Sockets

La biblioteca estándar de clases de Java nos proporciona todo lo que necesitamos para utilizar sockets en nuestras aplicaciones en el paquete `java.net`, por lo que tendremos que añadir la siguiente línea al comienzo de nuestros ficheros de código:

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

Los **sockets** son un mecanismo de comunicación entre procesos que se utiliza en Internet.

> Un socket (literalmente, “enchufe”) es un extremo de una comunicación en Internet, por lo que se identifica con una dirección IP (un número entero de 32 bits) y un puerto (un número entero de 16 bits).
>
> > NOTA: Java encapsula el concepto de dirección IP con la clase `java.net.InetAddress`

Existen dos tipos de sockets, en función de si queremos utilizar TCP (orientado a conexión) o UDP (no orientado a conexión)

- `Socket` y `ServerSocket` se utilizan para establecer conexiones y enviar datos utilizando el protocolo TCP.
- `DatagramSocket` se utiliza para transmitir datos usando UDP.

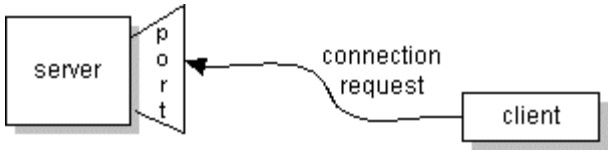
Sockets TCP

Las clases `Socket` y `ServerSocket` permiten utiliza el protocolo TCP en Java:

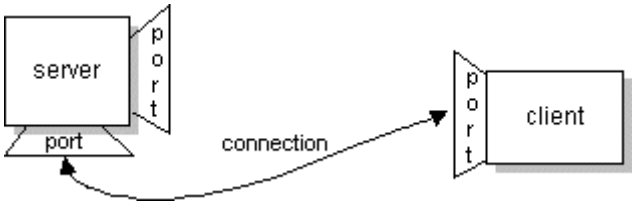
- Un `Socket` se utiliza para transmitir y recibir datos.
- Un `ServerSocket` nunca se utiliza para transmitir datos. Su único cometido es, en el servidor, esperar a que un cliente quiera establecer una conexión con el servidor.

Funcionamiento

El cliente crea un `Socket` para solicitar una conexión con el servidor al que desea conectarse.



Cuando el `ServerSocket` recibe la solicitud, crea un `Socket` en un puerto que no se esté usando y la conexión entre cliente y servidor queda establecida.



Entonces, el `SocketServer` vuelve a quedarse escuchando para recibir nuevas peticiones de clientes.

Transmisión de datos

Cuando la conexión queda establecida, los dos sockets conectados pueden comunicarse entre sí mediante operaciones de lectura y escritura idénticas a las cualquier otro "stream" en Java (`read` y `write`).

La entrada y la salida de un socket se representan en Java mediante las clases `InputStream` y `OutputStream`, respectivamente (las mismas que se utilizan para trabajar con ficheros).

Creación de un cliente TCP: Conexión a un servidor web

Un caso particular de socket TCP es el socket que se utiliza para conectarse a un servidor web. Los servidores web suelen escuchar peticiones en el puerto 80 TCP y emplean el protocolo HTTP.

Tras establecer la conexión TCP con el servidor web, se realiza una petición HTTP como la siguiente:

```
GET ejemplo.html^
^
```

Si el fichero `ejemplo.html` existe, el servidor nos lo devuelve:

```
<HTML>
  <HEAD>
     <TITLE>Página web de ejemplo</TITLE>
  </HEAD>
  <BODY>
    …
  </BODY>
</HTML>
```

El siguiente fragmento de código muestra cómo nos podemos conectar a un servidor web para obtener un fichero de forma remota (/ en este caso, la página principal del servidor web):

```
Socket            socket;
PrintWriter       out;
InputStreamReader reader;
BufferedReader    in;
String            line;

// Conexión TCP

try {
    socket = new Socket("elvex.ugr.es", 80);
} catch (UnknownHostException e) {
    System.err.println("Host desconocido";
}

// Streams de E/S

out = new PrintWriter(socket.getOutputStream());
reader = new InputStreamReader(socket.getInputStream());
in = new BufferedReader(reader);

// Solicitud HTTP

out.println("GET /");
out.println("");
out.flush();

// Respuesta HTTP

line = in.readLine();

while (line!=null) {
      System.out.println(line);
      line = in.readLine();
}

// Cierre de la conexión

out.close();
in.close();
socket.close();
```

