UNIVERSIDADE DA CORUÑA
Departamento de Tecnoloxías da Información
e as Comunicacións

LABORATORIO DE RC:
TUTORIAL DE SOCKETS EN JAVA

Índice

1. Presentación

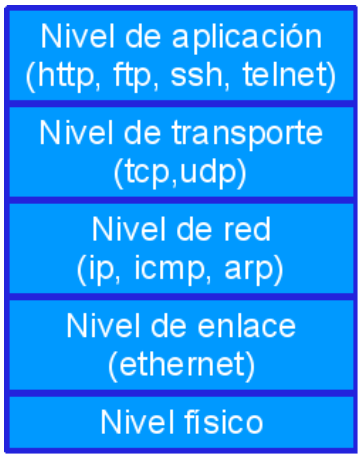
El laboratorio de Redes de Comunicaciones constará de un conjunto de prácticas **no obligatorias**. Para la elaboración de las prácticas se dispondrá del laboratorio 1.1 y todas ellas han de ser realizadas **individualmente**. La evaluación de la primera práctica consistirá en una defensa ante el profesor de prácticas y para las dos últimas se realizarán dos pequeños exámenes en el propio laboratorio.

Este tutorial proporciona una introducción a los sockets en Java. La parte práctica de dicho tutorial **NO** será evaluada, si bien es muy recomendable la realización de la misma.

Cada práctica constará de un enunciado en donde se plantearán las tareas a realizar, y podrá incluir información complementaria para el desarrollo de la misma. Las prácticas que requieran el empleo de un lenguaje de programación deberán ser realizadas en **Java**.

2. Conceptos básicos

En clase de teoría se ha mencionado en repetidas ocasiones la pila de protocolos TCP/IP, que permite la transmisión de datos entre redes de computadores. Dicha pila consta de una serie de capas tal y como se puede apreciar en el diagrama siguiente:



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

2.1. TCP

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

2.2. UDP

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

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 dichos 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 dicha máquina. Si a través de una de dichas 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 cómo 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.

3.1. Sockets en Java

Java incluye la librería `java.net` para la utilización de sockets, tanto TCP como UDP. Este tutorial se basa exclusivamente en clases y métodos de esta librería, por lo que será necesario importarla en todos los ejemplos y prácticas.

4. Sockets UDP

Los sockets UDP son no orientados a conexión. Los clientes no se conectarán con el servidor sino que cada comunicación será independiente, sin poderse garantizar la recepción de los paquetes ni el orden de los mismos. Es en este momento en donde podemos definir el concepto de **datagrama**, que no es más que un mensaje independiente, enviado a través de una red cuya llegada, tiempo de llegada y contenido no están garantizados.

En primer lugar, vamos a 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 máquina 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.

A continuación se muestra el código fuente del cliente de eco UDP.

```java
import java.net.*;

/** Ejemplo que implementa un cliente de eco usando UDP. */

public class ClienteUDP {

  public static void main(String argv[]) {
    if (argv.length != 3) {
      System.err.println("Formato: ClienteUDP <maquina> <puerto> <mensaje>");
      System.exit(-1);
    }

    DatagramSocket sDatagram = null;

    try {

      // Creamos el socket no orientado a conexión
      // (en cualquier puerto libre)
      sDatagram = new DatagramSocket();

      // Establecemos un timeout de 30 segs
      sDatagram.setSoTimeout(30000);

      // Obtenemos la dirección IP del servidor
      // (recibida en el primer argumento por linea de comandos)
      InetAddress dirServidor = InetAddress.getByName(argv[0]);
      // Obtenemos el puerto del servidor
      // (recibido en el segundo argumento por linea de comandos)
      int puertoServidor = Integer.parseInt(argv[1]);
      // Obtenemos el mensaje
      // (tercer argumento de la linea de comandos)
      String mensaje = argv[2];

      // Preparamos el datagrama que vamos a enviar y lo enviamos
      DatagramPacket dgramEnv = new DatagramPacket(mensaje.getBytes(),
              mensaje.getBytes().length, dirServidor, puertoServidor);
      // Enviamos el datagrama
      sDatagram.send(dgramEnv);
      System.out.println("CLIENTE: Enviando "
              + new String(dgramEnv.getData()) + " a "
              + dgramEnv.getAddress().toString() + ":"
              + dgramEnv.getPort());

      // Preparamos el datagrama de recepción
      byte array[] = new byte[1024];
      DatagramPacket dgramRec = new DatagramPacket(array, array.length);
      // Recibimos el mensaje
      sDatagram.receive(dgramRec);
      System.out.println("CLIENTE: Recibido "
              + new String(dgramRec.getData(), 0, dgramRec.getLength())
              + " de " + dgramRec.getAddress().toString() + ":"
              + dgramRec.getPort());


    } catch (SocketTimeoutException e) {
      System.err.println("30 segs sin recibir nada");
    } catch (Exception e) {
      System.err.println("Error: " + e.getMessage());
    } finally {
      // Cerramos el socket para liberar la conexión
      sDatagram.close();
    }
  }
}
```

