Sitio:	Campus Virtual - Curso 2020-2021
Curso:	SISTEMAS DISTRIBUIDOS
Libro:	Seminario 1 - Comunicación con UDP
Imprimido por:	LLAMAS BELLO, CESAR
Día:	martes, 16 de marzo de 2021, 17:32

Seminario práctico sobre la utilización de sockets UDP en Java.

Tabla de contenidos

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, para ello veremos cosas muy interesantes como:

- Un ejemplo **ECHO** que **con un sólo hilo** habla con muchos clientes UDP.
- El ejemplo **ECHO** que recuerda a los clientes con el concepto de ¡**Sesión**!
- ¿Serializamos? Te enseñamos una clase de utilidad que lo hace todo. Y lo aplicamos a las clases `Primitiva` y `MensajeProtocolo`.
- ¿Te gustaría ver cómo se soporta un `envía()` / `recibe()` asíncrono/asíncrono? Aquí viene una variante de nuestro servicio de colas de mensajes en versión asíncrona ilustrando el concepto de **Futuro completable** y **Promesa** .

¿Qué debes hacer tú?

- Atiende bien en clase porque al final tendrás que responder a algunas preguntas, y sobre todo...
- prueba el código que te proporcionamos en este documento!!.
- Y si te quedan dudas... pregunta.

Obviamente hay mucho código implicado. *No te preocupes* porque aparecerá mágicamente en la página principal del Campus Virtual o al lado del documento en OneNote.

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?"

Sockets UDP en Java (cliente-servidor Echo).

Varios tutoriales te dirán que usar sockets UDP en Java es sencillo, pero decir que es más fácil que usar sockets TCP sería mentir a la verdad. A pesar de ello, el usar datagramas nos resulta más familiar a la forma en que se produce la entrega y rutado de paquetes en una red a bajo nivel, y nos permite pequeños fragmentos de datos bajo la forma de arrays de bytes.

Por contra, los problemas surgen cuando necesitamos gestionar cualquier tipo de dato más complejo que dicho array de bytes. El resultado es tener que vérselas 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.

Clase `ClienteSimple`

```
1  package udp.echo;
2
3  public class ClienteSimple {
4    public static void main(String[] args) throws java.io.IOException {
5      String host = "localhost";
6      String linea;
7      int port = 1919;
8      java.net.InetAddress ia = java.net.InetAddress.getByName(host);
9      java.net.DatagramSocket socket = new java.net.DatagramSocket();
10     socket.connect(ia, port);
11
12
13     java.io.Reader r1 = new java.io.InputStreamReader(System.in);
14     java.io.BufferedReader teclado = new java.io.BufferedReader(r1); //
   teclado
15
16     while ((linea = teclado.readLine()) ≠ null) {
17       byte[] dataOut = linea.getBytes();
18       java.net.DatagramPacket output =
19                 new java.net.DatagramPacket(dataOut, dataOut.length, ia,
   port);
20       socket.send(output);
21       byte[] dataIn = new byte[160];
22       java.net.DatagramPacket input =
23             new java.net.DatagramPacket(dataIn, dataIn.length);
24
25       socket.receive(input);
26       linea = new String(input.getData());
27       System.out.println("Echo: "+linea);
28       if (linea.startsWith("Adios.")) System.exit(0);
29     }
30   }
31 }
```

La clase `ClienteSimple` servirá para poner a prueba el siguiente servidor y algún otro más que sigue en este seminario, por lo que guárdalo a mano porque lo usarás en otro momento posterior.

La mayor parte del trabajo del cliente consiste en **asignar correctamente los canales**, a parte de la E/S basada en datagramas propiamente. En la línea 9 puede verse cómo se inicia un socket de datagrama asignado a un puerto anónimo. La conexión de la línea 10 no tiene ningún efecto práctico en este caso pues sólo se va a realizar una conexión con un servidor muy concreto, que se ha establecido en la máquina local y en un puerto conveniente (1919).

La línea 17 detalla la forma sencilla en que **serializamos un `String` en Java** mediante un método de la propia clase. A partir de él creamos un `DatagramPacket` *ad-hoc* que se envía en la línea 20. En la línea 21 se crea un espacio de almacenamiento suficiente para el datagrama de la línea 23 que se emplea en la línea 25 donde se realiza la recepción correspondiente. La **deserializacion** utiliza un método de `byte[]` y un constructor de `String`.

El servidor es muy sencillo y **sólo tiene un hilo**, donde nos limitamos a devolver a cada cliente que nos envía un datagrama, el eco del suyo. Puede mantener conversaciones con varios clientes concurrentemente, pero con mucha menos dificultad que su homólogo multienhebrado que emplea TCP.

Clase `UDPEchoServer`

```
1  package udp.echo;
2
3  public class UDPEchoServer {
4    static final int BUFFERSIZE = 256;
5
6    public static void main(String[] args) {
7      java.net.DatagramSocket sock;
8      java.net.DatagramPacket pack =
9       new java.net.DatagramPacket(new byte[BUFFERSIZE], BUFFERSIZE);
10     try {
11       sock = new java.net.DatagramSocket(1919);
12     } catch (java.net.SocketException e) {
13       System.out.println(e);
14       return;
15     }
16     // echo back everything
17     while (true) {
18       try {
19         sock.receive(pack);
20         System.out.println("["+pack.getAddress().getHostAddress()+":"+
21                 pack.getPort()+"> "+
22                 (new String(pack.getData())));
23         sock.send(pack);
24       } catch (java.io.IOException ioe) {
25         System.out.println(ioe);
26       }
27     }
28   }
29 }
```

Partiendo de este sencillo escenario podemos plantearnos muchos desafíos:

- 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.

