1 SOCKETS EN JAVA

La programación en red siempre ha sido dificultosa, el programador debía de conocer la mayoría de los detalles de la red, incluyendo el hardware utilizado, los distintos niveles en que se divide la capa de red, las librerías necesarias para programar en cada capa, etc.

Pero, la idea simplemente consiste en obtener información desde otra maquina, aportada por otra aplicación software. Por lo tanto, de cierto modo se puede reducir al mero hecho de leer y escribir archivos, con ciertas salvedades.

El sistema de Entrada/Salida de Unix sigue el paradigma que normalmente se designa como Abrir-Leer-Escribir-Cerrar. Antes de que un proceso de usuario pueda realizar operaciones de entrada/salida, debe hacer una llamada a Abrir (`open`) para indicar, y obtener los permisos del fichero o dispositivo que se desea utilizar.

Una vez que el fichero o dispositivo se encuentra abierto, el proceso de usuario realiza una o varias llamadas a Leer (`read`) y Escribir (`write`), para la lectura y escritura de los datos.

El proceso de lectura toma los datos desde el objeto y los transfiere al proceso de usuario, mientras que el de escritura los transfiere desde el proceso de usuario al objeto. Una vez concluido el intercambio de información, el proceso de usuario llamará a Cerrar (close) para informar al sistema operativo que ha finalizado la utilización del fichero o dispositivo.

En Unix, un proceso tiene un conjunto de descriptores de entrada/salida desde donde leer y por donde escribir. Estos descriptores pueden estar referidos a ficheros, dispositivos, o canales de comunicaciones *sockets*.

El ciclo de vida de un descriptor, aplicado a un canal de comunicación (por ejemplo, un *socket*), está determinado por tres fases :

- Creación, apertura del *socket*
- Lectura y Escritura, recepción y envío de datos por el *socket*
- Destrucción, cierre del *socket*

La interface IPC en Unix-BSD está implementada sobre los protocolos de red TCP y UDP. Los destinatarios de los mensajes se especifican como direcciones de *socket*; cada dirección de *socket* es un identificador de comunicación que consiste en una dirección Internet y un número de puerto.

Las operaciones IPC se basan en pares de *sockets*. Se intercambian información transmitiendo datos a través de mensajes que circulan entre un *socket* en un proceso y otro *socket* en otro proceso. Cuando los mensajes son enviados, se encolan en el *socket* hasta que el protocolo de red los haya transmitido. Cuando llegan, los mensajes son encolados en el *socket* de recepción hasta que el proceso que tiene que recibirlos haga las llamadas necesarias para recoger esos datos.

El lenguaje Java fue desarrollado por la empresa Sun MicroSystems hacia el año 1990, mediante la creación de un grupo de trabajo en cuya cabeza estaba James Gosling. Este grupo de trabajo fue ideado para desarrollar un sistema de control de electrodomésticos y de PDAs o asistentes personales (pequeños ordenadores) y que además tuviese la posibilidad de interconexión a redes de ordenadores. Todo ello implicaba la creación de un hardware polivalente, un sistema operativo eficiente (SunOS) y un lenguaje de desarrollo (Oak). El proyecto concluyó dos años más tarde con un completo fracaso que condujo a la disolución del grupo.

Pero el desarrollo del proyecto relativo al lenguaje *oak* siguió adelante gracias entre otras cosas a la distribución libre del lenguaje por Internet mediante la incipiente, por aquellos años, World Wide Web. De esta forma el lenguaje alcanzó cierto auge y un gran número de programadores se encargaron de su depuración así como de perfilar la forma y usos del mismo.

El nombre de Java, surgió durante una de las sesiones de *brain storming* que se celebraban por el equipo de desarrollo del lenguaje. Hubo que cambiar el nombre debido a que ya existía otro lenguaje con el nombre de oak.

Sun MicroSystems lanzó las primeras versiones de Java a principios de 1995, y se han ido sucediendo las nuevas versiones durante estos últimos años, fomentando su uso y extendiendo las especificaciones y su funcionalidad.

