Los Sockets

Los *sockets* (zócalos, referido a los enchufes de conexión de cables) son mecanismos de comunicación entre programas a través de una red TCP/IP. De hecho, al establecer una conexión via Internet estamos utilizando sockets: los sockets realizan la interfase entre la aplicación y el protocolo TCP/IP.

Dichos mecanismos pueden tener lugar dentro de la misma máquina o a través de una red. Se usan en forma cliente-servidor: cuando un cliente y un servidor establecen una conexión, lo hacen a través de un socket. Java proporciona para esto las clases ServerSocket y Socket.

Los sockets tienen asociado un *port* (puerto). En general, las conexiones via internet pueden establecer un puerto particular (por ejemplo, en http://www.rockar.com.ar:80/index.html el puerto es el 80). Esto casi nunca se especifica porque ya hay definidos puertos por defecto para distintos protocolos: 20 para ftp-data, 21 para ftp, 79 para finger, etc. Algunos servers pueden definir otros puertos, e inclusive pueden utilizarse puertos disponibles para establecer conexiones especiales.

Justamente, una de las formas de crear un objeto de la clase URL permite especificar también el puerto:

URL url3 = new URL ("http", "www.rockar.com.ar", 80,"sbits.htm");

Para establecer una conexión a través de un socket, tenemos que programar por un lado el servidor y por otro los clientes.

En el servidor, creamos un objeto de la clase ServerSocket y luego esperamos algún cliente (de clase
Socket) mediante el método accept():

ServerSocket conexion = new ServerSocket(5000); // 5000 es el puerto en este caso
Socket cliente = conexion.accept(); // espero al cliente

Desde el punto de vista del cliente, necesitamos un Socket al que le indiquemos la dirección del servidor y el número de puerto a usar:

Socket conexion = new Socket (direccion, 5000);

Una vez establecida la conexión, podemos intercambiar datos usando streams como en el ejemplo anterior.

Como la clase URLConnection, la clase Socket dispone de métodos getInputStream y getOutputStream que nos dan respectivamente un InputStream y un OutputStream a través de los cuales transferir los datos.

Los sockets son puntos finales de enlaces de comunicaciones entre procesos. Los procesos los tratan como descriptores de ficheros, de forma que se pueden intercambiar datos con otros procesos transmitiendo y recibiendo a través de sockets. El tipo de sockets describe la forma en la que se transfiere información a través de ese socket.

Sockets Stream (TCP)

Son 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 de tal suceso para que tomen las medidas oportunas.

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.

Sockets Datagrama (UDP)

Son un servicio de transporte sin conexión. Son más eficientes que TCP, pero en su utilización 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 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, aunque tiene la ventaja de que se pueden indicar direcciones globales y el mismo mensaje llegará a un muchas máquinas a la vez.

Sockets Raw

Son sockets que dan acceso directo a la capa de software de red subyacente o a protocolos de más bajo nivel. Se utilizan sobre todo para la depuración del código de los protocolos.

Diferencias entre Sockets Stream y Datagrama

Ahora se presenta un problema, al haber varias opciones, ¿qué protocolo, o tipo de sockets, se debe emplear - UDP o TCP? La decisión depende de la aplicación cliente/servidor que se esté escribiendo; aunque hay algunas diferencias entre los protocolos que sirven para ayudar en la decisión y decantar la utilización de sockets de un tipo.

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 los mensajes son más grandes que los TCP. Como el protocolo TCP está orientado a conexión, hay que establecer esta conexión entre los dos sockets antes de nada, lo que implica un cierto tiempo empleado en el establecimiento de la conexión, que no es necesario emplear 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, la comunicación a través de sockets TCP es 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.

Un servidor atento

Vamos a crear un servidor Ejemplo26a.java (que podemos correr en una ventana) que atenderá a un cliente de la misma máquina (lo vamos a correr en otra ventana). Para hacerlo simple, el servidor sólo le enviará un mensaje al cliente y éste terminará la conexión. El servidor quedará entonces disponible para otro cliente.

Es importante notar que, para que el socket funcione, los servicios TCP/IP deben estar activos (aunque ambos programas corran en la misma máquina). Los usuarios de Windows asegúrense que haya una conexión TCP/IP activa, ya sea a una red local o a Internet.

