PRÁCTICA 4

4.1. Objetivo

El objetivo de esta práctica es la introducción al concepto de socket como mecanismo para el envío y recepción de paquetes a través de la red de datos utilizando la pila TCP/IP.

4.2. Conceptos básicos

La pila de protocolos TCP/IP permite la transmisión de datos entre redes de ordenadores. Dicha pila consta de las siguientes capas:

Nivel de aplicación (http, ftp, ssh, telnet)
Nivel de transporte (tcp, udp)
Nivel de red (ip, icmp, arp)
Nivel de enlace (ethernet)
Nivel físico

Normalmente, cuando se escriben aplicaciones Java en red trabajaremos con el nivel de aplicación, y utilizaremos además protocolos de nivel de transporte. Por este motivo es preciso recordar las principales diferencias entre los dos protocolos básicos de nivel de transporte: TCP (Transmission Control Protocol) y UDP (User Datagram Protocol).

TCP

TCP es un protocolo orientado a conexión. Existe un proceso de conexión al inicio, unos datos de control durante la transmisión y un proceso de desconexión. Ofrece el equivalente a un canal libre de errores.

- Es un protocolo orientado a conexión
- Provee un flujo de bytes fiable entre dos ordenadores (llegada en orden, sin errores, sin pérdidas y sin duplicados, control de flujo, etc.).
- Protocolos de nivel de aplicación que usan TCP: telnet, HTTP, FTP, SMTP, etc.

UDP

En UDP no se ofrece nigún tipo de control o fiabilidad en la transmisión de datos: los paquetes enviados pueden llegar, no llegar o hacerlo de forma desordenada. Tiene la ventaja de añadir muy poca sobrecarga a los datos que enviamos a diferencia de TCP. Una aplicación muy típica de UDP es el *streaming* de vídeo o audio en la que no importa si algún paquete que otro se pierde puesto que la información multimedia suele tener bastante redundancia y lo más importante es que los paquetes que llegan lo hagan a tiempo para ser reproducidos. En este caso se prefiere la eficiencia de UDP.

- Es un protocolo no orientado a conexión
- Envía paquetes de datos (datagramas) independientes, sin garantía de llegada.
- Permite *broadcast* y *multicast*.
- Protocolos de nivel de aplicación que usan UDP: DNS, TFTP, etc.

4.3. Sockets

Cuando estamos trabajando en una red de ordenadores y queremos establecer una comunicación (recibir o enviar datos) entre dos procesos que se están ejecutando en dos máquinas diferentes de dicha red, ¿qué necesitamos para que esos procesos se puedan comunicar entre sí?

Supongamos que una de las aplicaciones solicita un servicio (cliente), y la otra lo ofrece (servidor).

Una misma máquina puede tener una o varias conexiones físicas a la red y múltiples servidores pueden estar escuchando en ese equipo. Si a través de una de esas conexiones físicas se recibe una petición por parte de un cliente ¿cómo se identifica qué proceso debe atender dicha petición? Es aquí donde surge el concepto de puerto, que permite tanto a TCP como a UDP dirigir los datos a la aplicación correcta de entre todas las que se están ejecutando en la máquina. Todo servidor, por tanto, ha de estar registrado en un puerto para recibir los datos que a él se dirigen (veremos en los ejemplos como se hace esto).

Los datos transmitidos a través de la red tendrán, por tanto, información para identificar la máquina mediante su dirección IP (si IPv4 32 bits, y si IPv6 128 bits) y el puerto (16 bits) a los que van dirigidos.

Los puertos:

- son independientes para TCP y UDP
- se identifican por un número de 16 bits (de 0 a 65535)
- algunos de ellos están reservados (de 0 a 1023), puesto que se emplean para servicios conocidos como HTTP, FTP, etc. y no deberían ser utilizados por aplicaciones de usuario.

Por tanto, un socket se puede definir como un extremo de un enlace de comunicación bidireccional entre dos programas que se comunican por la red (se asocia a un número de puerto). Se identifica por una dirección IP de la máquina y un número de puerto. Existe tanto en TCP como un UDP.

4.3.1 Sockets en Java

Java incluye la librería java.net para la utilización de sockets, tanto TCP como UDP. Será necesario importarla para su utilización.

Los sockets UDP son no orientados a conexión. Los clientes no se conectaran con el servidor sino que cada comunicación será independiente, sin poderse garantizar la recepción de los paquetes ni el orden de los mismos.

Clase *DatagramSocket*

Esta clase representa un socket UDP. Un objeto *DatagramSocket* es un “conector” a través del cual enviamos y recibimos paquetes UDP. De entre los varios constructores ofrecidos (http://docs.oracle.com/javase/7/docs/api/) destacaremos dos:

- *DatagramSocket()*: crea un socket en cualquier puerto local disponible. Se suele utilizar en el cliente puesto que no importa desde qué puerto se envía el paquete (puerto efímero).

- *DatagramSocket(int port)*: crea un socket en el puerto local especificado. Se suele utilizar en el servidor cuando se quieren recibir paquetes. De esta forma el *DatagramSocket* estará "escuchando" en el puerto especificado, preparado para recibir cualquier paquete entrante.

