

# **MÓDULO PROGRAMACIÓN DE SERVICIOS Y PROCESOS**

## **TEMA 3 PROGRAMACIÓN DE COMUNICACIONES EN RED**

**CICLO DE GRADO SUPERIOR INFÓRMATICA  
DESARROLLO DE APLICACIONES MULTIPLATAFORMA**

**Autor: Luis Miguel Lestón López a partir de los materiales generados por el Ministerio de Educación para este módulo. Obra derivada.**

Este documento se publica bajo licencia Creative Commons No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.



# ÍNDICE

1. Conceptos básicos	4
2. Sockets TCP	9
3. Sockets UDP	16
4. Paradigma cliente / servidor	20
5. Optimización de sockets	26

# 1. Conceptos básicos

Con la fuerte expansión que ha tenido Internet se ha generalizado la utilización de redes en las empresas y en nuestros hogares. Hoy en día para un empresa es totalmente necesario disponer de una red interna que le permita compartir información, conectarse a Internet o incluso, ofrecer sus servicios en Internet.

Cada día es más frecuente que las empresas utilicen aplicaciones que se comunican por Internet para poder compartir información entre sus empleados. Por ejemplo, aplicaciones de gestión y facturación que permiten que varias tiendas puedan realizar de forma centralizada la facturación de toda la empresa, gestionar el stock, etc.

A continuación vamos a ver los conceptos más importantes sobre redes, necesarios para más adelante poder programar nuestras aplicaciones.



## 1.1. Recordando TCP/IP

En 1969 la agencia ARPA (Advanced Research Projects Agency) del Departamento de Defensa de los Estados Unidos inició un proyecto de interconexión de ordenadores mediante redes telefónicas. Al ser un proyecto desarrollado por militares en plena guerra fría un principio básico de diseño era que la red debía poder resistir la destrucción de parte de su infraestructura (por ejemplo a causa de un ataque nuclear), de forma que dos nodos cualesquiera pudieran seguir comunicados siempre que hubiera alguna ruta que los uniera. Esto se consiguió en 1972 creando una red de conmutación de paquetes denominada ARPAnet, la primera de este tipo que operó en el mundo. La conmutación de paquetes unida al uso de topologías malladas mediante múltiples líneas punto a punto dio como resultado una red altamente fiable y robusta.



ARPAnet fue creciendo paulatinamente, y pronto se hicieron experimentos utilizando otros medios de transmisión de datos, en particular enlaces por radio y vía satélite; los protocolos existentes tuvieron problemas para interoperar con estas redes, por lo que se diseñó un nuevo conjunto o pila de protocolos, y con ellos una arquitectura. Este nuevo conjunto se denominó TCP/IP (Transmission Control Protocol/Internet Protocol), nombre que provenía de los dos protocolos más importantes que componían la pila; la nueva arquitectura se llamó sencillamente modelo TCP/IP. A la nueva red, que se creó como consecuencia de la fusión de ARPAnet con las redes basadas en otras tecnologías de transmisión, se la denominó Internet.

La aproximación adoptada por los diseñadores del TCP/IP fue mucho más pragmática que la de los autores del modelo OSI. Mientras que en el caso de OSI se emplearon varios años en definir con sumo cuidado una arquitectura de capas donde la función y servicios de cada una estaban perfectamente definidas, y sólo después se planteó desarrollar los protocolos para cada una de ellas, en el caso de TCP/IP la operación fue a la inversa; primero se especificaron los protocolos, y luego se definió el modelo como una simple descripción de los protocolos ya existentes. Por este motivo el modelo TCP/IP es mucho más simple que el OSI. También por este motivo el modelo OSI se utiliza a menudo para describir otras arquitecturas, como por ejemplo TCP/IP, mientras que el modelo TCP/IP nunca suele emplearse para describir otras arquitecturas que no sean la suya propia.