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.

La **sesión contiene el contexto de la conversación entre el cliente y el servidor** ya sea TCP o UDP. Obviamente, con Threads, es más sencillo pues las variables de cada Thread no interfieren entre sí. Con un solo hilo para todas las conexiones las cosas se complican un poco, pero a cambio el sistema es más ágil y rápido conmutando de contexto.

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.

Clase `Sesion`

```java
package udp.common;

public class Sesion {
    final java.net.SocketAddress sa;
    static private int idCounter = 0;
    private final int id;
    public Sesion(java.net.SocketAddress sa) {
        this.sa = sa;
        this.id = ++Sesion.idCounter;
    }

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

La clase `Sesion` guardará la información de estado, con un contador que se incrementará según los clientes se conectan con una IP y/o puerto distinto (`SocketAddress`). En su interior pueden reflejarse **tantas variables de instancia como sean necesarias** teniendo en cuenta que cada vez que el bucle principal recibe una trama y la decodifica va a ser capaz de identificar el origen de la trama y adaptarse al estado de la conexión con el cliente origen del dato.

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

Clase `UDPEchoSessionServer`

```
1  package udp.server;
2
3  public class UDPEchoSessionServer {
4    static final int BUFFERSIZE = 256;
5    static final java.util.HashMap<java.net.SocketAddress,udp.common.Sesion>
6          sesiones = new java.util.HashMap<>();
7
8    public static void main(String[] args) {
9      java.net.DatagramSocket sock;
10     try {
11       sock = new java.net.DatagramSocket(1919);
12     } catch (java.net.SocketException e) {
13       System.out.println(e);
14       return;
15     }
16     // echo de todo sin importar el origen.
17     while (true) {
18       try {
19         java.net.DatagramPacket pack =
20            new java.net.DatagramPacket(new byte[BUFFERSIZE], BUFFERSIZE);
21
22         sock.receive(pack);
23         udp.common.Sesion s;
24         if ((s = sesiones.get(pack.getSocketAddress())) == null) {
25             s = new udp.common.Sesion(pack.getSocketAddress());
26             sesiones.put(pack.getSocketAddress(), s);
27         }
28         String linea = "<"+s.getId()+">
   ["+pack.getAddress().getHostAddress()+
29                 ":"+pack.getPort()+"] "+(new String(pack.getData()));
30         System.out.println(linea);
31
32     byte[] buffer = linea.getBytes();
33
34     pack.setData(buffer); // no es preciso setLength para enviar.
35
36         sock.send(pack);      // se aprovecha el mismo datagrama.
37       } catch (java.io.IOException ioe) {
38         System.out.println(ioe);
39       }
40     }
41   }
42 }
```

Observese el tipo de estructura que mantiene la sesión, de un modo sencillo y transparente. Las instancias de `Sesión` se almacenan en un mapa asociativo (`sesiones`) que utiliza como clave elementos de tipo `SocketAddress` . Las líneas 24 a 27 permiten inscribir nuevas conexiones en el mapa.

La línea 28 construye un nuevo `String` de respuesta y se reenvía al cliente aprovechando el mismo datagrama utilizado para recibir la petición. Esta es una práctica habitual que permite ahorrar tiempo y economizar asignaciones de memoria.

El siguiente paso consiste en añadir un nuevo nivel de complicación aprovechando la serialización de objetos que ya se ha presentado en la Lección 02. Ese es nuestro siguiente desafío.

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.

Usar la serialización de **Java parece un poco complicado** de entender al principio, pero te aseguro que es pan comido en comparación con la alternativa. Al final, usaremos un poco de **código *boilerplate*** que concentraremos en una clase de utilidad, y lo aplicaremos a nuestro Protocolo de Colas de Strings, y empezaremos por el cliente. El servidor se parecerá mucho al ya visto, por eso necesitarás copiar unas cuantas clases que ya has usado en tu proyecto anterior:

- Por parte del paquete "`Mensajes`"
    - `Primitiva` (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.

Clase `UDPHelper`

```
1  package protocol.common;
2
3  public class UDPHelper {
4      public static byte[] aBytes(MensajeProtocolo mensaje) {
5          java.io.ByteArrayOutputStream bos =
6              new java.io.ByteArrayOutputStream();
7
8          try (java.io.ObjectOutput out = new java.io.ObjectOutputStream(bos))
   {
9              out.writeObject(mensaje);
10             return bos.toByteArray();
```

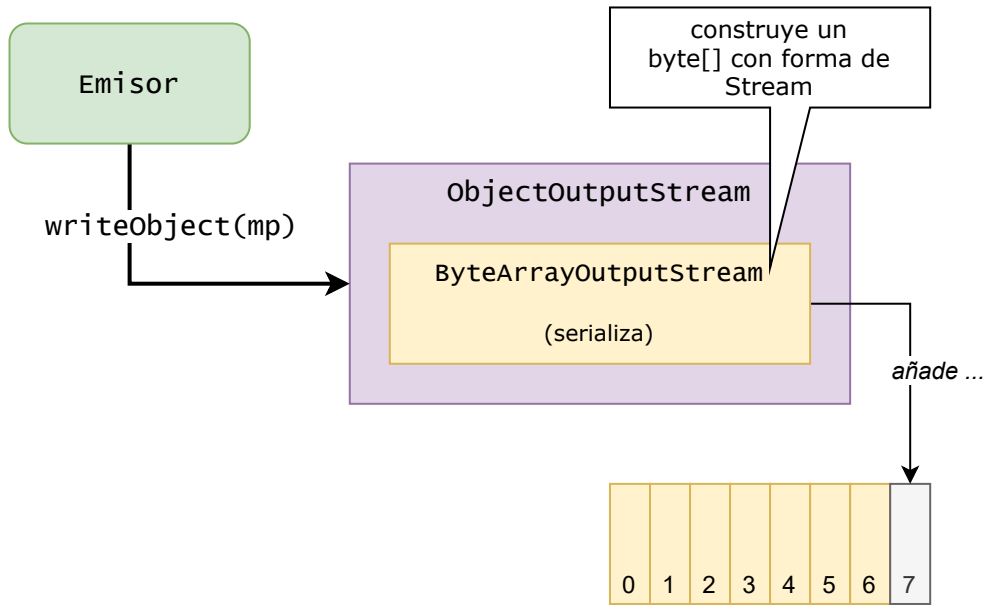
```
        } catch (java.io.IOException ioe) {
            return null;
        }
    }