El servidor correrá "para siempre", así que para detenerlo presionen control-C.

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

public class Ejemplo26a {
	public static void main(String argv[]) {
		ServerSocket servidor;
		Socket cliente;
		int numCliente = 0;
		try {
			servidor = new ServerSocket(5000);
			do {
				numCliente++;
				cliente = servidor.accept();
				System.out.println("Llega el cliente "+numCliente);
				PrintStream ps = new PrintStream(cliente.getOutputStream());
				ps.println("Usted es mi cliente "+numCliente);
				cliente.close();
			} while (true);
		}
		catch (Exception e) {
			e.printStackTrace();
		}
	}
}
```

Utilizamos un PrintStream para enviar los datos al cliente, ya que es sencillo de utilizar para mandar Strings.

El método PrintStream.println maneja los datos como System.out.println, simplemente hay que indicarle el stream a través del cual mandarlos al crearlo (en este caso, el OutputStream del cliente, que obtenemos con cliente.getOutputStream()).

El cliente satisfecho

Ahora vamos a crear la clase cliente, Ejemplo26b.java. El cliente simplemente establece la conexión, lee a través de un DataInputStream (mediante el método readLine()) lo que el servidor le manda, lo muestra y corta.

```
// cliente:
import java.io.*;
import java.net.*;

public class Ejemplo26b {
	public static void main(String argv[]) {
		InetAddress direccion;
		Socket servidor;
		int numCliente = 0;
		try {
			direccion = InetAddress.getLocalHost(); // direccion local
			servidor = new Socket(direccion, 5000);
			DataInputStream datos =
			new DataInputStream(servidor.getInputStream());
			System.out.println( datos.readLine() );
			servidor.close();
		}
		catch (Exception e) {
			e.printStackTrace();
		}
	}
}
```

Para probar esto, asegúrense que los servicios TCP/IP estén activos, corran java Ejemplo26a en una ventana y corran varias veces java Ejemplo26b en otra. Las salidas serán más o menos así:

Ventana servidor:

```
C:\java\curso>java Ejemplo26a
Llega el cliente 1
Llega el cliente 2
Llega el cliente 3
(----- cortar con control-C -----)
```

Ventana cliente:

```
C:\java\curso>java Ejemplo26b
Usted es mi cliente 1
C:\java\curso>java Ejemplo26b
Usted es mi cliente 2
C:\java\curso>java Ejemplo26b
Usted es mi cliente 3
(----- aquí cerramos el servidor -----)