A continuación se muestra un ejemplo de uso tanto para la parte de servidor como para la parte de cliente.

```
DatagramSocket ds1 = new DatagramSocket(123);
/* Aquí usamos este DatagramSocket para recibir datos... */
/* ...              */
/* Hemos terminado, cerramos el socket      */
ds1.close();

DatagramSocket ds2 = new Datagramsocket();
/* Aquí lo usamos para transmitir datos...  */
/* ...            */
/* Hemos terminado, cerramos el socket      */
ds2.close();
```

Clase *DatagramPacket*

Esta clase representa a los paquetes de datos que vamos a enviar o recibir a través de los objetos *DatagramSocket*. A estos paquetes también se les denomina datagramas.

Para construir un paquete de este tipo es necesario indicar los datos que tiene que contener, la dirección IP del equipo al que se le envía y el puerto en el que está escuchando el programa al que se envía el paquete. En la documentación (http://docs.oracle.com/javase/7/docs/api/) se describen los diferentes constructores de la clase. El más genérico es el siguiente:

DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port)

El campo *buf* es el array de bytes que contiene los datos que se quieren enviar como mensaje, *offset* y *length* se refieren a los bytes de *buf* que queremos copiar en el paquete. La dirección se especifica mediante un objeto de tipo *InetAddress* mientras que el puerto se indica mediante un número entero.

En el siguiente ejemplo se envía un datagrama a una determinada dirección, suponiendo que tenemos un objeto *InetAddress* correctamente creado:

```
int tam = 1024;
InetAddress direcc = ...;
byte[] datos = new byte[tam];
int puerto = 543;
for (int n=0;n<tam;n++){
        /* Generamos los datos que vamos a enviar */
        datos[n] = ...;
        }

DatagramSocket ds = new DatagramSocket();
DatagramPacket dp = new DatagramPacket(datos, tam, direcc, puerto);

ds.send(dp);            /* Aquí enviamos el paquete     */
```

Si lo que queremos es recibir datos necesitamos reservar espacio para la información entrante (un array de bytes), poner un objeto *DatagramSocket* escuchando en un puerto y esperar a recibir un paquete mediante el método *receive()*.

```
int tam = 1024;
byte[] buffer = new byte[tam];

int puerto = 987;
```

```
DatagramSocket ds = new Datagramsocket(puerto);
DatagramPacket dp = new DatagramPacket(buffer,tam);

ds.receive(dp);
// Ahora tenemos en buffer la información que nos interesa
```

En este caso el método *receive()* es bloqueante puesto que el programa se detiene sin límite de tiempo hasta que se reciba un paquete. Se puede habilitar una temporización para que sea no bloqueante mediante el método *setSoTimeout(int timeout)*. En este caso la ejecución se detiene un tiempo máximo; si transcurrido ese tiempo no se ha recibido ningún paquete se lanza una excepción de tipo *SocketTimeoutException* y se continúa con la ejecución del programa. Un valor de *timeout* igual a 0 indica una espera bloqueante.

Clase *InetAddress*

Mediante un objeto de la clase *InetAddress* se especifica la dirección IP de un equipo, ya sea el cliente o el servidor. La forma de crear un objeto *InetAddress* es mediante el método *InetAddress.getByName(String)*, que recibe un nombre de host en notación alfanumérica (por ejemplo "www.ub.edu" o "161.116.100.2"). Si la dirección no existe o no puede ser encontrada se generará una excepción del tipo *UnknownHostException*. Por ejemplo, si queremos enviar un array de bytes al puerto 90 de la dirección "www.ub.edu" tendríamos que hacer:

```
int tam = ...;
int puerto = 90;
String maquina = "www.ub.edu";
byte[] buffer = new byte[tam];
/* ...              */
/* Generamos el contenido del buffer  */
/* ...              */
InetAddress direcc = InetAddress.getByName(maquina);
DatagramSocket ds = new DatagramSocket();
DatagramPacket dp = new DatagramPacket(buffer, tam, direcc, puerto);

ds.send(dp);            /* Aquí enviamos el paquete     */
```

Ejercicio 1

Implementar un cliente de eco UDP. El formato de ejecución será:

```
java ClienteUDP <máquina_servidor> <puerto_servidor> <mensaje>
```

En donde:

- ***máquina_servidor*** será el nombre (o la dirección IP) de la maquina en donde se está ejecutando un servidor de eco UDP.
- ***puerto_servidor*** será el puerto en el que está escuchando el servidor de eco UDP.
- ***mensaje*** será el mensaje que queremos enviar.

El programa deberá controlar las excepciones que puedan generar la resolución de nombres (*UnknownHostException*), la apertura del socket (*SocketException*) y cualquier otra excepción más genérica (*Exception*) en cuyo caso se imprimirá el error mediante el método *getMessage()* tal como se ve en el esquema siguiente:

```
try{

   /* ...              */
   /*    uso de InetAddress           */
   /* ...              */
} catch (UnknownHostException uhe){
     System.err.println("No puedo saber la dirección IP local : " + uhe);
 }


 try{
    /* ...              */
    /*   creación del socket          */
    /* ...              */

 } catch(SocketException se){
      System.err.println("Se ha producido un error al abrir el socket : " + se);
      System.exit(-1);
   }

  try{
     /* ...              */
     /*   recibir o enviar paquete tipo datagrama        */
     /* ...              */

  } catch (SocketTimeoutException e) {
      System.err.println("<xxx> segs sin recibir nada");
  } catch(Exception e){
       System.err.println("Se ha producido el error " + e);
  }
```