Seminario 1 - Comunicación con UDP

Seminario práctico de laboratorio sobre temas de comunicación con UDP/IP en Java.

Tabla de contenidos

1 Presentación del seminario.

La programación de aplicaciones cliente-servidor en Java mediante sockets UDP plantea conceptualmente problemas muy similares a los que hay que resolver con sockets tipo TCP. Sin embargo, al utilizar sockets UDP se ponen de manifiesto cietas dificultades técnicas que no son frecuentes.

En este seminario a desarrollar en una sola sesión, el profesor expondrá en el aula

- los principios tecnológicos básicos precisos para construir aplicaciones sencillas usando sockets UDP en Java.
- A continuación se expondrá un caso de desarrollo con el protocolo sencillo mostrado en la Lección 2, en el caso monoenhebrado.
- Seguídamente un caso de desarrollo de sockets UDP con sesiones, y finalmente,
- durante los últimos 10 minutos, se presentarán algunas cuestiones para evaluar el seguimiento de la actividad desarrollada en el aula.

Presta especial atención en el aula a los que se presenta, toma notas, reflexiona y pregunta si te quedan dudas!!

2 Porqué UDP, si tenemos TCP.

¿Porqué en los 90 se decidió usar HTTP con conexión TCP por recurso, y no una sesión por cliente web?

- No fiabilidad de UDP.
- Fragilidad de las conexiones TCP a largo plazo.
- Envío de datos de tamaño desconocido.
- Sencillez de la programación con hilos en servidores emergentes.

Razones para usar TCP+multienhebrado en cliente-servidor:

- En el servidor, cada hilo mantiene la sesión del cliente.
- La conversación es muy sencilla mediante primitivas de manejo de streams.
- Podemos leer y escribir una cantidad indeterminada de datos.
- En el servidor, es fácil de gestionar la espera y el bloqueo con varios hilos, pues el resto de hilos no deberían verse afectados.

Razones para no usar TCP+multienhebrado en cliente-servidor:

- Los hilos y las conexiones consumen recursos fijos, para servicios que puede no necesitar tanto.
- Estos recursos están limitados por máquina (espacio y tiempo).
- Las conexiones son frágiles y hay que reconstruirlas a menudo.

Razones para no usar UDP en cliente-servidor:

- No es fiable.
- Se transmiten tramas de un tamaño limitado.
- Hay que recordar la sesión expresamente.

Razones para usar UDP en cliente-servidor:

- Con un único hilo podemos gestionar muchas (muchas) conversaciones.
- UDP ya no es lo que era, y es más fiable.
- Si necesitamos hilos podemos crearlos cuando sea necesario.
- Es más ágil.

Item más: "¿Qué pasa con el streaming?"

3 Sockets UDP en Java (cliente-servidor Echo).

Varios tutoriales te dirán que usar sockets UDP en Java es sencillo, pero ninguno te dirá que es más fácil que usar sockets TCP, por que sería mentir a la verdad. Lo que sí es cierto es que usar datagramas tiene la cercanía de la comprensibilidad de estar manejando pequeños trozos de datos que se reducen a arrays de bytes.

Por contra, existen inconvenientes cuando queremos manejar cualquier tipo de dato que no sea un array de bytes, y tenémos que vérnoslas con un montón de código boiler-plate, y casuística. Comencemos con el caso más sencillo, un cliente y un servidor monoenhebrado que reproducen el ejemplo Echo de siempre.

```
package UDPClient;

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

/*
* Cliente sencillo de terminal, tipo ping-pong
*/
public class ClienteSimple {

    public static void main(String[] args) throws IOException {
        String host = "localhost";
        String linea;
        int port = 1919;
        InetAddress ia = InetAddress.getByName(host);
        DatagramSocket socket = new DatagramSocket();
        socket.connect(ia, port);


        Reader r1 = new InputStreamReader(System.in);
        BufferedReader teclado = new BufferedReader(r1); /* teclado */

        while ((linea = teclado.readLine()) != null) {
            byte[] dataOut = linea.getBytes();
            DatagramPacket output =
                    new DatagramPacket(dataOut, dataOut.length, ia, port);
            socket.send(output);
            byte[] dataIn = new byte[160];
            DatagramPacket input = new DatagramPacket(dataIn, dataIn.length);

            socket.receive(input);
            linea = new String(input.getData());
            System.out.println("Echo: "+linea);
            if (linea.equals("Adios.")) System.exit(0);
        }
    }
}
```