    public static MensajeProtocolo aMensaje(byte[] entrada) {
        java.io.ByteArrayInputStream bis = new
java.io.ByteArrayInputStream(entrada);

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

```

Estudia atentamente los dos métodos y su signatura. Esta clase está diseñada para concentrar toda la dificultad de la serialización en dos métodos, y así poder empaquetar y desempaquetar de un modo más transparente, como se podrá ver después en el cliente y en el servidor.

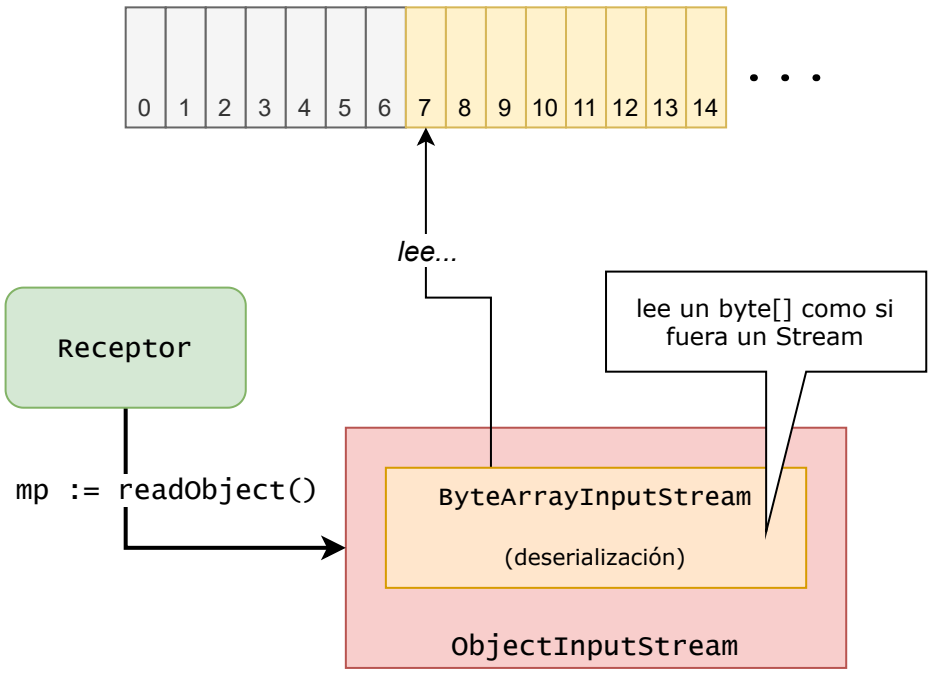
La serialización se realiza apoyándose en dos clases importantes `ByteArrayOutputStream` y `ByteArrayInputStream`. Ambas clases se apilan sobre array de bytes, y nos permiten escribir sobre arrays de bytes y leer de **arrays de bytes como si fueran streams** de salida y de entrada, respectivamente. La única complicación es asumir este comportamiento.

Nuestra clase de utilidad no tiene constructor. Después de repensar mucho la idea los seudoficheros y objetos necesarios se construirán al vuelo (*as needed*):

- En `aBytes()` lo construye la plataforma Java, y



- en `aMensaje()` proviene del `byte[]` que forma parte del datagrama.



Podemos echar en cara a estos métodos que son muy *gastones de recursos* pero te garantizo que **son muy robustos** frente a concurrencia y que el compilador ya se encarga de optimizar un montón de cosas como esta. Observa ahora la clase cliente:

Clase `ClienteProtocolo`

```
1  package protocol.cliente;
2
3  import protocol.common.UDPHelper;
4  import protocol.common.MensajeProtocolo;
5  import protocol.common.Primitiva;
6  import protocol.common.MalMensajeProtocoloException;
7
8  public class ClienteProtocolo {
9    static final private int PUERTO = 1919;
10   static final private int MAXDATAGRAMA = 1024;
11
12   public static void main(String[] args) throws java.io.IOException {
13     String host = "localhost";
14     String linea;
15
16     java.net.DatagramSocket socket = new java.net.DatagramSocket();
17     java.net.InetAddress iaServer = java.net.InetAddress.getByName(host);
18     socket.connect(iaServer, PUERTO);
19
20     try {
21       System.out.println("Pulsa <Enter> para comenzar"); System.in.read();
22
23       prueba(new MensajeProtocolo(Primitiva.HELLO, "Pedro"));
24       prueba(new MensajeProtocolo(Primitiva.PUSH, "010", "abc"));
25       prueba(new MensajeProtocolo(Primitiva.PULL_NOWAIT, "112"));
26       prueba(new MensajeProtocolo(Primitiva.PULL_NOWAIT, "010"));
27       prueba(new MensajeProtocolo(Primitiva.PULL_WAIT, "010"));
28
29       // aquí se espera que otro cliente inserte un mensaje
```