Una de las características más importantes de Java es su capacidad y, a la vez, facilidad para realizar aplicaciones que funcionen en red. La mayoría de los detalles de implementación a bajo nivel están ocultos y son tratados de forma transparente por la JVM (*Java Virtual Machine*). Los programas son independientes de la arquitectura y se ejecutan indistintamente en una gran variedad de equipos con diferentes tipos de microprocesadores y sistemas operativos.

2 CLASES PARA LAS COMUNICACIONES DE RED EN JAVA: java.net

En las aplicaciones en red es muy común el paradigma cliente-servidor. El servidor es el que espera las conexiones del cliente (en un lugar claramente definido) y el cliente es el que lanza las peticiones a la maquina donde se está ejecutando el servidor, y al lugar donde está esperando el servidor (el puerto(s) específico que atiende). Una vez establecida la conexión, ésta es tratada como un *stream* (flujo) típico de entrada/salida.

Cuando se escriben programas Java que se comunican a través de la red, se está programando en la capa de aplicación. Típicamente, no se necesita trabajar con las capas TCP y UDP, en su lugar se puede utilizar las clases del paquete **`java.net`**. Estas clases proporcionan comunicación de red independiente del sistema.

A través de las clases del paquete **`java.net`**, los programas Java pueden utilizar TCP o UDP para comunicarse a través de Internet. Las clases **`URL`**, **`URLConnection`**, **`Socket`**, y **`SocketServer`** utilizan TCP para comunicarse a través de la Red. Las clases **`DatagramPacket`** y **`DatagramServer`** utilizan UDP.

TCP proporciona un canal de comunicación fiable punto a punto, lo que utilizan para comunicarse las aplicaciones cliente-servidor en Internet. Las clases **`Socket`** y **`ServerSocket`** del paquete **`java.net`** proporcionan un canal de comunicación independiente del sistema utilizando TCP, cada una de las cuales implementa el lado del cliente y el servidor respectivamente.

Así el paquete java.net proporciona, entre otras, las siguientes clases, que son las que se verán con detalle:

- **`Socket`**: Implementa un extremo de la conexión TCP.
- **`ServerSocket`**: Se encarga de implementar el extremo Servidor de la conexión en la que se esperarán las conexiones de los clientes.
- **`DatagramSocket`**: Implementa tanto el servidor como el cliente cuando se utiliza UDP.
- **`DatagramPacket`**: Implementa un *datagram packet*, que se utiliza para la creación de servicios de reparto de paquetes sin conexión.

- **`InetAddress`**: Se encarga de implementar la dirección IP.

La clase **`Socket`** del paquete **`java.net`** es una implementación independiente de la plataforma de un cliente para un enlace de comunicación de dos vías entre un cliente y un servidor. Utilizando la clase **`java.net.Socket`** en lugar de tratar con código nativo, los programas Java pueden comunicarse a través de la red de una forma independiente de la plataforma.

El entorno de desarrollo de Java incluye un paquete, **`java.io,`** que contiene un juego de canales de entrada y salida que los programas pueden utilizar para leer y escribir datos. Las clases **`InputStream`** y **`OutputStream`** del paquete **`java.io`** son superclases abstractas que definen el comportamiento de los canales de I/O de tipo stream de Java. **`java.io`** también incluye muchas subclases de **`InputStream`** y **`OtputStream`** que implementan tipos especificos de canales de I/O.

2.1 DATAGRAM SOCKET (Servicio sin Conexión)

Es el más simple, lo único que se hace es enviar los datos, mediante la creación de un *socket* y utilizando los métodos de envío y recepción apropiados.

Se trata de un servicio de transporte sin conexión. Son más eficientes que TCP, pero no está garantizada la fiabilidad: los datos se envían y reciben en paquetes, cuya entrega no está garantizada; los paquetes pueden ser duplicados, perdidos o llegar en un orden diferente al que se envió.