El servidor también es muy sencillo, y nos limitamos a devolver a cada cliente que nos envía un datagrama, el eco del suyo. Visto de esta forma, se comporta como un servidor multienhebrado que mantiene varias sesiones abiertas con varios clientes, pero sin tanta parafernalia.

```java
package UDPServer;

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

/*
* Servidor monoenhebrado sencillo de terminal tipo ping-pong
*/
public class UDPEchoServer {
  static final int BUFFERSIZE = 256;

  public static void main(String[] args) {
    DatagramSocket sock;
    DatagramPacket pack = new DatagramPacket(new byte[BUFFERSIZE],
        BUFFERSIZE);
    try {
      sock = new DatagramSocket(1919);
    } catch (SocketException e) {
      System.out.println(e);
      return;
    }
    // echo back everything
    while (true) {
      try {
        sock.receive(pack);
        System.out.println("["+pack.getAddress().getHostAddress()+":"+
                pack.getPort()+"> "+
                (new String(pack.getData())));
        sock.send(pack);
      } catch (IOException ioe) {
        System.out.println(ioe);
      }
    }
  }
}
```

Los desafíos que se nos plantean a corto plazo son muy simples:

- Conseguir enviar y recibir objetos Java.
- Conseguir que los diversos clientes mantengan una sesión distinta con el servidor.

¿Te atreves, a conseguirlo sin pasar a las siguientes páginas? Es difícil pero compensa. Primero veamos cómo podemos mantener diversas sesiones en nuestro cliente.servidor.

4 Sesiones en un servidor monoenhebrado.

Cuando utilizamos un mismo hilo para mantener varias conexiones abiertas con distintos clientes, desde el servidor, es necesario conmutar de contexto con cada uno de ellos convenientemente para que a cada cliente le de la impresión de estar conectado "aisladamente" al servidor, sin que interfieran los demás, en apariencia.

Al contexto de la conversación y el protocolo que mantiene cada cliente con el servidor lo denominamos "sesión", y se da tanto en el caso de la comunicación TCP como UDP. Obviamente, mantener sesiones es mucho más sencillo cuando disponemos de Threads, pues el contexto de cada hilo nos permite mantener ciertas variables de contexto aisladas, que no interfieren unas con otras. Cuando manejamos un solo hilo para todas las conexiones las cosas se complican un poco, a cambio de hacer un sistema más ágil y rápido.

Necesitamos un pequeño bean para mantener los datos de la sesión. Esta pequeña clase se puede ampliar todo lo que necesitemos para almacenar un montón de datos del contexto del servicio.

```
package UDPServerSession;

import java.net.SocketAddress;

/**
 * Sesion donde almacenamos los datos del contexto que necesitemos.
 * @author cllamas
 */
public class Sesion {
    final SocketAddress sa;
    static private int idCounter = 0;
    private final int id;
    Sesion(SocketAddress sa) {
        this.sa = sa;
        this.id = ++Sesion.idCounter;
    }

    public int getId() {
        return id;
    }
}
```