```
30
31        // FIN del escenario 1
32      } catch (java.io.IOException e) {
33        System.err.println("Cliente: Error de apertura o E/S sobre objetos: "
    + e);
34      } catch (Exception e) {
35        System.err.println("Cliente: Excepción. Cerrando Sockets: " + e);
36      } finally {
37        socket.close();
38      }
39    }
40    static void prueba(MensajeProtocolo mp) throws java.io.IOException,
41            MalMensajeProtocoloException, ClassNotFoundException {
42
43      System.out.println("> "+mp);
44      // envía primitiva
45      bytesSalida = UDPHelper.aBytes(mp);
46      java.net.DatagramPacket output =
47         new java.net.DatagramPacket(bytesSalida, bytesSalida.length,
48                     iaServer, PUERTO);
49      socket.send(output);
50
51      // lee primitiva
52      byte[] bytesEntrada = new byte[NMAXDATAGRAMA];
53      input  = new java.net.DatagramPacket(bytesEntrada, bytesEntrada.length);
54      socket.receive(input);
55      System.out.println("< "+(MensajeProtocolo) UDPHelper.aMensaje());
56      System.out.println("Pulsa <Enter>"); System.in.read();
57    }
58  }
```

Este tipo de código de cliente donde se acumulan varias pruebas en un escenario es un poco incómodo. Por esta razón se ha acumulado desvergonzadamente en el método estático `prueba()` toda la tarea de la operación básica del protocolo **petición-respuesta** (líneas 40 a 57). El paso obvio para dar seriedad a este tipo de pruebas sería adoptar una solución similar a la presentada en la Lección 02 basáda en clientes unitarios para cada primitiva del protocolo.

Veamos el aspecto del servidor.

El servidor.

Esta es la primera aproximación al servidor. Cuenta con un sólo hilo y se ha mantenido todo al límite de sencillez, sin sesiones. En la siguiente sección verás cómo podemos seguir complicando este tipo de servidores mediante la técnica de ejecución *a futuros* y sesiones.

Clase `Servidor`

```
1  package protocol.servidor;
2
3  import protocol.common.MensajeProtocolo;
4  import protocol.common.Primitiva;
5  import protocol.common.UDPHelper;
```

```java
import protocol.common.MalMensajeProtocoloException;
import distribuidos.mapqueue.MultiMap;

public class Servidor {
  static final private int PUERTO = 1919;
  static final private int MAXDATAGRAMA = 1024;

  public static void main(String[] args) throws java.io.IOException {
    String mensaje;
    MensajeProtocolo me;
    MensajeProtocolo ms;
    MultiMap<String, String> mapa = new MultiMap<>();

    java.net.DatagramSocket socket;
    java.net.InetAddress iaServer = null;

    try {
      socket = new java.net.DatagramSocket(PUERTO);
    } catch (java.net.SocketException e) {
      System.out.println("Puerto ocupado en el servidor: " + e);
      return;
    }

    // attend protocol primitives.
    while (true) {
      try {
        byte[] bytesIn = new byte[MAXDATAGRAMA];
        java.net.DatagramPacket datagrama =
               new java.net.DatagramPacket(bytesIn, bytesIn.length);
        socket.receive(datagrama);

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

        switch (me.getPrimitiva()) {
        case HELLO:
             ms = new MensajeProtocolo(Primitiva.HELLO, "Hola!");

             datagrama.setData(UDPHandler.aBytes(ms));
             socket.send(datagrama);
             break;
        case PUSH:
             mapa.push(me.getIdCola(), me.getMensaje());
             synchronized (mapa) { // sí, con sus fallos :D
               mapa.notify();
             }
             ms = new MensajeProtocolo(Primitiva.PUSH_OK);

             datagrama.setData(UDPHandler.aBytes(ms));
             socket.send(datagrama);
             break;
        case PULL_NOWAIT:
         if (null ≠ (mensaje = mapa.pop(me.getIdCola()))) {
               ms = new MensajeProtocolo(Primitiva.PULL_OK, mensaje);
             } else {
```

