al método main el nombre del servidor (o su dirección IP) y un mensaje. El cliente crea un conector ligado al nombre del host y al puerto 7896, mediante la declaración de una nueva instancia de la clase Socket que ya se describió. Obtiene DataInputStream y DataOutputStream de los streams de los conectores de entrada y de salida respectivamente, mediante la declaración de instancias de las clases respectivas. Luego escribe el mensaje en su stream de salida y se bloquea a la espera de leer la respuesta en el stream de entrada. Como nuestro mensaje está compuesto por una cadena de caracteres, el cliente (y como se verá luego también el servidor) utilizan el método writeUTF() de DataOutputStream para escribirlo en el stream de salida, y el método readUTF() de DataInputStream para leerlo del stream de entrada.

```
/*
 * Cliente TCP
 */
import java.net.*;
import java.io.*;

public class ClienteTCP {
        public static void main(String args[])  {
           // args proporciona el Nombre/IP de Server Destino y el mensaje
           if (args.length != 2)  {
                System.out.println("2 argumentos: servidor y mensaje");
                System.exit(1);
           }

           try {
                int puertoServicio = 7896;

                Socket s = new Socket(args[0], puertoServicio);
                DataInputStream entrada = new DataInputStream(s.getInputStream());
                DataOutputStream salida = new DataOutputStream(s.getOutputStream());
                salida.writeUTF(args[1]);
                String datos = entrada.readUTF();
                System.out.println("Recibido: " + datos);
                s.close();
           }
           catch( Exception e) {
                e.printStackTrace();
           }
        }
}
```

Código del Servidor TCP

Se muestra el código fuente de un servidor TCP que espera una conexión de un cliente, lee una petición y envía un eco como respuesta. El programa de servidor:

1) Abre un conector de servidor en su puerto de servicio (7896) con

```
ServerSocket escuchandoSocket = new ServerSocket(puertoServicio);
```

2) Escucha las peticiones de conexión con

```
Socket socketCliente = escuchandoSocket.accept();
```

3) Cuando llega una petición, crea DataInputStream y DataOutputStream de los streams de los conectores de entrada y de salida de su conector con

```
DataInputStream entrada;
DataOutputStream salida;
entrada = new DataInputStream(socketCliente.getInputStream());
salida = new DataOutputStream(socketCliente.getOutputStream());
```

4) Espera a leer el mensaje del cliente y lo devuelve.

Cuando un proceso ha cerrado su conector, no le será posible utilizar sus streams de entrada y de salida. El proceso al que le ha enviado datos puede leerlos de su cola, pero cualquier lectura después de que la cola esté vacía resultará en una excepción EOFException. Los intentos de utilizar un conector cerrado o de escribir en un stream roto tienen como resultado una excepción IOException.

```
/*
 * ServidorTCP
 * un cliente a la vez, sin concurrencia
 * conexiones por el port 7896 (hardcoded)
 */

import java.net.*;
import java.io.*;

public class ServidorTCP {
	public static void main(String args[])  {
		try {
			int puertoServicio = 7896;
			ServerSocket escuchandoSocket = new ServerSocket(puertoServicio);
			while (true) {
				/* Espera conexion de un cliente */
				Socket socketCliente = escuchandoSocket.accept();

				/* Ya hay una conexion con un cliente, streams de I/O */
				DataInputStream entrada;
				DataOutputStream salida;
				entrada = new DataInputStream(socketCliente.getInputStream());
				salida = new DataOutputStream(socketCliente.getOutputStream());

				/* Provee servicio e imprime */
				String datos = entrada.readUTF();
				salida.writeUTF(datos);
				System.out.println("Emitido: " + datos);

				/* Fin de un servicio */
				socketCliente.close();
			}
		}
		catch(Exception e) {
			e.printStackTrace();
		}
	}
}
```

3. Hilos (Threads)

Los sistemas operativos actuales permiten el **procesamiento multitarea**. Esto es, la aparente realización simultánea de dos o más actividades. Cuando un sistema Operativo realiza más de un proceso en modo multitarea con aparente simultaneidad, se dice que estos procesos son concurrentes. El sistema operativo posee un planificador que es el que otorga el quantum o tiempo de ejecución a los distintos procesos, repartiendo la actividad de la CPU.

Un proceso es un programa ejecutándose con un espacio propio de memoria. Un hilo o thread es un flujo secuencial simple dentro de un proceso. Un mismo proceso puede tener varios hilos concurrentes en ejecución. Si un hilo dentro de un proceso está bloqueado a la espera de la entrada de un dato, no implicará que se bloqueen los hilos restantes del proceso en cuestión. Sin el uso de threads hay tareas que serían prácticamente imposibles de ejecutar, especialmente aquellas que poseen tiempos de espera importantes entre etapas.