C:\java\curso>java Ejemplo26b
java.net.SocketException: connect
at java.net.PlainSocketImpl.doConnect(PlainSocketImpl.java:223)
at java.net.PlainSocketImpl.connectToAddress(PlainSocketImpl.java:128)
at java.net.PlainSocketImpl.connect(PlainSocketImpl.java:115)
at java.net.Socket.<init>(Socket.java:125)
at java.net.Socket.<init>(Socket.java:101)
at Ejemplo26b.main(Ejemplo26b.java:12)
```

Se mencionan los constructores que tiene esta clase y también sus métodos, pero no se describen detalladamente, se puede consultar toda la información en el siguiente link:

http://java.sun.com/j2se/1.4.2/docs/api/java/net/Socket.html

Constructor Summary	
	Socket() Creates an unconnected socket, with the system-default type of SocketImpl.
	Socket(InetAddress address, int port) Creates a stream socket and connects it to the specified port number at the specified IP address.
	Socket(InetAddress host, int port, boolean stream) Deprecated. Use DatagramSocket instead for UDP transport.
	Socket(InetAddress address, int port, InetAddress localAddr, int localPort) Creates a socket and connects it to the specified remote address on the specified remote port.
protected	**Socket(SocketImpl impl)** Creates an unconnected Socket with a user-specified SocketImpl.
	Socket(String host, int port) Creates a stream socket and connects it to the specified port number on the named host.
	Socket(String host, int port, boolean stream) Deprecated. Use DatagramSocket instead for UDP transport.
	Socket(String host, int port, InetAddress localAddr, int localPort) Creates a socket and connects it to the specified remote host on the specified remote port.

Method Summary	
void	**bind(SocketAddress bindpoint)** Binds the socket to a local address.
void	**close()** Closes this socket.
void	**connect(SocketAddress endpoint)** Connects this socket to the server.
void	**connect(SocketAddress endpoint, int timeout)** Connects this socket to the server with a specified timeout value.
SocketChannel	**getChannel()** Returns the unique SocketChannel object associated with this socket, if any.
InetAddress	**getInetAddress()** Returns the address to which the socket is connected.
InputStream	**getInputStream()** Returns an input stream for this socket.
boolean	**getKeepAlive()** Tests if SO_KEEPALIVE is enabled.
InetAddress	**getLocalAddress()** Gets the local address to which the socket is bound.
int	**getLocalPort()**

	Returns the local port to which this socket is bound.
SocketAddress	**getLocalSocketAddress()** Returns the address of the endpoint this socket is bound to, or null if it is not bound yet.
boolean	**getOOBInline()** Tests if OOBINLINE is enabled.
OutputStream	**getOutputStream()** Returns an output stream for this socket.
int	**getPort()** Returns the remote port to which this socket is connected.
int	**getReceiveBufferSize()** Gets the value of the SO_RCVBUF option for this Socket, that is the buffer size used by the platform for input on this Socket.
SocketAddress	**getRemoteSocketAddress()** Returns the address of the endpoint this socket is connected to, or null if it is unconnected.
boolean	**getReuseAddress()** Tests if SO_REUSEADDR is enabled.
int	**getSendBufferSize()** Get value of the SO_SNDBUF option for this Socket, that is the buffer size used by the platform for output on this Socket.
int	**getSoLinger()** Returns setting for SO_LINGER.
int	**getSoTimeout()** Returns setting for SO_TIMEOUT.
boolean	**getTcpNoDelay()** Tests if TCP_NODELAY is enabled.
int	**getTrafficClass()** Gets traffic class or type-of-service in the IP header for packets sent from this Socket
boolean	**isBound()** Returns the binding state of the socket.
boolean	**isClosed()** Returns the closed state of the socket.
boolean	**isConnected()** Returns the connection state of the socket.
boolean	**isInputShutdown()** Returns wether the read-half of the socket connection is closed.
boolean	**isOutputShutdown()** Returns wether the write-half of the socket connection is closed.
void	**sendUrgentData(int data)** Send one byte of urgent data on the socket.
void	**setKeepAlive(boolean on)** Enable/disable SO_KEEPALIVE.
void	**setOOBInline(boolean on)** Enable/disable OOBINLINE (receipt of TCP urgent data) By default, this option is disabled and TCP urgent data received on a socket is silently discarded.
void	**setReceiveBufferSize(int size)** Sets the SO_RCVBUF option to the specified value for this Socket.
void	**setReuseAddress(boolean on)** Enable/disable the SO_REUSEADDR socket option.
void	**setSendBufferSize(int size)** Sets the SO_SNDBUF option to the specified value for this Socket.
static void	**setSocketImplFactory(SocketImplFactory fac)**

	Sets the client socket implementation factory for the application.
void	**setSoLinger(boolean on, int linger)** Enable/disable SO_LINGER with the specified linger time in seconds.
void	**setSoTimeout(int timeout)** Enable/disable SO_TIMEOUT with the specified timeout, in milliseconds.
void	**setTcpNoDelay(boolean on)** Enable/disable TCP_NODELAY (disable/enable Nagle's algorithm).
void	**setTrafficClass(int tc)** Sets traffic class or type-of-service octet in the IP header for packets sent from this Socket.
void	**shutdownInput()** Places the input stream for this socket at "end of stream".
void	**shutdownOutput()** Disables the output stream for this socket.
String	**toString()** Converts this socket to a String.

Referencias

http://java.sun.com/j2se/1.4.2/docs/api/java/net/Socket.html
Java Desde Cero, Bourdette, Jorge; Hardcover; Spanish; Castilian;
www.itapizaco.edu.mx/paginas/ JavaTut/froufe/parte20/cap20-4.html