El protocolo de comunicaciones con datagramas UDP, es un protocolo sin conexión, es decir, cada vez que se envíen datagramas es necesario enviar el descriptor del *socket* local y la dirección del *socket* que debe recibir el datagrama. Como se puede ver, hay que enviar datos adicionales cada vez que se realice una comunicación.

`public class java.net.DatagramSocket extends java.lang.Object`

A) Constructores :

`public DatagramSocket () throws SocketException`
Se encarga de construir un *socket* para datagramas y de conectarlo al primer puerto disponible.

`public DatagramSocket (int port) throws SocketException`
Ídem, pero con la salvedad de que permite especificar el número de puerto asociado.

`public DatagramSocket (int port, InetAddress ip) throws SocketException`
Permite especificar, además del puerto, la dirección local a la que se va a asociar el socket.

B) Métodos :

`public void close()`
Cierra el *socket*.

`protected void finalize()`
Asegura el cierre del *socket* si no existen más referencias al mismo.

`public int getLocalPort()`
Retorna el número de puerto en el *host* local al que está conectado el *socket*.

`public void receive (DatagramPacket p) throws IOException`
Recibe un `DatagramPacket` del *socket*, y llena el búfer con los datos que recibe.

`public void send (DatagramPacket p) throws IOException`
Envía un `DatagramPacket` a través del *socket*.

2.2 DATAGRAM PACKET

Un `DatagramSocket` envía y recibe los paquetes y un `DatagramPacket` contiene la información relevante. Cuando se desea recibir un datagrama, éste deberá almacenarse bien en un búfer o

un array de bytes. Y cuando preparamos un datagrama para ser enviado, el `DatagramPacket` no sólo debe tener la información, sino que además debe tener la dirección IP y el puerto de destino, que puede coincidir con un puerto TCP.

`pulic final class java.net.DatagramPacket extends java.lang.Object`

A) Constructores:

`public DatagramPacket(byte ibuf[], int ilength)`
Implementa un `DatagramPacket` para la recepción de paquetes de longitud `ilength`, siendo el valor de este parámetro menor o igual que `ibuf.length`.

`public DatagramPacket(byte ibuf[], int ilength, InetAddress iaddr, int iport)`
Implementa un `DatagramPacket` para el envío de paquetes de longitud `ilength` al número de puerto especificado en el parámetro `iport`, del *host* especificado en la dirección de destino que se le pasa por medio del parámetro `iaddr`.

B) Métodos:

`public InetAddress getAddress ()`
Retorna la dirección IP del *host* al cual se le envía el datagrama o del que el datagrama se recibió.

`public byte[] getData()`
Retorna los datos a recibir o a enviar.

`public int getLength()`
Retorna la longitud de los datos a enviar o a recibir.

`public int getPort()`
Retorna el número de puerto de la máquina remota a la que se le va a enviar el datagrama o del que se recibió.

2.3 STREAM SOCKET (Servicio Orientado a Conexión)

Es un servicio orientado a conexión donde los datos se transfieren sin encuadrarlos en registros o bloques. Si se rompe la conexión entre los procesos, éstos serán informados. El protocolo de comunicaciones con *streams* es un protocolo orientado a conexión, ya que para establecer una comunicación utilizando el protocolo TCP, hay que establecer en primer lugar una conexión entre un par de *sockets*. Mientras uno de los *sockets* atiende peticiones de conexión (servidor), el otro solicita una conexión (cliente). Una vez que los dos *sockets* estén conectados, se pueden utilizar para transmitir datos en ambas direcciones.

Permite a las aplicaciones Cliente y Servidor, disponer de un *stream* que facilita la comunicación entre ambos, obteniéndose una mayor fiabilidad.

El funcionamiento es diferente al anterior ya que cada extremo se comportará de forma diferente, el servidor adopta un papel (inicial) pasivo y espera conexiones de los clientes. Mientras que el cliente adoptará un papel (inicial) activo, solicitando conexiones al servidor.

