Sockets UDP en Java

Un **Socket UDP** es un punto de comunicación por el que un proceso puede emitir o recibir información. Su funcionamiento es similar al de un Socket TCP, excepto por un par de detalles: no se mantiene el orden de entrega de los paquetes ni se garantiza la recepción de todos ellos.

Al igual que antes, normalmente existe un **Servidor UDP** sobre una máquina concreta y tiene un **DatagramSocket** que se pone en espera y escucha el tráfico que le llega de uno o más **Clientes**. El uso de Hilos para gestionar múltiples peticiones entrantes es también la norma.

Dado que los Sockets UDP NO están orientados a la conexión, el servidor NO se detiene si en algún momento el cliente cierre su lado de la conexión… porque NO existe.

En este caso, el código es mucho más simple que el de un Socket TCP. Se crean DatagramPacket para gestionar la escritura y lectura a partir del DatagramSocket. Cuando se termina de utilizar, basta con cerrar el DatagramSocket, que no genera excepción.

```
String mensaje = "100";
byte[] bufferEscritura = new String(mensaje).getBytes();
datagramPacketOut = new DatagramPacket(bufferEscritura, bufferEscritura.length,
        inetAddress, puertoServer);
datagramSocket.send(datagramPacketOut);

System.out.println("Cliente - Mensaje enviado: " + mensaje);
byte[] bufferLectura = new byte[64];
datagramPacketIn = new DatagramPacket(bufferLectura, bufferLectura.length,
        inetAddress, puertoServer);
datagramSocket.receive(datagramPacketIn);

System.out.println("Cliente - Mensaje de respuesta: " + new String(bufferLectura));
```

‘Arreglando’ UDP

Sí, es posible hacer apaños para solventar los problemas de UDP referentes al orden de los datagramas y controlar su posible pérdida. La solución consiste en escribir tú tu propio programa que gestione todo esto. Por ejemplo:

- Añades una ID secuencial a cada Datagrama. El Cliente envía mensajes con ID par, mientras que el Servidor envía sólo mensajes de ID impar.
- Para saber cuál es la respuesta del Servidor para el Datagrama 7, basta con esperar a un Datagrama 8. Si no lo tenemos o no llega, es que algo ha fallado.
- Una marca temporal permite también saber si un Servidor o un Cliente ha dejado de responder, o reordenar la cola de mensajes recibidos.

Pero… ¿para qué molestarse? Estás inventando un **protocolo** sobre UDP que te ofrece las funcionalidades que ya te ofrece TCP…

Este problema me lo encontré hace mucho en un proyecto que usaba el protocolo **SOAP**.

Sockets Multihilo

Si deseamos dar servicio a varios **Clientes** mediante sockets, es necesario usar Hilos. Lo habitual es generar hilos para atender cada petición e forma independiente, sin bloquear al servidor.

El mecanismo es simple: cuando el **Servidor** recibe una petición, crea un socket y se lo asigna a ese hilo, dejando al servidor a la espera para atender la siguiente petición. En general:

- El Server se queda a la espera de recibir peticiones mediante el método **accept ()** de la clase **ServerSocket**. Cada petición se traduce en un nuevo Socket TCP entre cliente y servidor.

- El Server crea un nuevo Hilo al que le asigna el Socket TCP.

- El Hilo es quien crea los streams y gestiona los mensajes, liberando del trabajo al Servidor para que pueda seguir atendiendo peticiones.

- El Cliente puede que necesite de usar Hilos, sobre todo si existe una interfaz que debe seguir siendo operativa. El método join () puede ser necesario para forzar al programa a esperar a que el socket complete su trabajo (por ejemplo, esto sucede en un **Activity** de **Android**).