A partir de la clase anterior, el servidor resulta muy sencillo.

```
package UDPServerSession;

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.SocketAddress;
import java.net.SocketException;
import java.util.HashMap;
import java.util.LinkedHashMap;

/**
 * UDPEchoServerSession mantiene la sesión en función del origen
 *                      del datagrama.
 * @author cllamas
 */
public class UDPEchoServerSession {
  static final int BUFFERSIZE = 256;
  static final HashMap<SocketAddress,Sesion> sesiones = new HashMap();

  public static void main(String[] args) {
    DatagramSocket sock;
    try {
      sock = new DatagramSocket(1919);
    } catch (SocketException e) {
      System.out.println(e);
      return;
    }
    // echo de todo sin importar el origen.
    while (true) {
      try {
        DatagramPacket pack = new DatagramPacket(new byte[BUFFERSIZE], BUFFER
SIZE);

        sock.receive(pack);
        Sesion s;
        if ((s = sesiones.get(pack.getSocketAddress())) == null) {
            s = new Sesion(pack.getSocketAddress());
            sesiones.put(pack.getSocketAddress(), s);
        }
        System.out.println("Sesion: "+s.getId());

        System.out.println("<"+s.getId()+">["
                +pack.getAddress().getHostAddress()+":"+pack.getPort()+"] "+
                (new String(pack.getData())));
        sock.send(pack);
      } catch (IOException ioe) {
        System.out.println(ioe);
      }
    }
  }
}
```

Observese el tipo de estructura que mantiene la sesión, de un modo sencillo y transparente. La complicación vendrá después cuando queramos añadir funcionalidades a este servidor, como enviar y recibir objetos cualesquiera. Ese es nuestro siguiente desafío.

5 Serialización con sockets UDP.

Si queremos enviar y recibir objetos mediante los datagramas tenemos básicamente dos posibilidades:

1. Descomponer y recomponer los datos que enviamos y recibimos en cadenas de bytes. En otras palabras, empaquetar (serializar) y desempaquetar (deserializar) los datos sobre byte[ ], a mano.
2. Como se verá en clase, los expertos ponen montones de pegas a esta alternativa que no funciona más que con los casos más sencillos. Los proyectos más habituales en Java se basan en middleware que soporta:
    1. XML
    2. JSON
    3. Objetos Java.

Nosotros vamos a apoyarnos en la serialización de Java, al coste de incluir un poco de complicación, pero que al final acabamos de incorporando a nuestro proyecto en forma de código boilerplate.

Precaución, el siguiente código que vas a ver es demostrativo y requiere un poquito de afinación para funcionar redondo, pero es suficiente.

Como es lógico, vamos a aprovechar para ilustrar el caso de nuestro Protocolo de Colas de Strings, y empezaremos por el cliente. El servidor aprovecha la mayor parte del trabajo realizado. Las clases que utilizaremos son:

- Por parte del paquete "Mensajes"
    - Primitive (igual que en la anterior práctica),
    - MensajeProtocolo (igual que en la anterior práctica), y
    - UDPHelper para ayudarnos con la serialización.
- Por parte del Cliente lo concentraremos todo en una misma clase "Cliente" reformada.

```java
package Mensajes;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectInputStream;
import java.io.ObjectOutput;
import java.io.ObjectOutputStream;

/**
  * Clase de ayuda para la serialización.
  * depende de MensajeProtocolo aunque podría ser genérica.
  * @author cllamas
  */
public class UDPHelper {
    private final ByteArrayOutputStream bos;
    private final ByteArrayInputStream  bis;

    public UDPHelper(byte[] entrada) {
        bos = new ByteArrayOutputStream();
        bis = new ByteArrayInputStream(entrada);
    }

    public byte[] aBytes(MensajeProtocolo mensaje) {
        bos.reset();
        try (ObjectOutput out = new ObjectOutputStream(bos)) {
            out.writeObject(mensaje);
            return bos.toByteArray();
        } catch (IOException ioe) {
            return null;
        }
    }

    public MensajeProtocolo aMensaje() {
        bis.reset(); /* sólo leemos un objeto en cada datagrama */

        try (ObjectInput in = new ObjectInputStream(bis)) {
            return (MensajeProtocolo) in.readObject();
        } catch (IOException | ClassNotFoundException ex) {
            return null;
        }
    }
}
```

Estudia atentamente los dos métodos y su signatura (lo explicaré en clase). A tener en cuenta está, que hay que tener especial cuidado con la persistencia del objeto cuando empleamos los constructores, y el reinicio de los streams. (Sí, empleamos streams para convertir los objetos!!!)

