Práctica 5

Manejo de sockets UDP en Java

Grado Ingeniería Telemática
GSyC, Universidad Rey Juan Carlos

1. Introducción

En esta práctica vamos a trabajar con los sockets UDP que nos ofrece `Java`. `Java` proporciona tres clases para dar soporte a la comunicación via datagramas UDP, todas ellas contenidas en el paquete `java.net`. Estas clases son `DatagramSocket`, `DatagramPacket` y `MulticastSocket`.

2. Descripción de las clases *DatagramaPacket* y *DatagramSocket*

2.1. DatagramPacket

Esta proporciona constructores para crear instancias a partir de los datagramas recibidos y para crear instancias de datagramas que van a ser enviados.

- **Constructores para datagramas que van a ser enviados:** `DatagramPacket(byte[] buf, int length)` y `DatagramPacket(byte[] buf, int length, InetAddress address, int port)`. Estos constructores crean una instancia de datagrama compuesta por una cadena de bytes que almacena el mensaje, la longitud del mensaje, la dirección de Internet y el número de puerto local del conector destino, tal y como sigue: `MENSAJE - LONGITUD MENSAJE - DIRECCION IP - NÚMERO DE PUERTO`
- **Constructores para datagramas recibidos:** `DatagramPacket(byte[] buf, int offset, int length)` y `DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port)`. Estos constructores nos permiten crear instancias de los datagramas recibidos, especificando la cadena de bytes en la que alojar el mensaje, la longitud de la misma y el *offset* dentro de la cadena. Dentro de esta clase hay métodos para obtener los diferentes componentes de un datagrama, tanto recibido como enviado:

  `getData()`: para obtener el mensaje contenido en el datagrama.

  `getAddress()`: para obtener la dirección IP.

  `getPort()`: para obtener el puerto.

2.2. DatagramSocket

Esta clase maneja sockets para enviar y recibir datagramas UDP. Proporciona tres constructores:

- `DatagramSocket()`: constructor sin argumentos que permite que el sistema elija un puerto entre los que estén libres y selecciona una de las direcciones locales.
- `DatagramSocket(int port)`: constructor que toma un número de puerto como argumento, apropiado para los procesos que necesitan un número de puerto (servicios).
- `DatagramSocket(int port, InetAddress laddr)`: constructor que toma como argumentos el número de puerto y una determinada dirección local.

La clase `DatagramSocket` proporciona varios métodos, destacamos los más utilizados:

- `send(DatagramPacket p)` y `receive(DatagramPacket p)`: estos métodos sirven para transmitir datagramas entre un par de conectores. El argumento de send es una instancia de `DatagramPacket` conteniendo el mensaje y el destino. El argumento de `receive` es un `DatagramPacket` vacío en el que colocar el mensaje, su longitud y su origen. Ambos métodos pueden lanzar excepciones `IOException`.
- `setSoTimeout(int timeout)`: este método permite establecer un tiempo de espera límite. Cuando se fija un límite, el método `receive` se bloquea durante el tiempo fijado y después lanza una excepción `InterruptedIOException`.
- `connect(InetAddress address, int port)`: este método se utiliza para conectarse a un puerto remoto y a una dirección Internet concretos, en cuyo caso el conector sólo podrá enviar y recibir mensajes de esa dirección.

3. Ejemplo sencillo de Cliente/Servidor

En el siguiente ejemplo veremos una aplicación sencilla cliente/servidor en el que el servidor reenvía el mismo mensaje que le envían los diferentes clientes.

Para probar cómo funciona hay que arrancar primero el servidor, que se quedará a la escucha de peticiones, y después el cliente. Como se puede comprobar, el cliente envía una única petición y finaliza; el servidor por su parte se queda a la espera de nuevas peticiones.

3.1. Código del Cliente

Disponible en el fichero `ClienteUDP.java` adjunto a esta tarea.