En Java hay dos formas de crear threads. La más simple y sobre la que se otorgará un ejemplo más adelante, consiste en crear una clase que hereda de la clase **java.lang.Thread,** sobrecargando su método **run().** La otra forma consiste en declarar una clase que implemente la interface **java.lang.Runnable.** Este último mecanismo más complejo, sobre el que no daremos aquí más detalles, posee la ventaja que la clase declarada no debe heredar necesariamente de Threads, con lo cual es posible hacerlo heredar de otra clase. Esto es necesario por ejemplo en el caso de una **Applet**, clase que siempre debe heredar de la superclase **java.applet.Applet.**

A continuación se presenta un ejemplo de la creación de Threads con el primer método citado, derivando de la clase Threads:

```
/*
 * SimpleThread.  Un ejemplo muy simple de Hilos.
 */
public class SimpleThread extends Thread {
        private int countDown = 5;
        private static int threadCount = 0;
        private int threadNumber = ++threadCount;
        public SimpleThread() {
                System.out.println("Haciendo el hilo: " + threadNumber);
        }

        public void run() {
                while(true) {
                        System.out.println("Hilo: " + threadNumber + "(" + countDown + ")");
                        try {
                            Thread.sleep(1000);
                        }    catch  (InterruptedException ie)  {
                                System.out.println(ie.getMessage());       }

                        if(--countDown == 0) return;
                }
        }

        public static void main(String[] args) {
                for(int i = 0; i < 5; i++)
                        new SimpleThread().start();
                System.out.println("Todos los Hilos fueron creados");
        }
}
```

Como puede observarse, se ha creado la clase SimpleThread que hereda de Thread. Su método más importante es run() que es sobrescrito para que el método haga lo que se desea. Run() es el método que se ejecutará en forma concurrente para cualquiera de los hilos del proceso.

El ejemplo crea 5 hilos de los que se realiza un seguimiento asignando a cada hilo un número único, generado con una variable static. El método run() de Thread, se sobrescribe para que disminuya cada vez que pase por el bucle for y acabe cuando valga 0. En el momento que acabe run(), acaba el hilo respectivo.

A continuación, como otro ejemplo de utilización de Threads, se presenta el código java de un Servidor TCP que establece conexiones concurrentes de clientes, otorgándole a cada uno de los clientes un hilo dentro del proceso Servidor.

Código del Servidor TCP

El listado de código Java presenta un servidor TCP que establece conexiones concurrentes de clientes, lee sus peticiones y las reenvía en eco como respuesta. El programa del servidor abre un conector de servidor en su puerto de servicio (7896) y escucha las peticiones de conexión: accept. Cuando llega una petición, crea un nuevo hilo para comunicarse con el cliente. El nuevo hilo crea un *DataInputStream* y un *DataOutputStream* para los flujos de entrada y salida de su conector, y entonces espera a leer el mensaje del cliente. Recibido éste, lo escribe en retorno.

```
/*
 * Servidor TCP2
 * con threads (concurrencia de clientes)
 * conexiones por el port 7896 (hardcoded)
 */

import java.net.*;
import java.io.*;

public class ServidorTCP2 {
	public static void main(String args[]) {
		try {
			int puertoServicio = 7896;
			ServerSocket escuchandoSocket = new ServerSocket(puertoServicio);
			while (true) {
				/* Espera conexion de un cliente */
				Socket socketCliente = escuchandoSocket.accept();
				/* Crea un thread para el servicio al cliente */
				Conexion c = new Conexion(socketCliente);
			}
		}
		catch(Exception e) {
			e.printStackTrace();
		}
	}
}

class Conexion extends Thread {
	Socket socketCliente;
	DataInputStream entrada;
	DataOutputStream salida;
	public Conexion (Socket unSocketCliente) {
		try {
			/* Ya hay una conexion con un cliente, streams de I/O */
			socketCliente = unSocketCliente;
			entrada = new DataInputStream(socketCliente.getInputStream());
			salida = new DataOutputStream(socketCliente.getOutputStream());
			/* Al servicio */
			this.start();
		}
		catch(Exception e){
			e.printStackTrace();
		}
	}

	public void run()  {
		try {
			/* Provee servicio e imprime */
			String datos = entrada.readUTF();
			salida.writeUTF(datos);
			System.out.println("Emitido: " + datos);

			 /* Fin de un servicio */
			socketCliente.close();
		}
		catch(Exception e) {
			e.printStackTrace();
		}
	}
}
```

Ricardo López 2007 - 2013

4. Bibliografía

- G. Coulouris, J. Dollimore, T. Kindberg, Distributed Systems: Concepts and Design, 4th Edition, Addison Wesley, 2005, ISBN: 0321263545.
- B. Eckel, Thinking in Java (3rd Edition), Prentice Hall Ptr, 2006, ISBN: 0131002872.
- J. García de Jalón, J. I. Rodríguez, I. Mingo, A. Imaz, A. Brazález, A. Larzabal, J. Calleja, J. García, Aprenda Java como si estuviera en Primero, Universidad de Navarra, San Sebastián, 2000, disponible en http://www.tecnun.es/asignaturas/Informat1/ayudainf/aprendainf/Java/Java2.pdf
- Lesson: Basic I/O, http://java.sun.com/docs/books/tutorial/essential/io/, (The Java tutorials > Essential Classes)