Observa ahora la clase Cliente.

```
package ProtocolClient;

import Mensajes.UDPHelper;
import Mensajes.MensajeProtocolo;
import Mensajes.Primitive;
import java.io.*;
import java.net.*;

/**
 * Cliente0 con un escenario de uso del protocolo de la Lección 2.
 * @author cllamas
 */
public class Cliente0 {

    static final private int PUERTO = 1919;
    static final private int MAXDATAGRAMA = 1024;

    public static void main(String[] args) throws IOException {
        String host = "localhost";
        String linea;

        byte[] bytesSalida  = null;
        byte[] bytesEntrada = new byte[MAXDATAGRAMA];
        UDPHelper serialHandler = new UDPHelper(bytesEntrada);
        DatagramPacket output = null;
        DatagramPacket input  = null;

        DatagramSocket socket = new DatagramSocket();
        InetAddress iaServer = InetAddress.getByName(host);
        socket.connect(iaServer, PUERTO);

        try {
            System.out.println("Pulsa <Enter> para comenzar");
System.in.read();
/* Caso 1 */
            /* >> */ bytesSalida = serialHandler.aBytes(new
MensajeProtocolo(Primitive.HELLO, "Pedro"));
            output = new DatagramPacket(bytesSalida, bytesSalida.length, iaSe
rver, PUERTO);
            socket.send(output);

/* << */    input       = new DatagramPacket(bytesEntrada, bytesEntrada.lengt
h);
            socket.receive(input);
            System.out.println((MensajeProtocolo) serialHandler.aMensaje());

            System.out.println("Pulsa <Enter> para continuar"); System.in.rea
d();
/* Caso 2.1 */
/* >> */    bytesSalida = serialHandler.aBytes(new
MensajeProtocolo(Primitive.PUSH, "Estamos en Estambul."));
            output      = new DatagramPacket(bytesSalida, bytesSalida.length,
 iaServer, PUERTO);
            socket.send(output);

/* << */    input       = new DatagramPacket(bytesEntrada, bytesEntrada.lengt
h);
            socket.receive(input);
            System.out.println((MensajeProtocolo) serialHandler.aMensaje());

            System.out.println("Pulsa <Enter> para continuar"); System.in.rea
d();
/* Caso 2.2 */
/* >> */    bytesSalida = serialHandler.aBytes(new
MensajeProtocolo(Primitive.PUSH, "Estamos en Lisboa."));
            output      = new DatagramPacket(bytesSalida, bytesSalida.length,
```

```
 iaServer, PUERTO);
            socket.send(output);

/* << */    input       = new DatagramPacket(bytesEntrada, bytesEntrada.lengt
h);
            socket.receive(input);
            System.out.println((MensajeProtocolo) serialHandler.aMensaje());

            System.out.println("Pulsa <Enter> para continuar"); System.in.rea
d();
/* Caso 3 */
/* >> */    bytesSalida = serialHandler.aBytes(new
MensajeProtocolo(Primitive.PULL));
            output      = new DatagramPacket(bytesSalida, bytesSalida.length,
 iaServer, PUERTO);
            socket.send(output);

/* << */    input       = new DatagramPacket(bytesEntrada, bytesEntrada.lengt
h);
            socket.receive(input);
            System.out.println((MensajeProtocolo) serialHandler.aMensaje());

            System.out.println("Pulsa <Enter> para continuar"); System.in.rea
d();
/* Caso 4 */
/* >> */    bytesSalida = serialHandler.aBytes(new
MensajeProtocolo(Primitive.PULL_WAIT));
            output      = new DatagramPacket(bytesSalida, bytesSalida.length,
 iaServer, PUERTO);
            socket.send(output);

/* << */    input       = new DatagramPacket(bytesEntrada, bytesEntrada.lengt
h);
            socket.receive(input);
            System.out.println((MensajeProtocolo) serialHandler.aMensaje());

            System.out.println("Pulsa <Enter> para finalizar"); System.in.rea
d();


            /**
             * * aquí se supone que tiene que llegar otro cliente e insertar
 un
             * mensaje en la cola
             */
            /* FIN del escenario 1 */
        } catch (IOException e) {
            System.err.println("Cliente: Error de apertura o E/S sobre objeto
s: " + e);
        } catch (Exception e) {
            System.err.println("Cliente: Excepción. Cerrando Sockets: " + e);
        } finally {
            socket.close();
        }
    }
}
```