En la parte del **servidor** se tiene:

`public final class java.net.ServerSocket extends java.lang.Object`

A) Constructores :

`public ServerSocket (int port) throws IOException`
Se crea un *socket* local al que se enlaza el puerto especificado en el parámetro `port`, si se especifica un 0 en dicho parámetro creará el *socket* en cualquier puerto disponible. Puede aceptar hasta 50 peticiones en cola pendientes de conexión por parte de los clientes.

`public ServerSocket (int port , int count) throws IOException`
Aquí, el parámetro `count` sirve para que puede especificarse, el número máximo de peticiones de conexión que se pueden mantener en cola.

Hay que recordar, que es fundamental que el puerto escogido sea conocido por el cliente, en caso contrario, no se podría establecer la conexión.

B) Métodos :

public Socket accept () throws IOException
Sobre un `ServerSocket` se puede realizar una espera de conexión por parte del cliente mediante el método `accept()`. Hay que decir, que este método es de bloqueo, el proceso espera a que se realice una conexión por parte del cliente para seguir su ejecución. Una vez que se ha establecido una conexión por el cliente, este método devolverá un objeto tipo `Socket`, a través del cual se establecerá la comunicación con el cliente.

public void close () throws IOException
Se encarga de cerrar el *socket*.

public InetAddress getInetAddress ()
Retorna la dirección IP remota a la cual está conectado el *socket*. Si no lo está retornará **null** .

public int getLocalPort ()
Retorna el puerto en el que está escuchando el *socket*.

public static void setSocketImplFactory (SocketImplFactory fac) throws IOException
Este método establece la compañía de implementación del *socket* para la aplicación. Debido a que cuando una aplicación crea un nuevo *socket*, se realiza una llamada al método `createSocketImpl()` de la compañía que implementa al *socket*. Es por tanto en el parámetro `fac`, donde se especificará la citada compañía.

public String toString( )
Retorna un *string* representando el *socket*.

En la parte del **cliente** :

public final class java.net.Socket extends java.lang.Object

A) Constructores :

public Socket (InetAddress address, int port) throws IOException
Crea un `StreamSocket` y lo conecta al puerto remoto y dirección IP remota especificados.

public Socket (InetAddress address, int port , boolean stream) throws IOException
Ídem a la anterior incluyendo el parámetro booleano `stream` que si es `true` creará un `StreamSocket` y si es `false` un `DatagramSocket` (En desuso).

public Socket (String host, int port) throws UnKnownHostException, IOException
Crea un `StreamSocket` y lo conecta al número de puerto y al nombre de *host* especificados.

public Socket (String host , int port , boolean stream) throws IOException
Ídem al anterior incluyendo el parámetro booleano stream que si es `true` creará un `StreamSocket` y si es false un `DatagramSocket` (En desuso).

B) MÉTODOS :

public void close() throws IOException
Se encarga de cerrar el socket.

public InetAddress getInetAddress ()
Retorna la dirección IP remota a la que se conecta el *socket*.

public InputStream getInputStream () throws IOException
Retorna un *input stream* para la lectura de bytes desde el *socket*.

public int getLocalPort()
Retorna el puerto local al que está conectado el *socket*.

public OutputStream getOutputStream () throws IOException
Retorna un *output stream* para la escritura de bytes hacia el *socket*.

public int getPort ()
Retorna el puerto remoto al que está conectado el *socket*.

public static void setSocketImplFactory (SocketImplFactory fac) throws IOException
Este método establece la compañía de implementación del *socket* para la aplicación. Debido a que cuando una aplicación crea un nuevo *socket*, se realiza una llamada al método `createSocketImpl( )` de la

compañía que implementa al *socket*. Es por tanto en el parámetro `fac` , donde especificaremos la citada compañía.

2.4 La Clase InetAddress

Esta clase implementa la dirección IP.

