# 5 Práctica 4: Programación de aplicaciones con sockets (I) Sockets TCP

## 5.1 Objetivos

El principal objetivo de esta práctica consiste en introducir la API (*Application Programming Interface*) de Java que permite utilizar *sockets*. Los sockets son uno de los principales mecanismos que se ofertan en la pila de protocolos TCP/IP desde la capa de transporte a la capa de aplicación. Con ellos se pueden construir aplicaciones que hacen uso de los protocolos TCP (*Transmission Control Protocol*) y UDP (*User Datagram Protocol*). En concreto, en esta práctica se usan sockets TCP para construir aplicaciones orientadas a conexión. A modo de ejemplo se propone construir una sencilla aplicación de mensajería entre dos ordenadores y un servicio básico de envío de ficheros de texto.

## 5.2 Introducción

La capa de transporte de la pila de protocolos TCP/IP se centra en temas relacionados con la entrega de los datos a otro ordenador. La capa de transporte realiza varias funciones siendo las principales la detección y recuperación ante errores y el control de flujo. En la pila de TCP/IP esta tarea la realizan los protocolos TCP (*Transmission Control Protocol*) y UDP (*User Datagram Protocol*). La principal diferencia entre ambos es que TCP proporciona una amplia variedad de servicios a la capa de aplicación (detección y recuperación ante errores, reorganización de los paquetes, control de flujo, etc.), mientras que UDP apenas oferta servicios. Esto hace que cada protocolo sea más adecuado para un tipo de aplicación diferente. Para entender la diferencia entre TCP y UDP conviene distinguir entre las siguientes formas de entregar los datos:

1. **Servicios orientados a conexión:** Requieren el establecimiento de un canal de comunicación (Circuito virtual) que permita una comunicación fiable. Una vez establecido el canal, los datos se entregan protocolo de trasporte que a su vez hará uso de la capa de Internet para distribuirlos. La capa de transporte se asegurará de que los datos han llegado de forma correcta y en orden. En caso de que no sea así, intentará corregir el error si es posible y, si no es posible recuperar el error, se informará a la capa de aplicación de que los datos no han sido debidamente entregados. Este es el enfoque seguido por el protocolo TCP.

2. **Servicios no orientados a conexión:** Los datos se envían como unidades independientes. Se entregan a la capa de transporte, quien intenta distribuirlos sin ningún tipo de garantias de que los datos llegan a destino ni de que van a llegar en orden. La capa de aplicación (no la de transporte) puede introducir mecanismos que permitan notificar o corregir errores. Este es el enfoque seguido por el protocolo UDP.

### 5.2.1 El protocolo TCP

TCP es usado por aplicaciones o protocolos del nivel de aplicación que requieren un envío fiable de los datos. Conceptualmente, se establece un circuito virtual entre dos computadoras (extremo a extremo). Así, las aplicaciones de la capa superior tienen certeza de que la información entregada a la capa TCP es enviada al ordenador situado al otro extremo del circuito virtual. De no ser así se notificará del fallo de comunicación a la capa de aplicación. Este circuito virtual se

mantiene sobre la capa de transporte IP, para la entrega de los datos de extremo a extremo, incluyendo las tareas de enrutamiento (Ver figura 4.1). En otras palabras, TCP sólo realiza parte de las funciones necesarias para entregar los datos entre aplicaciones mientras que el protocolo IP se encarga de los detalles de buscar los caminos más adecuados para enviar los datos y convertir los paquetes a tramas de acuerdo a las tecnologías implicadas en las redes físicas subyacentes (p.e. redes Ethernet o X.25, ver figura 4.1).