Este tipo de código farragoso es algo a los que nos tenemos que ir acostumbrando. Obviamente, si esto fuera algo más que una prueba, refinaríamos las clases de ayuda, y haríamos este cliente mucho más comprensible, apoyándonos también de pruebas unitarias.

Veamos a continuación cómo queda el servidor.

5.1 El servidor.

El servidor resulta sorprendentemente sencillo, aunque hay que tener en cuenta que es del tipo monoenhebrado. Veamos.

```
package ProtocolServer;

import Mensajes.MensajeProtocolo;
import Mensajes.Primitive;
import Mensajes.UDPHelper;
import UDPServer.*;
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.SocketException;

/**
 * UDPProtocolServer que obedece al protocolo de la Lección 2.
 * Es Monoenhebrado, emplea sockets UDP y serialización.
 * @author cllamas
 */
public class UDPProtocolServer {

   static final private int PUERTO = 1919;
   static final private int MAXDATAGRAMA = 1024;

   public static void main(String[] args) throws IOException {
     String linea;
     MensajeProtocolo me;
     MensajeProtocolo ms;
     ColaStrings cola = new ColaStrings();

     byte[] bytesSalida = null;
     byte[] bytesEntrada = new byte[MAXDATAGRAMA];
     UDPHelper serialHandler = new UDPHelper(bytesEntrada);
     DatagramPacket output = null;
     DatagramPacket input = null;

     DatagramSocket socket;
     InetAddress iaServer = null;

     try {
        socket = new DatagramSocket(PUERTO);
     } catch (SocketException e) {
        System.out.println("Puerto ocupado en el servidor: " + e);
        return;
     }
     // attend protocol primitives.
     while (true) {
        try {
           /* Recepción del mensaje del protocolo */
           /* << */ input = new DatagramPacket(bytesEntrada, bytesEntrada.len
gth);
                           socket.receive(input);

           me = (MensajeProtocolo) serialHandler.aMensaje();
           System.out.println("<<" + me);

           switch (me.getPrimitive()) {
              case HELLO:
                 ms = new MensajeProtocolo(Primitive.HELLO, "Hola desde el se
rvidor");
                 break;
              case PUSH:
                 cola.push(me.getMessage());
                 ms = new MensajeProtocolo(Primitive.PUSH_OK);
                 break;
              case PULL:
                 synchronized (cola) {
                   if (cola.estaVacia()) {
```

```
                    ms = new MensajeProtocolo(Primitive.NOTHING);
                  } else {
                    ms = new MensajeProtocolo(Primitive.PULL_OK,
cola.pop());
                  }
               }
               break;
            case PULL_WAIT:
               ms = new MensajeProtocolo(Primitive.PULL_OK, cola.pop());
               break;
            default:
               ms = new MensajeProtocolo(Primitive.NOTUNDERSTAND);
               // break;
         }
         /* Envío del mensaje del protocolo */
         /* >> */ bytesSalida = serialHandler.aBytes(ms);
         System.out.println("He convertido a bytes: "+ms);
         output = new DatagramPacket(bytesSalida, bytesSalida.length,
input.getAddress(), input.getPort());
         System.out.println("He creado el datagrama y voy a enviar");
         socket.send(output);
      } catch (IOException ioe) {
         System.out.println(ioe);
      }
   }
}
```

Observa bien, cómo nos asentamos en bloques estándar parecidos a los que emplea el cliente

6 Multienhebrado callable() en servidores monoenhebrados.

Por último, una pequeña rareza que podría omitirse si no fuera muy importante. En los servidores monoenhebrados, con todo lo ágiles que son, aparecen de vez en cuando tareas que hay que despachar asíncronamente. Piensa por un momento en el servidor anterior de nuestro protocolo; lo único que impide a nuestro servidor atender a miles de clientes a la vez es... que se bloquea en la primitiva "PULLWAIT" de modo que no hay forma de sacarle de ese bloque por muchos clientes que intenten insertar mensajes en la cola. Un servidor elegante despacharía esta primitiva en un hilo aparte previendo que su realización puede bloquear el servidor o, en otro caso, requerir mucho tiempo e interferir en el bucle de procesamiento.