`public final class java.net.InetAddress extends java.lang.Object`

A) Constructores :
Para crear una nueva instancia de esta clase se debe de llamar a los métodos `getLocalHost()`, `getByname()` o `getAllByName()`

B) Métodos :

`public boolean equals (Object obj)`
Devuelve un booleano a `true` si el parámetro que se la pasa no es `null` e implementa la misma dirección IP que el objeto. Dos instancias de `InetAddress` implementan la misma dirección IP si la longitud del vector de bytes que nos devuelve el método `getAddress()` es la misma para ambas y cada uno de los componentes del vector de componentes es el mismo que el vector de bytes.

`public byte[] getAddress ()`
Retorna la dirección raw IP del objeto `InetAddress`.
Hay que tener en cuenta que el byte de mayor orden de la dirección estará en `getAddress()[0]` .

`public static InetAddress[] getAllByName(String host) throws UnknownHostException`
Retorna un vector con todas las direcciones IP del *host* especificado en el parámetro.

`public static InetAddress getByName (String host) throws UnknownHostException`
Retorna la dirección IP del nombre del *`host`* que se le pasa como parámetro, aunque también se le puede pasar un string representando su dirección IP

`public String getHostName()`
Retorna el nombre del *host* para esta dirección IP.

`public static InetAddress getLocalHost() throws UnknownHostException`
Retorna la dirección IP para el *host* local.

`public int hashCode ()`
Retorna un código hash para esta dirección IP.

`public String toString ()`
Retorna un **`String`** representando la dirección IP.

Un ejemplo de utilización muy sencillo de esta clase es el siguiente, que se encarga de devolver la dirección IP de la máquina.

```
public class QuienSoy {
   public static void main (String[] args) throws Exception {
         if (args.length !=1) {
               System.err.println("Uso: java QuienSoy NombreMaquina");
               System.exit(1);
         }
         InetAddress direccion= InetAddress.getByName(args[0]);
         System.out.println(direccion);
   }
}
```

3 ENVIO Y RECEPCIÓN A TRAVÉS DE SOCKETS

El servidor creará un *socket*, utilizando **`ServerSocket`**, le asignará un puerto y una dirección, una vez haga el **`accept`** para esperar llamadas, se quedará bloqueado a la espera de las mismas. Una vez llegue una llamada el **`accept`** creará un **`Socket`** para procesarla.

A su vez, cuando un cliente desee establecer una conexión, creará un *socket* y establecerá una conexión al puerto establecido. Sólo es en este momento, cuando se da una conexión real y se mantendrá hasta su liberación mediante **`close()`**.

Para poder leer y escribir datos, los *sockets* disponen de unos *stream* asociados, uno de entrada (**`InputStream`**) y otro de salida (**`OutputStream`**) respectivamente.

Para obtener estos streams a partir del socket utilizaremos :

`ObjetoDeTipoSocket.getInputStream ()`
Devuelve un objeto de tipo **`InputStream`**.

`ObjetoDeTipoSocket.getOutputStream ()`
Devuelve un objeto de tipo **`OutputStream`**.

Para el envío de datos, puede utilizarse **`OutputStream`** directamente en el caso de que se quiera enviar un flujo de bytes sin búfer o también puede crearse un objeto de tipo *stream* basado en el **`OutputStream`** que proporciona el *socket*.