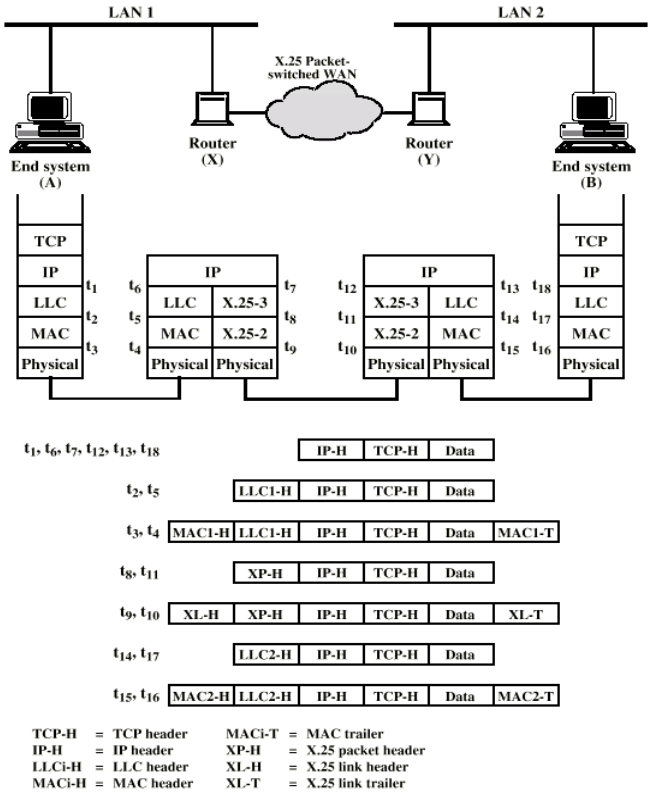


*Figura 4.1 Situación del protocolo TCP y relación con el resto de protocolos*

El protocolo TCP añade por cada paquete enviado una cabecera (ver figura 4.2) que incluye diversos tipos de información que van a permitir que en destino se descubran y recuperen algunos errores (*Checksum*, en realidad se trata de un código CRC) o reorganizar los paquetes (*Sequence Number* / *Acknowledgement Number*).

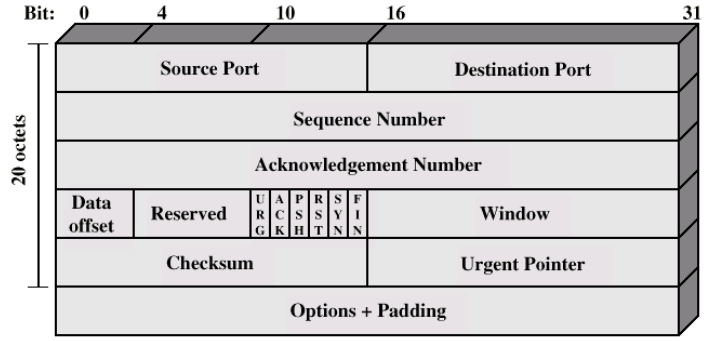


*Figura 4.2 Cabecera utilizada por el protocolo TCP*

Los principales servicios que TCP oferta a la capa de aplicación son:

1. **Multiplexación de conexiones usando puertos:** Permite que se establezcan varias conexiones virtuales entre las aplicaciones que se ejecutan dos ordenadores finales. (*Source Port*, *Destination Port*)
2. **Recuperación de errores:** Permite detectar la pérdida de paquetes a través de los números de secuencia de la comunicación (*Sequence Number* / *Acknowledgement Number*) y un código de integridad de los datos basado en técnicas CRC (*Checksum*)
3. **Control del flujo:** Permite definir búferes de envío y recepción de paquetes (*Window*)
4. **Establecimiento y terminación de conexiones:** Proceso utilizado para inicializar los números de puerto (*Source Port*, *Destination Port*) y los campos (*Sequence Number* / *Acknowledgement Number*).
5. **Transferencia de datos ordenados y segmentación de datos:** Proporciona una secuencia continua de bytes desde un proceso de la capa de aplicación en base a los campos (*Sequence Number* / *Acknowledgement Number*).

El paquete de la figura 4.2 precede a los datos de la aplicación, que irán codificados en diferentes tipos de protocolos de aplicación: HTTP, Telnet, FTP, SMTP o incluso protocolos diseñados *ad hoc*. Cabe señalar que el hecho de añadir una cabecera de 20-22 bytes como la mostrada en la figura 4.2 por cada paquete enviado introduce una sobrecarga en el rendimiento de envío de los protocolos subyacentes. En otras palabras, el uso del protocolo TCP implica un mayor consumo de ancho de banda y usa más ciclos de CPU de procesamiento. Además, el mantenimiento del circuito virtual puede requerir el reenvío de paquetes que será gestionada por los elementos finales de la comunicación en el protocolo TCP.

**Ejercicio Propuesto:**
Analizar la información contenida en las cabeceras TCP. ¿Por qué la cabecera TCP no tiene dirección IP de origen ni de destino? ¿Qué es el *Checksum*? Visualizar alguna cabecera TCP con Wireshark.

### 5.2.2 El concepto de puerto

Tal y como se ha indicado, el protocolo TCP (al igual que UDP) permite la multiplexación de las conexiones. Esto quiere decir que en el caso (habitual, por otra parte) de que una computadora esté ejecutando varias aplicaciones que requieren el uso simultáneo de una única tarjeta de red (p.e. un navegador web, un paquete de correo electrónico y una aplicación como *Skype*) se deben proporcionar mecanismos que permitan separar el tráfico entre las diferentes aplicaciones. Esta separación se hace con los puertos TCP / UDP. A cada aplicación se le asociará uno o más puertos para identificar el tráfico asociado en origen y en destino (Ver figura 4.3).