De manera más detallada, estos son los pasos que se ejecutan en el cliente:

1. Se crea un socket no orientado a conexión. Es importante recalcar que no es necesario especificar un número de puerto en el constructor, ya que el propio constructor se encargará de seleccionar un puerto libre (puerto efímero). Para más información ver la documentación de la clase `DatagramSocket` (en especial la sección de constructores).
2. Establecemos un tiempo de espera máximo para el socket. Si pasado ese tiempo no ha recibido nada se lanzará la excepción correspondiente.
3. Se obtiene la dirección IP de la máquina en la que se encuentra el servidor, a partir del primer argumento recibido por línea de comandos. La clase `InetAddress` representa en Java el concepto de dirección IP. Esta clase dispone del método estático `getByName()` que obtiene la dirección IP a partir del `String` que recibe como parámetro. Este parámetro puede ser un nombre o una dirección IP.
4. Se obtiene el número de puerto en el que está ejecutándose el servidor, a partir del segundo argumento recibido por línea de comandos.
5. Se obtiene el mensaje que queremos enviar al servidor, a partir del tercer argumento recibido por línea de comandos. En caso de querer enviar varias palabras es necesario utilizar comillas dobles (p.e. “Esto es una prueba”).
6. Preparamos el datagrama que vamos a enviar, indicando: el mensaje que queremos enviar (como un array de bytes), el número de bytes a enviar, la dirección IP del destinatario y el puerto del destinatario. Ver la clase `DatagramPacket`.
7. Enviamos el datagrama invocando el método `send()` del socket que hemos creado inicialmente.
8. Preparamos un nuevo datagrama para recibir la respuesta del servidor. Para ello, es necesario crear previamente un array de bytes que va a almacenar la respuesta que vayamos a recibir. Al crear el nuevo datagrama, se indicará el array en donde queremos almacenar la respuesta y el número máximo de bytes que puede almacenar este array.
9. Recibimos el datagrama, utilizando el método `receive()` de nuestro socket UDP. Este método es bloqueante, es decir, el programa se quedará esperando en este método hasta recibir algo o, como hemos establecido un timeout, hasta que venza el timeout.
10. Por último, cerramos el socket.

4.1. Ejercicio 1: Implementación del servidor de eco UDP

En base al cliente de eco UDP, implementa un servidor de eco UDP.

El servidor deberá seguir los siguientes pasos:

1 Crear un `DatagramSocket`, pero asociado a un número de puerto específico.

2 Establecer un tiempo de espera máximo para el socket.

3 Crear un bucle infinito:

3.1 Preparar un datagrama para recibir mensajes de los clientes. Es

recomendable crear un nuevo objeto `DatagramPacket` para cada mensaje que se vaya a recibir.

3.2 Recibir un mensaje.

3.3 Preparar el datagrama de respuesta para enviar al cliente. Recuerda que en cada datagrama recibido queda registrado la dirección IP y el número de puerto del remitente.

3.4 Enviar el mensaje de respuesta.

A continuación se muestra el pseudo-código del servidor:

```
import java.net.*;

/** Ejemplo que implementa un servidor de eco usando UDP. */

public class ServidorUDP {

  public static void main(String argv[]) {
    if (argv.length != 1) {
      System.err.println("Formato: ServidorUDP <puerto>");
      System.exit(-1);
    }

    try {
      // Creamos el socket del servidor
      // Establecemos un timeout de 30 segs
       while (true) {
        // Preparamos un datagrama para recepción
        // Recibimos el mensaje
        // Preparamos el datagrama que vamos a enviar
        // Enviamos el mensaje
      }
    } catch (SocketTimeoutException e) {
      System.err.println("30 segs sin recibir nada");
    } catch (Exception e) {
      System.err.println("Error: " + e.getMessage());
      e.printStackTrace();
    } finally{
       // Cerramos el socket
    }
  }
}
```

Una vez finalizado el servidor de eco UDP, para comprobar que funciona correctamente deberás ejecutar en un terminal el servidor:

`java ServidorUDP 5000`

El servidor se quedará esperando a que los clientes le envíen peticiones.