```
61                ms = new MensajeProtocolo(Primitiva.NOTHING);
62              }
63              datagrama.setData(UDPHandler.aBytes(ms));
64              socket.send(datagrama);
65              break;
66          case PULL_WAIT: // a consecuencia de 1 solo hilo ...
67              String idCola = me.getIdCola();
68              Runnable respuesta = () -> {
69                String mensaje_;
70                try {
71                  java.net.SocketAddress sa =
72                      datagrama.getSocketAddress();
73                  synchronized (mapa) { // ya sabemos donde falla.
74                    while (null ==(mensaje_=mapa.pop(idCola))) {
75                      mapa.wait();
76                    }
77                  }
78                  MensajeProtocolo ms_ =
79                       new MensajeProtocolo(Primitiva.PULL_OK, mensaje_);
80                  byte[] out = UDPHandler.aBytes(ms_);
81                  socket.send(
82                    new java.net.DatagramPacket(out, out.length, sa));
83                } catch (MalMensajeProtocoloException mmpe) {
84                  System.out.println(mmpe);
85                } catch (InterruptedException ie) {
86                  System.out.println(ie);
87                } catch (java.io.IOException ioe) {
88                  System.out.println(ioe);
89                }
90              } ; // fin hilo respuesta
91              new Thread(respuesta).start();
92              break;
93          default:
94              ms = new MensajeProtocolo(Primitiva.NOTUNDERSTAND);
95
96              datagrama.setData(UDPHandler.aBytes(ms));
97              socket.send(datagrama);
98              // break;
99            }
100         } catch (java.io.IOException ioe) {
101            System.out.println(ioe);
102         } catch (MalMensajeProtocoloException mmpe) {
103            System.out.println(mmpe);
104         }
105     } // fin del bucle while del switch
106   }
107 }
```

Observa bien, cómo nos asentamos en bloques estándar parecidos a los que emplea el cliente. Un estudio detallado puede llevar un poco de tiempo, y lo realizarás con el profesor in situ. A grandes rasgos, puede verse que se ha optado por un gran bucle que abarca casi todo el método `main()` (líneas 30 a 105) y que es donde se realiza el despacho de cada datagrama que se recibe, independientemente del cliente que sea.

Al comienzo del bucle se construye un datagrama (líneas 32 y 34) que se utilizará tanto para recibir los datos del cliente, como para devolver la respuesta del protocolo. El **switch que comienza en la línea 40 multiplexa la ejecución en función de la primitiva**. En cada una de ellas se calcula una respuesta y se envía llanamente *in-situ*. La única complicación subyace en la primitiva `PULL_WAIT`. Como verás, para seguir atendiendo mensajes, debemos crear un hilo que se bloqueará cuando no haya mensajes en la cola indicada. La solución no es muy elegante, porque otra más evolucionada nos espera.

Servidor de colas de mensajes - En busca de Java RMI

El siguiente ejemplo que se muestra aquí, es una evolución natural del anterior. Una sencilla reflexión sobre el servidor anterior nos plantea un gran problema: **El diseño apesta**. La razón de ello está en que ***el código de servicio propio del problema, se mezcla con la infraestructura de comunicaciones!!***. Lo ideal sería que el código del servicio estuviera aislado, y algún mecanismo de programación nos permitiera **inyectar esta dependencia** en el soporte del servicio. Así es como está planteado en Java RMI, y quiero que veas el siguiente código de servidor, mientras el profesor te ilustra las secciones más importantes:

```
1  package protocol.servidor;
2
3  import java.io.IOException;
4  import java.util.EnumMap;
5  import java.util.Map;
6  import java.util.HashMap;
7
8  import distribuidos.mapqueue.MultiMap;
9  import protocol.common.MalMensajeProtocoloException;
10 import protocol.common.MensajeProtocolo;
11 import protocol.common.Primitiva;
12 import protocol.common.UDPHelper;
13
14 import java.net.DatagramPacket;
15 import java.net.DatagramSocket;
16 import java.net.SocketAddress;
17 import java.net.SocketException;
18
19 public class Servidor implements Runnable {
20   public final int PUERTO ;
21   public final int MAXDATAGRAMA ;
22   final Map<Primitiva,MetodoServicio> metodos; // para desacoplar
23   public java.net.DatagramSocket socket;
24   private MultiMap<String, String> mapa ;      // colas de mensajes
25
26   // Inicializaciones del mapa de métodos y del socket de recepcion
27   public Servidor(int puerto, int tallaDatagrama) {
28       PUERTO       = puerto;
29       MAXDATAGRAMA = tallaDatagrama;
30       metodos = new EnumMap<>(Primitiva.class); // especial para Enum
31       mapa = new MultiMap<>();
```

```
32
33          try {
34              socket = new DatagramSocket(PUERTO);
35          } catch (SocketException e) {
36              System.out.println("Puerto ocupado en el servidor: " + e);
37          }
38      }
39
40      // SUPER-SLIM Main() - Servidor y Lanzador a la vez.
41      public static void main(String [] args) {
42        Servidor server = new Servidor(1099, 1024);
43        server.init();
44        server.start();
45      }
46
47      public void setMetodo(Primitiva p, MetodoServicio m) {
48        metodos.put(p, m);
49      }
50
51      // MAPA Primitiva-Método (aquí colgamos los handlers)
52      public void init() {
53        this.setMetodo(Primitiva.HELLO,         this::hello);
54        this.setMetodo(Primitiva.PUSH,          this::push);
55        this.setMetodo(Primitiva.PULL_NOWAIT,   this::pull_nowait);
56        this.setMetodo(Primitiva.PULL_WAIT,     this::pull_wait);
57        this.setMetodo(Primitiva.PULL_PROMISE,  this::pull_promise);
58        this.setMetodo(Primitiva.NOTUNDERSTAND, this::notunderstand);
59      }
60
61      public void start() {
62          new Thread(this, "Sirviente").start();
63      }
64
65      // output sólo envía mensajes con contenido, no m == null
66      public void output(MensajeProtocolo m, SocketAddress sa)
67         throws IOException {
68          System.out.println(">>>>>> "+m); // DEBUG OUT
69          if (null ≠ m) {
70              byte[] out = UDPHelper.aBytes(m);        // finalmente esta.
71              socket.send(new DatagramPacket(out, out.length, sa));
72          } else {
73              System.err.println("output nulo para: "+sa);
74          }
75      }
76      // recuerda que el servidor es Runnable. Se ejecuta en un hilo a parte
77      public void run() {
78          MensajeProtocolo me;
79          MensajeProtocolo ms;
80          HashMap<SocketAddress, Sesion> sesiones = new HashMap<>();
81          byte[] bytesIn = new byte[MAXDATAGRAMA];
82
83          while (true) { // Nuestro amigo "bucle indefinido de servicio"
84              try {
85                  DatagramPacket datagrama =
86                          new DatagramPacket(bytesIn, bytesIn.length);
```