### 1.1.1. Recordando TCP/IP (II)

El modelo TCP/IP tiene sólo cuatro capas:

- **La capa host-red.** Esta capa permite comunicar el ordenador con el medio que conecta el equipo a la red. Para ello primero debe permitir convertir la información en impulsos físicos (p.ej. eléctricos, magnéticos, luminosos) y además, debe permitir las conexiones entre los ordenadores de la red. En esta capa se realiza un direccionamiento físico utilizando las direcciones MAC
- **La capa de red.** Esta capa es el eje de la arquitectura TCP/IP ya que permite que los equipos envíen paquetes en cualquier red y viajen de forma independiente a su destino (que podría estar en una red diferente). Los paquetes pueden llegar incluso en un orden diferente a aquel en que se enviaron, en cuyo caso corresponde a las capas superiores reordenarlos, si se desea la entrega ordenada. La capa de red define un formato de paquete y protocolo oficial llamado IP (Internet Protocol). El trabajo de la capa de red es entregar paquetes IP a su destino. Aquí la consideración más importante es decidir el camino que tienen que seguir los paquetes (encaminamiento), y también evitar la congestión. En esta capa se realiza el direccionamiento lógico o direccionamiento por IP, ya que ésta es la capa encargada de enviar un determinado mensaje a su dirección IP de destino
- **La capa de transporte.** La capa de transporte permite que los equipos lleven a cabo una conversación. Aquí se definieron dos protocolos de transporte: TCP (Transmission Control Protocol) y UDP (User Datagram Protocol). El protocolo TCP es un protocolo orientado a conexión y fiable, y el protocolo UDP es un protocolo no orientado a conexión y no fiable. En esta capa además se realiza el direccionamiento por puertos. Gracias a la capa anterior, los paquetes viajan de un equipo origen a un equipo destino. La capa de transporte se encarga de que la información se envíe a la aplicación adecuada (mediante un determinado puerto).

- **La capa de aplicación:** Esta capa engloba las funcionalidades de las capas de sesión, presentación aplicación del modelo OSI. Incluye todos los protocolos de alto nivel relacionados con las aplicaciones que se utilizan en Internet (por ejemplo HTTP, FTP, TELNET).

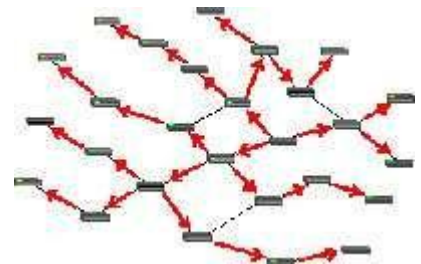


*Modelo TCP/IP.*

## 1.2. Conexiones TCP/UDP

Como se ha visto anteriormente, la capa de transporte cumple la función de establecer las reglas necesarias para establecer una conexión entre dos dispositivos.

Desde la capa anterior, la capa de red, la información se recibe en forma de paquetes desordenados y la capa de transporte debe ser capaz de manejar dichos paquetes y obtener un único flujo de datos. Recuerde, que la capa de red en la arquitectura TCP/IP no se preocupa del orden de los paquetes ni de los errores, es en esta capa donde se deben cuidar estos detalles.



La capa de transporte del modelo TCP/IP es equivalente a la capa de transporte del modelo OSI, por lo que es el encargado de la transferencia libre de errores de los datos entre el emisor y el receptor, aunque no estén directamente conectados, así como de mantener el flujo de la red. La tarea de este nivel es proporcionar un transporte de datos confiable de la máquina de origen a la máquina destino, independientemente de la red física.

Existen dos tipos de conexiones:

- **TCP (Transmission Control Protocol).** Es un protocolo orientado a la conexión que permite que un flujo de bytes originado en una máquina se entregue sin errores en cualquier máquina destino. Este protocolo fragmenta el flujo entrante de bytes en mensajes y pasa cada uno a la capa de red. En el diseño, el proceso TCP receptor reensambla los mensajes recibidos para formar el flujo de salida. TCP también se encarga del control de flujo para asegurar que un emisor rápido no pueda saturar a un receptor lento con más mensajes de los que pueda gestionar.