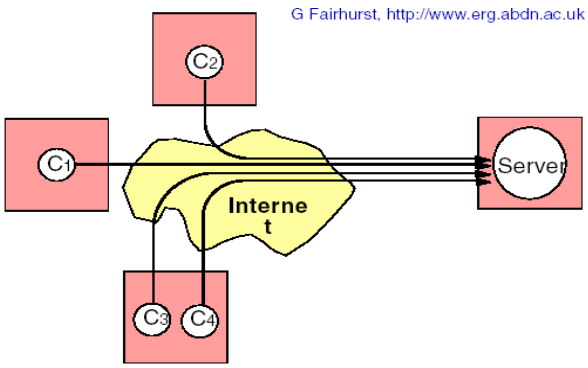


*Figura 4.3 Multiplexación del tráfico en el servidor*

A efectos de implementación el puerto es un número entero de 0 a 65535 ($2^{16}$-1) que identifica a cada aplicación en un ordenador destino de forma única, es decir, que dos aplicaciones no pueden utilizar el mismo número de puerto. Existen algunos números de puerto que están reservados a protocolos muy difundidos y que conviene no utilizar. De hecho, se recomienda no utilizar números de puerto menores de 1024. A modo de ejemplo, la tabla 4.1 muestra algunos valores de números de puerto reservados:

| Número de puerto | Protocolo | Aplicación |
|---|---|---|
| 20 | TCP | Datos FTP |
| 21 | TCP | Control FTP |
| 22 | TCP | SSH |
| 23 | TCP | Telnet |
| 25 | TCP | SMTP |
| 53 | TCP, UDP | DNS |
| 67, 68 | UDP | DHCP |
| 69 | UDP | TFTP |
| 80 | TCP | HTTP (www) |
| 110 | TCP | POP3 |
| 161 | UDP | SNMP |
| 443 | TCP | SSL |

*Tabla 4.1. Números de puerto reservados*

## 5.3 Sockets TCP

La multiplexión de las conexiones se apoya en el concepto de *socket*. Básicamente, la definición de un socket requiere tres tipos de información:

- Una dirección IP
- Un protocolo de transporte (TCP ó UDP)
- Un número de puerto

Por ejemplo, una aplicación de servidor web requeriría un socket destino (que en el caso del servidor web de la UPV/EHU, www.ehu.es, sería 158.227.0.65; un protocolo de transporte, TCP y un número de puerto: 80, puerto estándar para tráfico web, HTTP). Además, se requeriría en el

ordenador de origen otro socket que incluiría la dirección IP local (p.e. 158.227.63.15; un protocolo de transporte TCP y un puerto en origen que se asigna de forma automática al establecer la conexión con el destino).

**Ejercicio:** Se propone abrir varias sesiones con el navegador Web y los alumnos deben identificar la dirección IP local y los puertos de origen para cada conexión con el comando *netstat.*

### 5.3.1 API de Java para usar sockets

Desde el punto de vista del programador de aplicaciones se necesita acceder a la capa de transporte para poder enviar y recibir información. Esto normalmente se hace con una API (*Application Programming Interface*) que permite manipular los sockets. Existen APIs diferentes en diferentes lenguajes y sistemas operativos, siendo el API original la desarrollada para ser programada en C en Berkeley Unix. En esta práctica se usará el API de Java para comunicaciones con sockets. Más en concreto, las funciones que permiten utilizar sockets desde programas realizados en Java se encuentran en la librería `java.net`.

El paquete `java.net` (ver http://download.oracle.com/javase/1.5.0/docs/api/java/net/package-summary.html) ofrece diversas clases que permiten la creación de aplicaciones distribuidas en base al uso de sockets. La tabla 4.2 muestra algunas de las clases principales que proporciona el paquete `java.net` que se utilizan para la programación de aplicaciones que utilizan los protocolos TCP y UDP.