```
87                socket.receive(datagrama);
88
89                Sesion sesion;
90                java.net.SocketAddress sa = datagrama.getSocketAddress();
91
92                if ((sesion = sesiones.get(sa)) == null) {
93                    sesion = new Sesion(sa);
94                    sesiones.put(sa, sesion);
95                }
96
97                me = UDPHelper.aMensaje(bytesIn);
98                System.out.println("<<<<<< "+me); // DEBUG OUT
99                // En esta línea se busca el método según la Primitiva
recibida
100               ms = metodos.get(me.getPrimitiva()).metodo_servicio(sesion,
101                                   (MensajeProtocolo) me);
102               if (null != ms) // el servidor puede no devolver nada ahora,
as PULL_WAIT
103                   output(ms, sesion.sa);
104           } catch (IOException ioe) {
105               ioe.printStackTrace(System.err);
106           } catch (Exception e) {
107               e.printStackTrace(System.err);
108           }
109       }
110   }
111
112   // Observa cómo hemos extraído los handlers del interior de la
infraestructura.
113   MensajeProtocolo hello(Sesion ses, MensajeProtocolo me) {
114     ses.setNombre(me.getMensaje());
115     try {
116       return new MensajeProtocolo(Primitiva.HELLO, "Hola, soy UDPServer
v1.0!");
117     } catch (MalMensajeProtocoloException ignore) { return null; }
118   }
119
120   MensajeProtocolo push(Sesion ses, MensajeProtocolo me) {
121     mapa.push(me.getIdCola(), me.getMensaje());
122     synchronized (mapa) { // sí, con sus fallos :D
123       mapa.notify();
124     }
125     try {
126       return new MensajeProtocolo(Primitiva.PUSH_OK);
127     } catch (MalMensajeProtocoloException ignore) { return null; }
128   }
129
130   MensajeProtocolo pull_nowait(Sesion ses, MensajeProtocolo me) {
131     String mensaje;
132     MensajeProtocolo ms;
133     try {
134       if (null != (mensaje = mapa.pop(me.getIdCola())))
135          ms = new MensajeProtocolo(Primitiva.PULL_OK, mensaje);
136       else
137          ms = new MensajeProtocolo(Primitiva.NOTHING);
```

```
138     } catch (MalMensajeProtocoloException ignore) { ms = null; }
139     return ms;
140   }
141
142   MensajeProtocolo pull_wait(Sesion sesion, MensajeProtocolo me) {
143     String mensaje;
144     MensajeProtocolo ms;
145     try {
146       if (null =≠ (mensaje = mapa.pop(me.getIdCola()))) {
147          ms = new MensajeProtocolo(Primitiva.PULL_OK, mensaje);
148       } else {
149          // prepara un hilo para atender al cliente que espera un token
150          new Thread(() -→ {
151                String mensaje_;
152                MensajeProtocolo ms_;
153                try {
154                   synchronized (mapa) { // ya sabemos donde falla.
155                     while (null == (mensaje_ = mapa.pop(me.getIdCola())))
    {
156                       mapa.wait();
157                     }
158                   }
159                   try {
160                        ms_ = new MensajeProtocolo(Primitiva.PULL_OK,
    mensaje_);
161                   } catch (MalMensajeProtocoloException ignore) {
162                        ms_ = null;
163                   }
164                   output(ms_, sesion.sa);
165                } catch (InterruptedException ie) {
166                     System.out.println(ie);
167                } catch (java.io.IOException ioe) {
168                     System.out.println(ioe);
169                }
170             }, "Respuesta PullWait" ).start();  // fin Thread
171          // devuelve null para que output ignore el envío
172          ms = null;
173       }  // fin del if si hay mensaje en la cola
174     } catch (MalMensajeProtocoloException ignore) { ms = null; }
175     return ms;
176   }
177
178   MensajeProtocolo pull_promise(Sesion sesion, MensajeProtocolo me) {
179     String mensaje;
180     MensajeProtocolo ms;
181     try {
182       if (null =≠ (mensaje = mapa.pop(me.getIdCola()))) {
183          ms = new MensajeProtocolo(Primitiva.PULL_OK, mensaje);
184       } else {
185         // prepara un hilo para atender al cliente que espera un token
186         Runnable respuesta = () -→ {
187                String mensaje_;
188                try {
189                   MensajeProtocolo ms_;
190                   synchronized (mapa) {
```

```
                    while (null == (mensaje_ = mapa.pop(me.getIdCola()))) {
                        mapa.wait();
                    }
                }
                try {
                    ms_ = new MensajeProtocolo(Primitiva.PULL_OK, mensaje_);
                } catch (MalMensajeProtocoloException ignore) { ms_ = null; }
                // no vuelve con el socket de la sesión sino con la promesa
                output(ms_, me.getSocketPromise());
              } catch (InterruptedException ie) {
                System.out.println(ie);
              } catch (java.io.IOException ioe) {
                System.out.println(ioe);
              }
            }; // fin hilo respuesta
          new Thread(respuesta).start();
          // aquí devolvemos la promesa al cliente!
          try {
            return new MensajeProtocolo(Primitiva.PROMISED);
          } catch (MalMensajeProtocoloException ignore) { return null; }
        } // fin del if
      } catch (MalMensajeProtocoloException ignore) { ms = null; }
      return ms;
    }