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

public class ClienteUDP {

  // Los argumentos proporcionan el mensaje y el nombre del servidor
  public static void main(String args[]) {

    try {
      DatagramSocket socketUDP = new DatagramSocket();
      byte[] mensaje = args[0].getBytes();
      InetAddress hostServidor = InetAddress.getByName(args[1]);
      int puertoServidor = 6789;

      // Construimos un datagrama para enviar el mensaje al servidor
      DatagramPacket peticion =
        new DatagramPacket(mensaje, args[0].length(), hostServidor,
                           puertoServidor);

      // Enviamos el datagrama
      socketUDP.send(peticion);

      // Construimos el DatagramPacket que contendrá la respuesta
      byte[] bufer = new byte[1000];
      DatagramPacket respuesta =
        new DatagramPacket(bufer, bufer.length);
      socketUDP.receive(respuesta);

      // Enviamos la respuesta del servidor a la salida estandar
      System.out.println("Respuesta: " + new String(respuesta.getData()));
```

```
      // Cerramos el socket
      socketUDP.close();

    } catch (SocketException e) {
      System.out.println("Socket: " + e.getMessage());
    } catch (IOException e) {
      System.out.println("IO: " + e.getMessage());
    }
  }
}
```

3.2. Código del Servidor

El siguiente código, correspondiente al servidor, está disponible en el fichero `ServidorUDP.java` adjunto a la tarea:

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

public class ServidorUDP {
  public static void main (String args[]) {
    try {
      DatagramSocket socketUDP = new DatagramSocket(6789);
      byte[] bufer = new byte[1000];

      while (true) {
        // Construimos el DatagramPacket para recibir peticiones
        DatagramPacket peticion =
          new DatagramPacket(bufer, bufer.length);

        // Leemos una petición del DatagramSocket
        socketUDP.receive(peticion);

        System.out.print("Datagrama recibido del host: " +
                         peticion.getAddress());
        System.out.println(" desde el puerto remoto: " +
                         peticion.getPort());

        // Construimos el DatagramPacket para enviar la respuesta
        DatagramPacket respuesta =
          new DatagramPacket(peticion.getData(), peticion.getLength(),
                           peticion.getAddress(), peticion.getPort());

        // Enviamos la respuesta, que es un eco
        socketUDP.send(respuesta);
      }

    } catch (SocketException e) {
      System.out.println("Socket: " + e.getMessage());
    } catch (IOException e) {
      System.out.println("IO: " + e.getMessage());
    }
  }
}
```

4. MulticastSocket

La operación de *multicast* consiste en enviar un único mensaje desde un proceso a cada uno de los miembros de un grupo de procesos, de modo que la pertenencia a un grupo sea transparente al emisor, es decir, el emisor no conoce el número de miembros del grupo ni sus direcciones IP.

Un grupo multicast está especificado por una dirección IP clase D y un puerto. Las direcciones IP clase D están en el rango 224,0,0,0 a 239,255,255,255. Dentro de este rango existen direcciones reservadas, en concreto, la 224,0,0,1 y la 224,0,0,255. El resto de direcciones del rango pueden ser utilizadas por grupos temporales, los cuales deben ser creados antes de su uso y dejar de existir cuando todos los miembros lo hayan dejado.

`Java` proporciona una interfaz de datagramas para multicast IP a través de la clase `MulticastSocket`, que es una subclase de `DatagramSocket`, con la capacidad adicional de ser capaz de pertenecer a grupos multicast.

La clase `MulticastSocket` proporciona dos constructores alternativos:

- `MulticastSocket()`: que crea el socket en cualquiera de los puertos locales libres.
- `MulticastSocket(int port)`: que crea el socket en el puerto local indicado.

Un proceso puede pertenecer a un grupo multicast invocando el método `joinGroup(InetAddress mcastaddr)` de su socket multicast. Así, el socket pertenecerá a un grupo de multidifusión en un puerto dado y recibirá los datagramas enviados por los procesos en otros computadores a ese grupo en ese puerto. Un proceso puede dejar un grupo dado invocando el método `leaveGroup(InetAddress mcastaddr)` de su socket multicast.

Para enviar datos a un grupo multicast se utiliza el método `send(DatagramPacket p, byte ttl)`. Este es muy similar al de la clase `DatagramSocket`, la diferencia es que este datagrama será enviado a todos los miembros del grupo multicast. El parámetro *TTL* (*Time-To-Live*), lo pondremos siempre a 1, valor por defecto, para que sólo se difunda en la red local.

Para recibir datos de un grupo multicast se utiliza el método `receive(DatagramPacket p)` de la clase `DatagramSocket`, superclase de `MulticastSocket`. Es necesario pertenecer a un grupo para recibir mensajes multicast enviados a ese grupo, pero no es necesario para enviar mensajes.