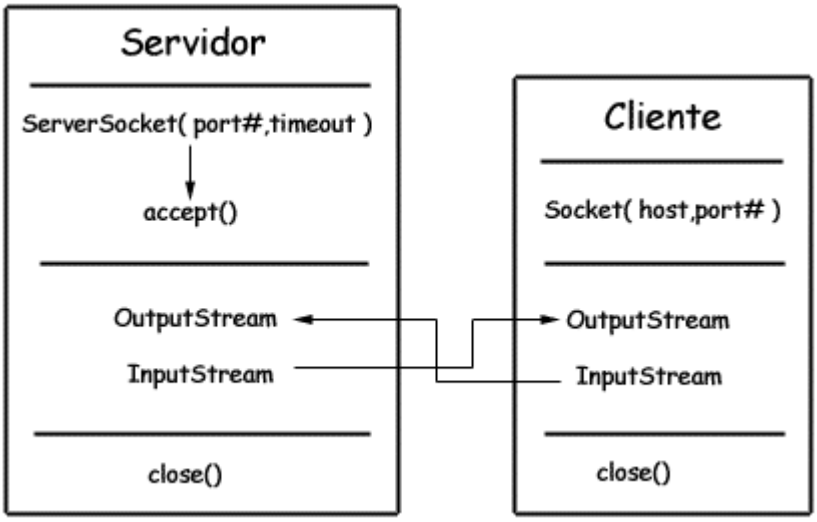

Figura 1. Esquema de conexión mediante *sockets stream*.

En Java, crear una conexión *socket* TCP/IP se realiza directamente con el paquete **`java.net`**. En la Figura 1 se muestra un diagrama de lo que ocurre en el lado del cliente y del servidor.

El servidor establece un puerto y espera a que el cliente establezca la conexión. Cuando el cliente solicite una conexión, el servidor abrirá la conexión *socket* con el método **`accept()`**.

El cliente establece una conexión con la máquina *host* a través del puerto que se designe en *port*#. El cliente y el servidor se comunican con manejadores **`InputStream`** y **`OutputStream`**

Si se está programando un cliente, el *socket* se abre de la forma:

```
Socket miSocket;
miSocket= new Socket(host, puerto);
```

Donde **host** es el nombre de la máquina sobre la que se está intentando abrir la conexión y **puerto** es el puerto (un número) que el servidor está atendiendo.

Cuando se selecciona un número de puerto, se debe tener en cuenta que los puertos en el rango 0-1023 están reservados. Estos puertos son los que utilizan los servicios estándar del sistema como **email**, **ftp**, **http**, etc. Por lo que, para aplicaciones de usuario, el programador deberá asegurarse de seleccionar un puerto por encima del 1023.

Hasta ahora no se han utilizado excepciones; pero deben tener en cuenta la captura de excepciones cuando se está trabajando con *sockets*. Así :

```
Socket miSocket;
try {
     miSocket= new Socket(host, puerto);
} catch(IOException e) {
     System.out.println(e);
} catch (UnknownHostException uhe) {
     System.out.println(uhe);
}
```

En el caso de estar implementando un servidor, la forma de apertura del *socket* sería como sigue :

```
Socket socketSrv;
try {
     socketSrv= new ServerSocket(puerto);
} catch(IOException e) {
     System.out.println(e);
}
```

Cuando se implementa un servidor se necesita crear un objeto **Socket** a partir del **ServerSocket**, para que éste continúe ateniendo las conexiones que soliciten potenciales nuevos clientes y poder servir al cliente, recién conectado, a través del **Socket** creado:

```
Socket socketServcio= null;
try {
     socketServicio= socketSrv.accept();
} catch(IOException e) {
     System.out.println(e);
}
```

4 CREACIÓN DE STREAMS

4.1 Creación de *Streams* de Entrada

En la parte cliente de la aplicación, se puede utilizar la clase **DataInputStream** para crear un *stream* de entrada que esté listo a recibir todas las respuestas que el servidor le envíe.

```
DataInputStream inSocket;
```

```
try {
     inSocket= new DataInputStream(miSocket.getInputStream());
} catch( IOException e ) {
System.out.println( e );
}
```

La clase **DataInputStream** permite la lectura de líneas de texto y tipos de datos primitivos de Java de un modo altamente portable; dispone de métodos para leer todos esos tipos como: **read()**, **readChar()**, **readInt()**, **readDouble()** y **readUTF()**.