| Clase | Ser |
|---|---|
| InetAddress | Esta clase representa una dirección IP. Una instancia de InetAddres consiste en una dirección IP y su correspondiente su *hostname*. Proporciona métodos para resolver las direcciones IP buscando en los DNS por defecto. |
| Socket | Esta clase se usa para implementar sockets cliente (También llamados *sockets*). Un socket es un punto final para la comunicación entre dos ordenadores. |
| ServerSocket | Esta clase implementa sockets servidor. Estos sockets esperan peticiones que vienen de la red. Realizan operaciones basados en estas peticiones y normalmente devuelven el resultado de la operación al peticionario. |
| DatagramSocket | Esta clase representa un socket para enviar y recibir datagramas. Una instancia de esta clase es el punto de envío y recepción de un servicio de distribución de paquetes. Cada paquete enviado o recibido en un socket datagrama se enruta de forma diferente. Varios paquetes enviados de una máquina a otra pueden enrutarse de forma diferente y pueden llegar en desorden. |
| DatagramPacket | Esta clase representa un datagrama. Los datagramas se usan para implementar un servicio de entrega de paquetes no orientado a conexión. Cada mensaje se enruta de una máquina a otra basada únicamente en la información contenida en cada paquete. Los paquetes enviados de una máquina a otra pueden ser enrutados de forma diferente y pueden llegar en cualquier orden. Además,la entrega de los paquetes no está garantizada. |
| MulticastSocket | Esta clase se usa para enviar y recibir paquetes IP multicast. Un objeto de la clase MulticastSocket hereda su estructura de la clase DatagramSocket, con propiedades adicionales para unirse a grupos o otros hosts multicast en |

| | |
|---|---|
| | Internet.<br>Un grupo multicast se especifica con una dirección IP de tipo D (entre `224.0.0.0` y `239.255.255.255`, ambas incluidas). |

*Tabla 4.2.Principales clases del paquete java.net*

### 5.3.2 Ejemplo 1: Programa para determinar las direcciones IP

La aplicación de la Tabla 4.3 hace uso de la API de java para obtener la dirección IP de una interfaz de red a partir de su nombre.

```
import java.net.*;
import java.util.*;
import java.lang.*;

public class VerIP {
   public static void main(String args[]) {
        InetAddress direccion;
        String nOrdenador;
        Scanner sc = new Scanner(System.in);
        try {
           System.out.println("Nombre del ordenador");
           nOrdenador= sc.nextLine();
           direccion = InetAddress.getByName(nOrdenador);

           System.out.print("Direccion IP: ");
           System.out.println(direccion.getHostAddress());
        } catch (UnknownHostException e) {
           System.out.println("Ordenador desconocido");
        } catch (Exception e) {
           System.out.println("Error en la lectura");
        }
   }
}
```

**Ejercicio Propuesto:**

Los alumnos deben compilar y ejecutar el programa anterior con diferentes nombres de ordenador. P.e. *www.ehu.es, ftp.rediris.es* y *www.nasa.gov* y anótese cuáles son sus direcciones IP. Compárese este resultado con el resultado obtenido de ejecutar el comando *nslookup*.

Mirar en el API de `java.net` el funcionamiento de las clases involucradas en el progama anterior.

### 5.3.3 Paradigma Cliente / Servidor

Se entiende por paradigma Cliente / Servidor un modelo de comunicación en el que un programa, el Cliente, realiza peticiones a otro programa, el Servidor, que le da respuesta. Este es uno de los modelos de comunicación más extendidos para realizar aplicaciones distribuidas, dado que la capacidad de procesamiento está repartida entre clientes y servidores. Una de las principales características de este modelo es que a nivel conceptual resulta sencillo diferenciar unidades de procesamiento (servidores) que realizan una funcionalidad o servicio definido frente a otras unidades de procesamiento (clientes) que utilizan la información procesada en los servidores.

Algunos ejemplos de servidores frecuentemente encontrados en redes TCP/IP incluyen servidores Web, servidores de ficheros, de correo electrónico, etc.

### 5.3.4 Ejemplo 2: Aplicación Cliente / Servidor

A modo de ejemplo, se muestra el código de dos programas, un servidor y un cliente. El siguiente listado es el del servidor que se queda a la escucha de que algún cliente le envíe una cadena de texto. Una vez que el servidor recibe la cadena, la convierte en mayúsculas y la devuelve al cliente que se la envío.