En otro terminal, deberás ejecutar el cliente UDP:

`java ClienteUDP localhost 5000 “Probando mi servidor UDP”`

El resultado debería ser que el servidor recibiese el mensaje del cliente y se lo reenviase, mientras que el cliente debería recibir la respuesta del servidor.

4.2. Preguntas UDP

Una vez que hayas implementado el servidor UDP, responde a las siguientes preguntas:

1. ¿Qué sucede si reservamos un array de menor tamaño (p.e. 5 bytes) que el tamaño del mensaje que estamos recibiendo?
2. ¿Qué sucede si queremos lanzar dos servidores UDP simultáneamente en el mismo número de puerto? ¿Por qué?
3. ¿Qué sucede si queremos lanzar varios clientes para que se comuniquen con el mismo servidor?¿Existe algún tipo de bloqueo en los clientes que impida la comunicación?
4. ¿Qué sucede si no está vacío el buffer de recepción al llegar al método necesario para recibir un `DatagramPacket`?

5. Sockets TCP

Los sockets TCP son orientados a conexión y fiables. Esto implica que antes de poder enviar y recibir datos es necesario establecer una conexión entre el cliente y el servidor. Una vez que la conexión está establecida, el protocolo TCP garantiza que los datos enviados son recibidos correctamente y debidamente ordenados en el otro extremo.

En primer lugar, vamos a implementar un cliente de eco TCP. El formato de ejecución será:

```
java ClienteTCP <máquina_servidor> <puerto_servidor> <mensaje>
```

En donde,

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

A continuación se muestra el código fuente del cliente de eco TCP.

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

/** Ejemplo que implementa un cliente de eco usando TCP. */

public class ClienteTCP {

  public static void main(String argv[]) {
    if (argv.length != 3) {
      System.err.println("Formato: ClienteTCP <maquina> <puerto> <mensaje>");
      System.exit(-1);
    }

    Socket socket = null;
    try {

      // Obtenemos la dirección IP del servidor
      InetAddress dirServidor = InetAddress.getByName(argv[0]);
      // Obtenemos el puerto del servidor
      int puertoServidor = Integer.parseInt(argv[1]);
      // Obtenemos el mensaje
      String mensaje = argv[2];

      // Creamos el socket y establecemos la conexión con el servidor
      socket = new Socket(dirServidor, puertoServidor);

      // Establecemos un timeout de 30 segs
      socket.setSoTimeout(30000);

      System.out.println("CLIENTE: Conexion establecida con "
              + dirServidor.toString() + " al puerto " + puertoServidor);
      // Establecemos el canal de entrada
      BufferedReader sEntrada = new BufferedReader(new InputStreamReader(
              socket.getInputStream()));
      // Establecemos el canal de salida
      PrintWriter sSalida = new PrintWriter(socket.getOutputStream(), true);

      System.out.println("CLIENTE: Enviando " + mensaje);
      // Enviamos el mensaje al servidor
      sSalida.println(mensaje);

      // Recibimos la respuesta del servidor
      String recibido = sEntrada.readLine();

      System.out.println("CLIENTE: Recibido " + recibido);
      // Cerramos los flujos y el socket para liberar la conexión
      sSalida.close();
      sEntrada.close();

    } catch (SocketTimeoutException e) {
      System.err.println("30 segs sin recibir nada");
    } catch (Exception e) {
      System.err.println("Error: " + e.getMessage());
    } finally {
      try {
        socket.close();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }

  }
}
```

De manera más detallada, estos son los pasos que se ejecutan en el cliente:

1. Se obtiene la dirección IP de la máquina en la que se encuentra el servidor, a partir del primer argumento recibido por línea de comandos.
2. Se obtiene el número de puerto en el que está ejecutándose el servidor, a partir del segundo argumento recibido por línea de comandos.
3. Se obtiene el mensaje que queremos enviar al servidor, a partir del tercer argumento recibido por línea de comandos.
4. Se crea un socket orientado a conexión. No se especifica ningún puerto para el cliente, ya que automáticamente se seleccionará un puerto libre (efímero). En el constructor se especifica la dirección IP y el puerto del servidor, estableciéndose inmediatamente después de la creación del socket, la conexión con el servidor. Para más información ver la documentación de la clase `Socket`, en especial la sección de constructores.
5. Establecemos un tiempo de espera máximo para el socket. Si pasado ese tiempo no ha recibido nada se lanzará la excepción correspondiente.
6. Creamos el canal de entrada para recibir los datos del servidor. Para leer los datos del servidor se utiliza la clase `BufferedReader`, construida a partir del método `getInputStream()` del socket cliente. Esta clase dispone del método `readLine()` que permite la lectura línea a línea.
7. Creamos el canal de salida para enviar datos al servidor. Se utiliza la clase `PrintWriter`, construida a partir del método `getOutputStream()` del socket cliente. Esta clase dispone de los métodos `println()` y `print()`, equivalentes a los utilizados habitualmente para imprimir por pantalla.
8. Enviamos el mensaje al servidor invocando el método `println()` del flujo de salida.
9. Esperamos y recibimos la respuesta del servidor invocando el método `readLine()` del flujo de entrada.
10. Por último, cerramos el socket.

5.1. Ejercicio 2: Implementación del servidor de eco TCP

En base al cliente de eco TCP, implementa un servidor **multihilo** de eco TCP. Habitualmente, los servidores TCP son multihilo para poder procesar múltiples conexiones simultáneamente. Para facilitar su implementación se recomienda realizar una primera versión del servidor monohilo, y comprobar su funcionamiento con el cliente de eco TCP. Una vez la versión monohilo funcione correctamente, se puede realizar la versión multihilo.

El servidor monohilo deberá seguir los siguientes pasos:

1 Crear un `ServerSocket`, asociado a un número de puerto específico.

2 Establecer un tiempo de espera máximo para el socket.

3 Crear un bucle infinito:

3.1 Invocar el método `accept()` del socket servidor. Este método se queda esperando hasta recibir la petición de conexión de un cliente. En cuanto se establece la conexión con el cliente, devuelve un nuevo socket que se utilizará para la comunicación con ese cliente.

3.2 Preparar los flujos de entrada y salida, a partir del nuevo socket.

3.3 Recibir el mensaje del servidor.

3.4 Enviar el mensaje de eco de vuelta al cliente.

3.5 Cerrar los flujos y la conexión del socket creado en el método `accept()`.

A continuación se muestra el pseudo-código del servidor monohilo:

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

/** Ejemplo que implementa un servidor de eco usando TCP. */

public class ServidorTCP {

  public static void main(String argv[]) {
    if (argv.length != 1) {
      System.err.println("Formato: ServidorTCP <puerto>");
      System.exit(-1);
    }

    try {
      // Creamos el socket del servidor
      // Establecemos un timeout de 30 segs
      while (true) {
        // Esperamos posibles conexiones
        // Establecemos el canal de entrada
        // Establecemos el canal de salida
        // Recibimos el mensaje del cliente
        // Enviamos el eco al cliente
        // Cerramos los flujos
      }
    } catch (SocketTimeoutException e) {
      System.err.println("30 segs sin recibir nada");
    } catch (Exception e) {
      System.err.println("Error: " + e.getMessage());
      e.printStackTrace();
   } finally {
       //Cerramos el socket
    }
  }
}
```

Una vez finalizado el servidor monohilo de eco TCP, para comprobar que funciona correctamente deberás ejecutar en un terminal el servidor:

`java ServidorTCP 5000`

El servidor se quedará esperando a que los clientes le envíen peticiones.

En otro terminal, deberás ejecutar el cliente TCP:

`java ClienteTCP localhost 5000 “Probando mi servidor TCP”`

El resultado debería ser que el servidor recibiese el mensaje del cliente y se lo reenviase, mientras que el cliente debería recibir la respuesta del servidor.