Deberá utilizarse la función que se crea necesaria dependiendo del tipo de dato que se espera recibir del servidor. En el lado del servidor, también se usará **DataInputStream**, pero en este caso para recibir las entradas que se produzcan de los clientes que se hayan conectado:

```
DataInputStream inSocket;
try {
     inSocket=new DataInputStream(socketServicio.getInputStream());
} catch(IOException e) {
     System.out.println(e);
}
```

4.2 Creación de *Streams* de Salida

En el lado del cliente, puede crearse un *stream* de salida para enviar información al *socket* del servidor utilizando las clases **PrintStream** o **DataOutputStream**:

```
PrintStream outSocket;
try {
     outSocket= new PrintStream(miSocket.getOutputStream());
} catch(IOException e) {
     System.out.println(e);
}
```

La clase **PrintStream** tiene métodos para la representación textual de todos los datos primitivos de Java. Sus métodos **write** y **println()** tienen una especial importancia en este aspecto. No obstante, para el envío de información al servidor también podemos utilizar **DataOutputStream**:

```
DataOutputStream outSocket;
try {
     outSocket= new DataOutputStream(miSocket.getOutputStream());
} catch(IOException e) {
     System.out.println( e );
}
```

La clase **DataOutputStream** permite escribir cualquiera de los tipos primitivos de Java, muchos de sus métodos escriben un tipo de dato primitivo en el *stream* de salida. De todos

esos métodos, el más útil quizás sea **writeBytes()**. En el lado del servidor, puede utilizarse la clase **PrintStream** para enviar información al cliente:

```
PrintStream outSocket;
try {
     outSocket= new PrintStream(socketServicio.getOutputStream());
} catch(IOException e) {
     System.out.println(e);
}
```

Pero también puede utilizarse la clase **DataOutputStream** como en el caso de envío de información desde el cliente.

5 CIERRE DE SOCKETS

Siempre deben cerrarse los canales de entrada y salida que se hayan abierto durante la ejecución de la aplicación.

En el lado del cliente:

```
try {
     outSocket.close(); inSocket.close(); miSocket.close();
} catch(IOException e) {
     System.out.println(e);
}
```

Y en el lado del servidor:

```
try {
     outSocket.close(); inSocket.close();
     socketServicio.close(); socketSrv.close();
} catch(IOException e) {
     System.out.println(e);
}
```

6 DIFERENCIAS ENTRE SOCKETS STREAM Y DATAGRAMA

Ahora se presenta un problema, ¿qué protocolo, o tipo de *sockets*, debe utilizarse - UDP o TCP? La decisión depende de la aplicación cliente/servidor que se esté escribiendo. Se muestran, a continuación, algunas diferencias entre los protocolos para ayudar en la decisión.

En UDP, cada vez que se envía un datagrama, hay que enviar también el descriptor del *socket* local y la dirección del *socket* que va a recibir el datagrama, luego éstos son más grandes que los TCP. Como el protocolo TCP está orientado a conexión, tiene que establecerse esta conexión entre los dos *sockets* antes de nada, lo que implica un cierto tiempo empleado en el establecimiento de la conexión, que no existe en UDP.

En UDP hay un límite de tamaño de los datagramas, establecido en 64 kilobytes, que se pueden enviar a una localización determinada, mientras que TCP no tiene límite; una vez que

se ha establecido la conexión, el par de *sockets* funciona como los *streams*: todos los datos se leen inmediatamente, en el mismo orden en que se van recibiendo.

UDP es un protocolo desordenado, no garantiza que los datagramas que se hayan enviado sean recibidos en el mismo orden por el *socket* de recepción. Al contrario, TCP es un protocolo ordenado, garantiza que todos los paquetes que se envíen serán recibidos en el *socket* destino en el mismo orden en que se han enviado.

Los datagramas son bloques de información del tipo lanzar y olvidar. Para la mayoría de los programas que utilicen la red, el usar un flujo TCP en vez de un datagrama UDP es más sencillo y hay menos posibilidades de tener problemas. Sin embargo, cuando se requiere un rendimiento óptimo, y está justificado el tiempo adicional que supone realizar la verificación de los datos, los datagramas son un mecanismo realmente útil.