4.1. Ejemplo sencillo de la clase *MulticastSocket*

Este programa de `Java` implementará las acciones que debe hacer un participante en un grupo multicast; esto es, primero se unirá al grupo y, a continuación, enviará un mensaje al mismo; después se quedará a la espera de recibir mensajes de otros participantes del grupo hasta que el mensaje recibido sea `Adios`. Para simplificar el programa, consideramos el caso de que tanto el mensaje que envía el participante como la dirección IP multicast del grupo se pasan como parámetros.

Al igual que los anteriores códigos, este también está disponible en el fichero `MiembroMulticast.java` adjunto a la tarea:

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

public class MiembroMulticast {
  public static void main(String args[]) {
    // args[0] es el mensaje enviado al grupo y args[1] es la direccion del grupo
    try {
      InetAddress grupo = InetAddress.getByName(args[1]);
      MulticastSocket socket = new MulticastSocket(6789);
```

```
      // Se une al grupo
      socket.joinGroup(grupo);

      // Envia el mensaje
      byte[] m = args[0].getBytes();
      DatagramPacket mensajeSalida =
new DatagramPacket(m, m.length, grupo, 6789);
      socket.send(mensajeSalida);

      byte[] bufer = new byte[1000];
      String linea;

      // Se queda a la espera de mensajes al grupo, hasta recibir "Adios"
      while (true) {
DatagramPacket mensajeEntrada =
  new DatagramPacket(bufer, bufer.length);
socket.receive(mensajeEntrada);
linea = new String(mensajeEntrada.getData(), 0, mensajeEntrada.getLength());
System.out.println("Recibido:" + linea);
if (linea.equals("Adios")) break;
      }

      // Si recibe "Adios" abandona el grupo
      socket.leaveGroup(grupo);
    } catch (SocketException e) {
      System.out.println("Socket:" + e.getMessage());
    } catch (IOException e) {
      System.out.println("IO:" + e.getMessage());
    }
  }
}
```

Para probar este programa crearemos varias instancias de `MiembroMulticast` y veremos cómo cada vez que se crea un nuevo participante su mensaje llega a todos los demás. Para que todos los participantes abandonen el grupo, crearemos una instancia de `MiembroMulticast` con el mensaje `Adios`. Un caso de uso podría ser el siguiente:

1. **Desde terminal 1**: `java MiembroMulticast "hola" "224.0.0.3"` (Salida en T1: Recibido:hola)
2. **Desde terminal 2**: `java MiembroMulticast "que" "224.0.0.3"` (Salidas en T1 y T2: Recibido:que)
3. **Desde terminal 3**: `java MiembroMulticast "tal" "224.0.0.3"` (Salidas en T1, T2 y T3: Recibido:tal)
4. **Desde terminal 4**: `java MiembroMulticast "Adios" "224.0.0.3"` (Salidas en T1, T2, T3 y T4: Recibido:Adios -y fin de todos los procesos-)

5. Ejercicios

Para la realización de esta práctica habrá que tomar como punto de partida los anteriores códigos y realizar las siguientes tareas:

1. **Ejercicio 1**: modificar el programa cliente para que envíe al servidor los mensajes que introduce un usuario por la entrada estándar.

2. **Ejercicio 2**: modificar lo que se considere oportuno para que en la aplicación cliente/servidor el servidor proporcione la hora y el dia a los clientes que lo soliciten. Para ello, el programa cliente realizará una petición al servidor y esperará la respuesta un tiempo limitado (5000 milisegundos). Si recibe la respuesta, enviará a la salida estándar el dia y la hora proporcionados por el servidor. Si después de ese tiempo no recibe una respuesta, enviará a la salida estándar un mensaje de error.

3. **Ejercicio 3**: realizar un chat utilizando multicast. Como en cualquier chat, deberán aparecer los mensajes enviados por los diferentes participantes en el mismo. Cada participante podrá escribir sus propios mensajes, de forma que cada vez que se escriba una línea se envíe a todos los participantes del chat.

6. Entrega

Para implementar cada ejercicio deberás crear su correspondiente proyecto. Los nombres de los proyectos serán, respectivamente:

- `tulogin-p5-ej1`
- `tulogin-p5-ej2`
- `tulogin-p5-ej3`

Para la entrega de esta práctica, deja todo el código fuente de los tres proyectos en un único fichero `udp.zip` que deberás dejar adjunto a la tarea (Moodle).