Para convertir el servidor monohilo en multihilo es necesario crear una clase nueva (`ThreadServidor`) que extienda la clase `Thread`. Esta clase será la que se encargue de atender una conexión con un cliente. Ahora, los pasos en el servidor TCP serían los siguientes:

1. Crear un `ServerSocket`, asociado a un número de puerto específico.
2. Establecer un tiempo de espera máximo para el socket.
3. Crear un bucle infinito:
   3.1 Invocar el método `accept()` del socket servidor. Al establecerse la conexión, devuelve un nuevo socket que se utilizará para la comunicación con ese cliente.
   3.2 Crear un nuevo objeto `ThreadServidor`, pasando como parámetro el nuevo socket de la conexión. De esta manera, la conexión es procesada con este nuevo socket en un hilo de ejecución independiente, quedando el socket servidor preparado para recibir nuevas peticiones de otros clientes.
   3.3 Iniciar la ejecución del hilo con el método `start()`. Importante: si se invoca el método `run()` en lugar del método `start()`, se realiza una ejecución secuencial, no multihilo.

Los pasos en la clase `ThreadServidor` son los siguientes (método `run()`):

1. Preparar los flujos de entrada y salida.
2. Recibir el mensaje del servidor.
3. Enviar el mensaje de eco de vuelta al cliente.
4. Cerrar los flujos y la conexión del socket creado en el método `accept()`.

A continuación se muestra el pseudo-código del servidor multihilo:

```
import java.net.*;

/** Ejemplo que implementa un servidor de eco usando TCP. */

public class ServidorTCP {

  public static void main(String argv[]) {
    if (argv.length != 1) {
      System.err.println("Formato: ServidorTCP <puerto>");
      System.exit(-1);
    }

    try {
      // Creamos el socket del servidor
      // Establecemos un timeout de 30 segs
      while (true) {
        // Esperamos posibles conexiones
        // Creamos un objeto ThreadServidor, pasándole la nueva conexión
        // Iniciamos su ejecución con el método start()
      }
    } catch (SocketTimeoutException e) {
      System.err.println("30 segs sin recibir nada");
    } catch (Exception e) {
      System.err.println("Error: " + e.getMessage());
      e.printStackTrace();
   } finally{
       //Cerramos el socket del servidor
    }
  }
}
```

Pseudo-código de la clase `ThreadServidor`:

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

/** Thread que atiende una conexión de un servidor de eco. */

public class ThreadServidor extends Thread {

  Socket socket;

  public ThreadServidor(Socket s) {
    // Almacenamos el socket de la conexión
  }

  public void run() {
    try {
      // Establecemos el canal de entrada
      // Establecemos el canal de salida
      // Recibimos el mensaje del cliente
      // Enviamos el eco al cliente
      // Cerramos los flujos
    } catch (SocketTimeoutException e) {
      System.err.println("30 segs sin recibir nada");
    } catch (Exception e) {
      System.err.println("Error: " + e.getMessage());
    } finally {
       // Cerramos el socket
    }
  }
}
```

Una vez finalizado el servidor multihilo de eco TCP, para comprobar que funciona correctamente deberás ejecutar en un terminal el servidor:

```
java ServidorTCP 5000
```

El servidor se quedará esperando a que los clientes le envíen peticiones.

En otro terminal ejecuta un comando `nc` que se conecte con el servidor, y déjalo conectado:

```
nc localhost 5000
```

En otro terminal, deberás ejecutar el cliente TCP:

```
java ClienteTCP localhost 5000 “Probando mi servidor TCP”
```

En este caso, el servidor lanzará un hilo para atender la conexión realizada con el `nc`. Al ser multihilo, lanzará otro hilo de ejecución en paralelo para atender la petición realizada con el cliente de eco. El resultado debería ser que el servidor recibiese el mensaje del cliente y se lo reenviase, mientras que el cliente debería recibir la respuesta del servidor. Para finalizar la conexión del `nc`, simplemente introduce algo por teclado y pulsa ENTER: esto enviará un mensaje al servidor que responderá con el eco y cerrará la conexión.

5.2. Preguntas TCP

Una vez que hayas implementado el servidor TCP multihilo, responde a las siguientes preguntas:

1. En la clase `ServidorTCP`, ¿cuál es la diferencia entre invocar el método `start()` e invocar al método `run()` de la clase `ThreadServidor`?
2. ¿En dónde se cierra el nuevo socket creado en el `ServidorTCP` cuando se acepta una nueva conexión? ¿Por qué?

6. Lectura recomendada

Lesson: All About Sockets. The Java Tutorials.

Disponible en http://java.sun.com/docs/books/tutorial/networking/sockets/