En resumen, TCP parece más indicado para la implementación de servicios de red como un control remoto (**`rlogin`**, **`telnet`**) y transmisión de ficheros (**`ftp`**), que necesitan transmitir datos de longitud indefinida. UDP es menos complejo y tiene una menor sobrecarga sobre la conexión, esto hace que sea el indicado en la implementación de aplicaciones cliente/servidor en sistemas distribuidos montados sobre redes de área local.

7 CLASES UTILES EN COMUNICACIONES

A continuación se introducen otras clases que resultan útiles cuando se está desarrollando programas de comunicaciones, aparte de las que ya se han visto. El problema es que la mayoría de estas clases se prestan a discusión, porque se encuentran bajo el directorio sun. Esto quiere decir que son implementaciones Solaris y, por tanto, específicas del Unix Solaris. Además su API no está garantizada, pudiendo cambiar. Pero, a pesar de todo, resultan muy interesantes y se comentarán un grupo de ellas, las que se encuentran en el paquete **`sun.net`**.

- **`MulticastSocket`**: Clase utilizada para crear una versión multicast de las clase socket datagrama. Múltiples clientes/servidores pueden transmitir a un grupo multicast (un grupo de direcciones IP compartiendo el mismo número de puerto).
- **`NetworkServer`**: Una clase creada para implementar métodos y variables utilizadas en la creación de un servidor TCP/IP.
- **`NetworkClient`**: Una clase creada para implementar métodos y variables utilizadas en la creación de un cliente TCP/IP.
- **`SocketImpl`**: Es un interface que permite crear nuestro propio modelo de comunicación. Tendrán que implementarse sus métodos cuando se use. Si se va a desarrollar una aplicación con requerimientos especiales de comunicaciones, como pueden ser la implementación de un cortafuegos (TCP es un protocolo no seguro), o acceder a equipos especiales (como un lector de código de barras o un GPS diferencial), se necesita una de **`Socket`** específica.

8 EJEMPLO CLIENTE DE ECO

Sea el programa cliente, **`PruebaEco`**, conecta con el Echo del servidor (en el puerto 7) mediante un *socket*. El cliente lee y escribe a través del *socket*. **`PruebaEco`** envía todo el texto tecleado en su entrada estándar al Echo del servidor, escribiéndole el texto al *socket*. El servidor repite todos los caracteres recibidos en su entrada desde el cliente de vuelta a través del *socket* al cliente. El programa cliente lee y muestra los datos pasados de vuelta desde el servidor.

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

public class PruebaEco {

    public static void main(String[] args) {
        Socket ecoSocket = null;
        DataOutputStream salida = null;
        DataInputStream entrada = null;
        DataInputStream stdIn = new DataInputStream(System.in);

       /* Establecimiento de la conexión del Socket entre el cliente y el servidor y
                                        apertura del canal E/S sobre el socket : */

        try {
            ecoSocket = new Socket(args[0], 7);
            salida = new DataOutputStream(ecoSocket.getOutputStream());
            entrada = new DataInputStream(ecoSocket.getInputStream());
        } catch (UnknownHostException e) {
            System.err.println("No conozco al host: " + args[0]);
        } catch (IOException e) {
            System.err.println("Error de E/S para la conexión con: " + args[0]);
        }

    /* Ahora lee desde el stream de entrada estándar una línea cada vez. El programa
escribe inmediatamente la entrada seguida por un carácter de nueva línea en el stream
                                                de salida conectado al socket. */

        if (ecoSocket != null && salida != null && entrada != null) {
            try {
                String userInput;
                while ((userInput = stdIn.readLine()) != null) {
                    salida.writeBytes(userInput);
                    salida.writeByte('\n');
                    System.out.println("eco: " + entrada.readLine());
                }

  /* Cuando el usuario teclea un carácter de fin de entrada, el bucle while termina.
      Cierre de los streams de entrada y salida conectados al socket, y cierre de la
                                        conexión del socket con el servidor. */
```

```
                salida.close();
                entrada.close();
                ecoSocket.close();
            } catch (IOException e) {
                System.err.println("E/S fallo en la conexión a: " + args[0]);
            }
        }
    }
}
```