Este es un pequeño ejemplo de utilización de callable, sobre el ejemplo Echo con UDP de un par de capítulos atrás.

El servidor es sumamente sencillo (dentro de los objetivos didácticos), y realiza invocaciones a Futuro por cada mensaje, no por cada cliente, sin esperar la finalización de los hilos (join) necesariamente.

```
package UDPServerSessionCallable;

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.SocketAddress;
import java.net.SocketException;
import java.util.HashMap;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

/**
 * UDPEchoServerSessionCallable mantiene sesiones por cliente e invoca
 * hilos adicionales por cada invocación, no por cada cliente.
 * @author cllamas
 */
public class UDPEchoServerSessionCallable {
  static final int BUFFERSIZE = 256;
  static final HashMap<SocketAddress,SesionCallableEcho> sesiones = new HashM
ap();

  public static void main(String[] args) {
    ExecutorService pool = Executors.newFixedThreadPool(3);
    DatagramSocket sock;
    try {
      sock = new DatagramSocket(1919);
    } catch (SocketException e) {
      System.out.println(e);
      return;
    }
    // echo back everything
    while (true) {
      try {
        DatagramPacket pack = new DatagramPacket(new byte[BUFFERSIZE],BUFFERS
IZE);

        sock.receive(pack);
        SesionCallableEcho s;

        if ((s = sesiones.get(pack.getSocketAddress())) == null) {
            s = new SesionCallableEcho(pack.getSocketAddress(), sock);
            sesiones.put(pack.getSocketAddress(), s);
        }
        s.putDP(pack);
        Future<Boolean> tarea = pool.submit(s);

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

Los sirvientes son callables, lo que quiere decir, que cada petición va a formar parte de una cola de atención, manejada por el Executor y que se irá despachando según se pueda, por cada petición.

```java
package UDPServerSessionCallable;

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.SocketAddress;
import java.util.concurrent.Callable;

/**
 * SesionCallableEcho que se invoca por petición, no por cliente.
 * @author cllamas
 */
public class SesionCallableEcho implements Callable<Boolean> {
    final SocketAddress sa;
    static private int idCounter = 0;
    private final int id;
    private DatagramPacket dp;
    private DatagramSocket ds;

    SesionCallableEcho(SocketAddress sa, DatagramSocket ds) {
        this.sa = sa;
        this.ds = ds;
        this.id = ++SesionCallableEcho.idCounter;
    }

    @Override
    public Boolean call() {
        String texto = new String(dp.getData());
        if (!texto.equals("Adios.") ) {
            System.out.println("<"+id+">["+sa+"]: "+texto);
            try { this.ds.send(dp); }
            catch (IOException ioe) {
                System.err.println("SessionCallableEcho: IOE: "+ioe);
            }
            return Boolean.TRUE;
        } else {
            return Boolean.FALSE;
        }
    }
    public int getId() {
        return id;
    }
    public void putDP(DatagramPacket dp) {
        this.dp = dp;
    }
}
```

7 Conclusión.

Faltan muchas propuestas por concluir, entre las que se encuentra transplantar el esquema de Procesadores planteado al final de la Lección 2, pero en versión UDP, que queda, si cabe, mucho más interesante y elegante.

En esta lección has podido ver cómo es en líneas generales la comunicación UDP sencilla, con serialización, con multienhebrado, y como se aplica a un protocolo sencillo. Ahora tu parte es completar con unas cuantas cuestiones sencillas planteadas por el profesor para comprobar el aprovechamiento del seminario.