Creación de un servidor TCP: Servido de eco

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

class EchoServer
{
  public static void main( String args[] )
         throws IOException
  {
    ServerSocket   serverSocket;
    Socket         clientSocket;
    BufferedReader in;
    PrintWriter    out;
    String         mensaje;
    Bolean         terminar = false;

    serverSocket = new ServerSocket(4444);

    while (!terminar) {

        clientSocket = serverSocket.accept();

        out = new PrintWriter(
                    clientSocket.getOutputStream());
        in = new BufferedReader(
                   new InputStreamReader(
                     clientSocket.getInputStream()));

        mensaje = in.readLine();

        out.println(mensaje);
        out.flush();

        terminar = mensaje.equals("FIN");

        in.close();
        out.close();
        clientSocket.close();
    }

    serverSocket.close();
  }
}
```

- El servidor anterior se instala en el puerto 4444 TCP y se queda esperando peticiones en la llamada al método `accept()`.
- Cuando se acepta la petición, a través del `InputStream` asociado al socket se lee una línea de texto enviada por el cliente.
- Esa misma línea de texto se le envía al cliente a través del `OutputStream` asociado al socket.
- Después, se cierra la conexión con el cliente y el servidor se vuelve a quedar esperando la llegada de nuevas solicitudes de eco.
- El servidor seguirá ejecutándose hasta que reciba una petición de un cliente que solicite el eco de "FIN".

Si queremos probar el servidor, podemos usar la utilidad `telnet` para conectarnos al puerto 4444 de la máquina local con:

```
telnet localhost 4444
```

Cualquier cliente Java puede acceder a nuestro servidor de eco utilizando el siguiente fragmento de código:

```
BufferedReader in;
PrintWriter    out;

Socket socket = new Socket("localhost",4444);

out = new PrintWriter(socket.getOutputStream());

in  = new BufferedReader(
           new InputStreamReader(
                 socket.getInputStream()));

out.println(args[0]);
out.flush();

String eco = in.readLine();
System.out.println(eco);

in.close();
out.close();
socket.close();
```

Creación de un servidor TCP: Servidor de hora

```
Wed May 18 18:18:18 CEST 2005
```

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

class DateServer
{
  public static void main( String args[] )
         throws IOException {

    ServerSocket serverSocket;
    Socket       clientSocket;
    PrintWriter  out;

    serverSocket = new ServerSocket(666);

    try {
      while (true) {  // ¡OJO!

          clientSocket = serverSocket.accept();

          out = new PrintWriter
                    (clientSocket.getOutputStream());

          out.println(new java.util.Date());
          out.flush();
          out.close();
          clientSocket.close();
      }

    } catch (IOException error) {
        serverSocket.close();
    }
  }
}
```

NOTA: El servidor es lo suficientemente simple como para atender todas las peticiones secuencialmente. Si el procesamiento de cada solicitud proveniente de un cliente fuese más costoso, deberíamos utilizar **hebras para atender en paralelo a distintos clientes** (y usaríamos protocolos algo más complejos para hacer algo más útil).

IMPORTANTE

Los servidores, usualmente, han de atender múltiples peticiones provenientes de distintos clientes, por lo que siempre debemos implementarlos como aplicaciones multihebra.

El pseudocódigo de cualquier servidor es siempre similar a:

```
while (!fin) {
      // 1. Aceptar una solicitud
      …
      // 2. Crear una hebra independiente
      //    para procesar la solicitud
      …
}
```

Sockets UDP

El tipo más sencillo de sockets, puesto que no es necesario establecer ninguna conexión para enviar y recibir datos.

En Java, un objeto de tipo `DatagramSocket` representa un socket UDP y puede enviar o recibir datos directamente de otro socket UDP.

Los datos se envían y reciben en paquetes autocontenidos denominados datagramas (igual que el correo convencional).

Los datagramas se representan en Java mediante la clase `DatagramPacket`, que consiste simplemente en un array de bytes dirigido a una dirección IP y a un puerto UDP concretos.

La clase `MulticastSocket` se puede emplear para enviar un mismo datagrama a muchos destinatarios simultáneamente.

Ejemplos disponibles en:
http://java.sun.com/docs/books/tutorial/networking/datagrams/

La clase URL

Cuando alguien accede a un servidor web, generalmente lo hace para obtener un documento, para lo que usa una URL (que en java se representa mediante la clase `java.net.URL`).

El siguiente fragmento de código muestra cómo podemos crear una URL y acceder a un recurso en Internet a partir de la URL:

```
URL url = new URL("http://elvex.ugr.es/");

// Datos de la URL

System.out.println("Protocolo = " + url.getProtocol());
System.out.println("Host = " + url.getHost());
System.out.println("Fichero = " + url.getFile());
System.out.println("Puerto = " + url.getPort());

// Acceso al contenido asociado a la URL

InputStreamReader reader = new InputStreamReader(
                                  url.openStream());

BufferedReader in = new BufferedReader(reader);

String inputLine;

while ((inputLine = in.readLine()) != null)
      System.out.println(inputLine);

in.close();
```

De esta forma no tenemos que conocer los detalles del protocolo HTTP que utilizan los servidores web (ni la dirección IP del servidor, ya que la clase `URL` se encarga de traducir el nombre del host a una dirección IP usando el servicio DNS).

NOTA:

Con `URL.openConnection()` podemos establecer una conexión TCP a través de la cual también se pueden transmitir datos.