    MensajeProtocolo notunderstand(Sesion sesion, MensajeProtocolo me) {
      try {
        return new MensajeProtocolo(Primitiva.NOTUNDERSTAND);
      } catch (MalMensajeProtocoloException ignore) { return null; }
    }
}
```

La primitiva `PULL_PROMISE`

Se ha aprovechado la ocasión para dar un paso más en pos de lo excelente... Se ha añadido una nueva primitiva. Su comportamiento es similar a `PULL_WAIT`, y de hecho el método de servicio `pull_promise(...)` es casi igual a `pull_wait(...)`:

- ambos lanzan un hilo a la espera que la cola de mensajes concreta tenga un `String` en el futuro,
- y se bloquean en él.

`PULL_PROMISE` sin embargo, vuelve inmediatamente si no hay datos, y devuelve `PROMISE`, de modo análogo a `PULL_NOWAIT` retornando `NOTHING` en el mismo caso. La diferencia está en que ***la promesa* devuelve un dato en cuanto es posible** (lo cual lo extrae de la cola) y el cliente lo recibirá en el futuro. Para se ha creado una entrada nueva en `Primitiva` y a continuación te enseño un cliente unitario que aprovecha esta primitiva para que veas cómo se atiende:

```
package protocol.cliente;

```

```java
import protocol.common.MensajeProtocolo;
import protocol.common.Primitiva;
import protocol.common.UDPHelper;
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.SocketAddress;

public class PullPromise {
  static final private int PUERTO = 1099;
  static final private int MAXDATAGRAMA = 1024;
  static DatagramSocket socket = null;

  public static void main(String[] args) throws java.io.IOException {
    if (args.length =/= 2) {
      System.out.println("Use:\njava protocol.cliente.PullPromise host clave");
      System.exit(-1);
    }
    String host  = args[0]; // loocalhost o ip/dn del servidor
    String clave = args[1]; // clave del deposito

    // para ilustrar un socket conectado
    socket = new DatagramSocket();
    socket.connect(java.net.InetAddress.getByName(host), PUERTO);
    SocketAddress saServer = socket.getRemoteSocketAddress();

    try {
      MensajeProtocolo ms = new MensajeProtocolo(Primitiva.PULL_PROMISE,
              clave, socket.getLocalSocketAddress());
      System.out.println(">>: "+ms);
      output(ms, saServer);
      // Espera mensaje PULL_OK o muerte, :P
      MensajeProtocolo me = input();
      System.out.println("<<: "+me);
      switch (me.getPrimitiva()) {
      case PROMISED :
          // Lanzamos dos hilos, uno que escucha y otro que pone puntitos
          boolean[] parada = {true}; // variable compartida entre ellos
          // Hilo que espera la promesa hecha por el servidor.
          new Thread(() -> {
              try {
                MensajeProtocolo m = input();
                System.out.println("Esto es lo que me han dado: "+m);
                parada[0] = false;
              } catch (IOException ignore) { }
             }, "Esperando...").start();

          // Hilo que simula hacer algo poniendo puntitos en pantalla
          System.out.println("Me han _prometido_ un dato\n");
          System.out.println("Voy a imprimir un rato mientras me lo pasan: ");
          try {
            while (true == parada[0]) {
              Thread.sleep(300); //dormir 300 millis
```

```
56                System.out.print('.');
57              }
58            } catch (InterruptedException ie) {
59                System.out.println("Adios!");
60            }
61            break;
62        case PULL_OK  : // tenemos suerte, había un dato.
63            System.out.println("Me han devuelto un dato");
64            break;
65        default :
66            System.out.println("???");
67        }
68      } catch (java.io.IOException e) {
69        System.err.println("Cliente: Error de apertura o E/S sobre objetos: "
    + e);
70      } catch (Exception e) {
71        System.err.println("Cliente: Excepción. Cerrando Sockets: " + e);
72      } finally {
73        socket.close();
74      }
75    }
76
77    // usamos socket externamente para no complicar.
78    // output sólo envía mensajes con contenido, no m == null
79    public static void output(MensajeProtocolo m, SocketAddress sa)
80            throws IOException {
81      if (null ≠ m) {
82        byte[] out = UDPHelper.aBytes(m);        // finalmente esta.
83        socket.send(new DatagramPacket(out, out.length, sa));
84      } else {
85        System.err.println("output nulo para: "+sa);
86      }
87    }
88
89    // usamos socket externamente para no complicar.
90    public static MensajeProtocolo input() throws java.io.IOException {
91      byte[] bytesIn = new byte[MAXDATAGRAMA];
92      DatagramPacket datagrama = new DatagramPacket(bytesIn, bytesIn.length);
93      socket.receive(datagrama);
94      return UDPHelper.aMensaje(bytesIn);
95    }
96  }
```

Cualquier cliente que use `PULL_PROMISE` tiene la responsabilidad de escuchar en un socket, la respuesta del servidor. Si no lo hace, se perderá el dato. Si observas la línea 30 del cliente, veras que la primitiva tiene un parámetro adicional donde le indica al servidor dónde espera la respuesta de `PULL_PROMISE` en caso de que no haya un dato en la cola elegida. A continuación, creamos dos hilos, aunque podría haberse hecho más sencillo:

- un hilo que espera la respuesta prometida, y
- otro hilo que simula un trabajo asíncrono con la espera.

Para terminar este último se ha utilizado una variable *mutable* bajo la forma de un array de un elemento. La razón de ello está en que cada vez que se emplea una lambda-definición, se congelan todas las variables del contexto y necesitamos algo parecido a un apuntador.

Epílogo: Multienhebrado `callable()` en servidores monoenhebrados.

Por último, una pequeña rareza que podría omitirse si no fuera por su trascendencia y de la cual has recibido una pincelada en la sección anterior. 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. Para una introducción muy interesante sobre el concepto de "futuros" puedes consultar [esta página de la wikipedia] y [este documento práctico en Java].

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

Clase `ServerCallable`