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

public class ServidorMayusculas {
/**
 * Programa Servidor que convierte una cadena en mayusculas y la devuelve
 */
  public static void main(String[] args) {
     boolean finalizar=false;
     String cadenaEntrada;
     String cadenaSalida;
     try {
       //Crea un servidor en un puerto concreto: 5002
       ServerSocket socketServidor=new ServerSocket(5002);
       int iPuertoServidor = socketServidor.getLocalPort();
       System.out.println("El servidor usa el puerto: " + iPuertoServidor);
       do {
          //Espera que se realice una conexion en este socket y la acepta
          Socket socketCliente = socketServidor.accept();

          String ordenadorEntrante = socketCliente.getInetAddress().getHostName();
          int puertoSalida = socketCliente.getPort();
          System.out.println("Aceptada la conexion de: " + ordenadorEntrante + ":" + puertoSalida);

          //Se obtiene un stream para recibir la cadena desde el cliente
          DataInputStream bufferEntrada = new DataInputStream(socketCliente.getInputStream());
          //Se obtiene un stream para enviar la cadena modificada al cliente
          DataOutputStream bufferSalida = new DataOutputStream(socketCliente.getOutputStream());

          //*****Recibe un mensaje del cliente, lo procesa y devuelve la respuesta al cliente
          System.out.println("Servidor esperando que le envíen un mensaje...");
          cadenaEntrada= bufferEntrada.readUTF();
          System.out.println("Texto recibido: " + cadenaEntrada);
          if(cadenaEntrada.equalsIgnoreCase("fin"))
             finalizar=true;

          cadenaSalida = ProcesarServicio(cadenaEntrada);

          bufferSalida.writeUTF(cadenaSalida);
          System.out.println("Texto enviado: " + cadenaSalida);

          //Se cierra circuito virtual
          socketCliente.close();
       } while (!finalizar);
       //Se cierra el socket del servidor y el servidor deja de aceptar conexiones
       socketServidor.close();

       }catch (IOException e) {
          System.out.println(e.getMessage());
       }
    }
    public static String ProcesarServicio(String sCadena){
        //Convierte la cadena en mayusculas
        return sCadena.toUpperCase();
    }
}
```

A continuación se muestra el código del programa cliente que tras solicitar la dirección IP y el puerto donde se encuentra el servidor (es decir, a dónde debe dirigirse), solicita una cadena de texto y espera que el servidor se la devuelva convertida en mayúsculas.

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

public class ClienteMayusculas {
/**
 * Programa cliente: Envía una cadena al servidor quien la convierte en mayuscula
```

```
 */
  public static void main(String[] args) {
     String nOrdenadorDestino;
     String sCadena, sCadenaMayusculas;
     int iPuertoDestino=4096; // Por defecto se usa el puerto 4096
     Socket conexion = null;

     if (args.length !=2) {
         System.out.println("Usar: java ClienteMayusculas <IP ordenador destino> <Puerto destino>");
     } else {
         try {
             iPuertoDestino= Integer.parseInt(args[1]);
         } catch (NumberFormatException e) {
             System.out.println("Numero de puerto erroneo");
          }
         try {
         // Se crea el socket a partir de la direccion destino y num. de puerto
             conexion = new Socket (args[0], iPuertoDestino);
         } catch (UnknownHostException e) {
              System.out.println("Ordenador desconocido");
         } catch (IOException e) {
              System.out.println("Error al crear el socket");
              return;
         }
         System.out.println("Conexion a " + conexion.getRemoteSocketAddress());
         //Pregunta la cadena a enviar al servidor
         Scanner sc = new Scanner(System.in);
         System.out.print("Cadena a enviar al servidor -> ");
         sCadena = sc.nextLine();

         try {
            //Se obtiene un stream para enviar informacion al servidor y se envia la cadena
            DataOutputStream bufferSalida = new DataOutputStream (conexion.getOutputStream());
            //Se obtiene un stream para recibir la cadena en mayusculas
            DataInputStream bufferEntrada = new DataInputStream(conexion.getInputStream());

            //Envío de la cadena
            bufferSalida.writeUTF(sCadena);
            System.out.println("Se ha enviado -> " + sCadena);

            //Se recibe la cadena enviada desde el servidor
            sCadenaMayusculas= bufferEntrada.readUTF();
            System.out.println("Cadena recibida del servidor -> " + sCadenaMayusculas);
         } catch (IOException e) {
            System.out.println(e.getMessage());
         }
     }
  }
}
```

**Ejercicio Propuesto:**
Modificar el ejemplo anterior para construir un chat que pueda usarse por dos personas para intercambiarse mensajes.

## 5.4 Tareas a realizar

Los alumnos deben crear un servicio de s que permita enviar un fichero de texto de una longitud indeterminada de un ordenador a otro.

Se recomienda que los alumnos busquen en Internet más información de cómo usar los sockets.
P.e.
http://www.chuidiang.com/java/sockets/socket.php
http://www.chuidiang.com/clinux/sockets/sockets_simp.php
http://www.scribd.com/doc/19795778/Sockets-y-su-Programacion-en-Java