SOCKET

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 vía Internet estamos utilizando sockets: los sockets realizan la interface 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 vía internet pueden establecer un puerto particular (por ejemplo, en http://www.facebook.com: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, 110 para pop, 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.facebook.com", 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
Socket cliente = conexion.accept(); // espera 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.

SOCKET 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.

SOCKET 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 muchas máquinas a la vez.

DIFERENCIAS ENTRE SOCKET STREAM Y DATAGRAMAS

La interrogante es: ¿qué protocolo, o tipo de sockets, debemos usar - UDP o TCP? La decisión depende de la aplicación cliente/servidor que estemos escribiendo. Vamos a ver 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, tenemos 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 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 (“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.

IMPLEMENTACION DE UN SERVIDOR TCP

Vamos a crear un servidor que atenderá a un cliente de la misma máquina. 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

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

public class EjemServidor {

  public static void main(String[] args) {
    ServerSocket servidor;
    Socket cliente;
    int cont=0;

    try{
      servidor = new ServerSocket(6000);  // 6000 es el puerto
      System.out.println("Esperando conexión del cliente");

      // Espera la conexión del cliente
      do {
        cont++;
        cliente = servidor.accept();    // Hace una pausa esperando al cliente
        System.out.println("Cliente conectado N° "+ cont);

        // Envia flujo de datos al cliente
        PrintStream envio = new PrintStream(cliente.getOutputStream());
        envio.println("Tu eres el cliente nro "+cont);

        // Recibe flujo de datos del cliente
        DataInputStream recibir = new DataInputStream(cliente.getInputStream());
        System.out.println(recibir.readUTF());
        cliente.close();
      }while(cont<3);

      servidor.close();

    }
    catch(IOException e){
      System.out.println(e);
    }
  }
}
```

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()).

IMPLEMENTACION DE UN CLIENTE TCP

Ahora vamos a crear la clase cliente. 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.

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

public class EjempCliente {

  public static void main(String[] args) {
    InetAddress direc;
    Socket servidor;

    try{
      direc = InetAddress.getLocalHost(); // Recupera nombre e IP del equipo.
      System.out.println(direc);
      servidor = new Socket(direc, 6000);

      // Recibe un flujo de datos del servidor
      System.out.println("Esperando respuesta del servidor");
      DataInputStream recibir = new DataInputStream(servidor.getInputStream());
      String dato = recibir.readLine();
      System.out.println(dato);

      System.out.println("Esperando segunda respuesta del servidor");
      DataInputStream recibir1 = new DataInputStream(servidor.getInputStream());
      System.out.println(recibir1.readLine());

      // Envía un flujo de datos hacia el servidor.
      DataOutputStream enviar = new DataOutputStream(servidor.getOutputStream());
      enviar.writeUTF("Hola, soy el cliente");

      System.out.println("Termino de conexion con el servidor");

      recibir.close();
      servidor.close();
    }
    catch(IOException e){
      System.out.println(e);
    }
  }
}
```

Para probar esto, asegúrense que los servicios TCP/IP estén activos, corran primero el ejemplo del servidor y luego corran varias veces el ejemplo del cliente

IMPLEMENTACION DE UN SERVIDOR UDP (Datagrama)

Se crea un servidor que atenderá a un cliente de la misma máquina. Este esperará un mensaje del cliente y terminará la conexión.

```
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;

public class EjemServData {

  public static void main(String[] args) {
    DatagramSocket ioConexion;
    byte datos[] = new byte[30];

    System.out.println("Esperando un datagrama...");

    try{
      ioConexion = new DatagramSocket(6001);  // Especifica el puerto del servidor

      // Crea paquete para recibir datos de tipo "byte", especificando su largo.
      DatagramPacket r = new DatagramPacket(datos, datos.length);

      ioConexion.receive(r);  // Espera a recibir paquete de datos.

      String cadena = new String(datos);  // Convierte de byte a String.

      System.out.println(cadena);
      ioConexion.close();
    }
    catch(IOException e){
      System.out.println(e);
    }
  }
}
```

IMPLEMENTACION DE UN CLIENTE UDP (Datagrama)

Ahora vamos a crear al cliente. El cliente simplemente envía un mensaje al servidor. En el caso del Datagrama no es necesario establecer conexión, por lo que el servidor podría no estar activo y aún así el cliente no lo notaría.

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

public class EjemClienteData {

  public static void main(String[] args) {
    DatagramSocket ioConexion;
    byte datos[] = new byte[30];
    Scanner sc = new Scanner(System.in);

    System.out.print("Ingrese texto a enviar: ");
    String cadena = sc.nextLine();

    datos = cadena.getBytes();  // Convierte de String a byte

    try{
      ioConexion = new DatagramSocket();

      // Crea paquete para el envío de datos de tipo "byte", el largo de los
      // datos y la dirección IP y el puerto del servidor
      DatagramPacket enviar = new DatagramPacket(datos, datos.length,
                              InetAddress.getLocalHost(), 6001);

      ioConexion.send(enviar);  // envía paquete
      ioConexion.close();
    }
    catch(IOException e){
      System.out.println(e);
    }
  }
}
```