```
1  package udp.callable;
2
3  public class ServerCallable {
4    static final int BUFSIZ = 256;
5    static final java.util.HashMap<java.net.SocketAddress,
6        SessionCallableEcho> sesiones = new java.util.HashMap<>();
7
8    public static void main(String[] args) {
9      java.util.concurrent.ExecutorService pool =
10         java.util.concurrent.Executors.newFixedThreadPool(3);
11     java.net.DatagramSocket sock;
12     try {
13       sock = new java.net.DatagramSocket(1919);
14     } catch (java.net.SocketException e) {
15       System.out.println(e);
16       return;
17     }
18     // echo back everything
19     while (true) {
20       try {
21         java.net.DatagramPacket pack =
22            new java.net.DatagramPacket(new byte[BUFSIZ],BUFSIZ);
23
24         sock.receive(pack);
25         SessionCallableEcho s;
26
27         if ((s = sesiones.get(pack.getSocketAddress())) == null) {
28             s = new SessionCallableEcho(pack.getSocketAddress(), sock);
29             sesiones.put(pack.getSocketAddress(), s);
```

```
30          }
31          s.putDP(pack);
32          java.util.concurrent.Future<Boolean> tarea = pool.submit(s);
33
34        } catch (java.io.IOException ioe) {
35          System.out.println(ioe);
36        }
37      }
38    }
39  }
```

En lugar de utilizar una declaración lambda, he preferido separar en otro código el sirviente de este tipo de servidor, para añadir claridad. En la línea 32 verás una gran diferencia en la forma de invocar el código que antes aparecía bajo la forma de objetos de tipo `Runnable`. Ahora son objetos `Callable` y el código de la tarea se va a concentrar en el método `call()`. A priori no hay mucha diferencia con lo anterior, pues ambos van a lanzar tareas de un *pool* de hilos a las órdenes de un `ExecutorService`.

Habrá una cola de tareas que se irán atendiendo y de cuya *finalización* dependerá qué deberá hacer el cliente en la siguiente interacción. El objeto `Future` , `tarea` dispone de los métodos:

- `cancel()` para eliminar la tarea pendiente.
- `get()` para sincronizarse con la finalización de la tarea, obteniendo el valor devuelto de `Callable`.
- `isDone()` e `isCancelled()` permiten ***sondear*** y cancelar la tarea, para evitar el bloqueo de `get() y aquí está precisamente el interés.

El interés de esta construcción pasa por el interés que tengamos en el servicio por comprobar las condiciones de finalización de los hilos pendientes. Este mecanismo nos permite monitorizar y evitar situaciones de bloqueo en los servidores. En definitiva, la ejecución de tareas *a futuro* establece un *protocolo* de comunicación entre el lanzador de cada tarea y la tarea una vez que finaliza (*terminación de la promesa*).

El contrato del sirviente con `Callable<T>` consiste en implementar el método `T call()`. Esta es la forma en que se puede enviar información al proceso que realiza `get()` sobre la tarea realizada o incluso el objeto resultado del trabajo. Nuestro hilo supervisor podrá sondear la finalización con `isDone()` atendiendo a varias tareas por turno rotatorio.

Clase `SessionCallableEcho`

```
1  package udp.callable;
2
3  public class SessionCallableEcho
4                 implements java.util.concurrent.Callable<Boolean> {
5      final java.net.SocketAddress sa;
6      static private int idCounter = 0;
7      private final int id;
8      private java.net.DatagramPacket dp;
9      private java.net.DatagramSocket ds;
10
11     SessionCallableEcho(java.net.SocketAddress sa,
12                     java.net.DatagramSocket ds) {
```

```
13          this.sa = sa;
14          this.ds = ds;
15          this.id = ++idCounter;
16      }
17
18      @Override
19      public Boolean call() {
20          String texto = new String(dp.getData());
21          if (!texto.startsWith("Adios.") ) {
22              System.out.println("<"+id+">["+sa+"]: "+texto);
23              try { this.ds.send(dp); }
24              catch (java.io.IOException ioe) {
25                  System.err.println("SessionCallableEcho: IOE: "+ioe);
26              }
27              return Boolean.TRUE;
28          } else {
29              return Boolean.FALSE;
30          }
31      }
32      public int getId() {
33          return id;
34      }
35      public void putDP(java.net.DatagramPacket dp) {
36          this.dp = dp;
37      }
38  }
```

Los clientes de este tipo servidores no tienen por qué cambiar, aunque en nuestro caso podemos utilizar este mecanismo para la retransmisión del resultado en forma de *Promesa* para lo cual el cliente tiene que estar preparado.

Conclusión.

Este es el auténtico fin. Faltan muchas cosas por decir y por programar. Pero todas llegan al mismo sitio:

```
Frameworks y contenedores de apliaciones distribuidas (J2EE o Spring) Java.
```

La manera más natural es pasar por Java RMI como primer paso, después, ver algo de CORBA y llegar a servicios REST. Los servicios de mensajería son harina de otro costal. Todas estas alternativas y pasos son muy, pero que muy divertidas, pero también son un farragosas. Si quieres saber más pulsa `<ENTER>`.

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.