- **UDP (User Datagram Protocol).** Es un protocolo sin conexión, para aplicaciones que no necesitan la asignación de secuencia ni el control de flujo TCP y que desean utilizar los suyos propios. Este protocolo también se utilizan para las consultas de petición y respuesta del tipo cliente-servidor, y en aplicaciones en las que la velocidad es más importante que la entrega precisa, como las transmisiones de voz o de vídeo. Uno de sus usos es en la transmisión de audio y vídeo en tiempo real, donde no es posible realizar retransmisiones por los estrictos requisitos de retardo que se tiene en estos casos.

De esta forma, a la hora de programar nuestra aplicación deberemos elegir el protocolo que queremos utilizar según nuestras necesidades: TCP o UDP.

### 1.3. Puertos de comunicación

Con la capa de red se consigue que la información vaya de un equipo origen a un equipo destino a través de su dirección IP. Pero para que una aplicación pueda comunicarse con otra aplicación es necesario establecer a qué aplicación se conectará. El método que se emplea es el de definir direcciones de transporte en las que los procesos pueden estar a la escucha de solicitudes de conexión. Estos puntos terminales se llaman puertos.

Aunque muchos de los puertos se asignan de manera arbitraria, ciertos puertos se asignan, por convenio, a ciertas aplicaciones particulares o servicios de carácter universal. De hecho, la IANA (Internet Assigned Numbers Authority) determina, las asignaciones de todos los puertos. Existen tres rangos de puertos establecidos:



- **Puertos conocidos [0, 1023].** Son puertos reservados a aplicaciones de uso estándar como: 21 – FTP (File Transfer Protocol), 22 – SSH (Secure SHell), 53 – DNS (Servicio de nombres de dominio), 80 – HTTP (Hypertext Transfer Protocol), etc.
- **Puertos registrados [1024, 49151].** Estos puertos son asignados por IANA para un servicio específico o aplicaciones. Estos puertos pueden ser utilizados por los usuarios libremente.
- **Puertos dinámicos [49152, 65535].** Este rango de puertos no puede ser registrado y su uso se establece para conexiones temporales entre aplicaciones.

Cuando se desarrolla una aplicación que utilice un puerto de comunicación, optaremos por utilizar puertos comprendidos entre el rango 1024-49151.

## 1.4. Nombres en Internet

Los equipos informáticos se comunican entre sí mediante una dirección IP como 193.147.0.29. Sin embargo nosotros preferimos utilizar nombres como www.mec.es porque son más fáciles de recordar y porque ofrecen la flexibilidad de poder cambiar la máquina en la que están alojados (cambiaría entonces la dirección IP) sin necesidad de cambiar las referencias a él.

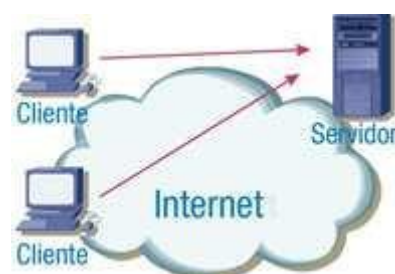
El sistema de resolución de nombres (DNS) basado en dominios, en el que se dispone de uno o más servidores encargados de resolver los nombres de los equipos pertenecientes a su ámbito, consiguiendo, por un lado, la centralización necesaria para la correcta sincronización de los equipos, un sistema jerárquico que permite una administración focalizada y, también, descentralizada y un mecanismo de resolución eficiente.

A la hora de comunicarse con un equipo, puedes hacerlo directamente a través de su dirección IP o puede poner su entrada DNS (p.ej. servidor.miempresa.com). En el caso de utilizar la entrada DNS el equipo resuelve automáticamente su dirección IP a través del servidor de nombres que utilice en su conexión a Internet.

## 1.5. Modelos de comunicaciones

Sin duda alguna, el modelo de comunicación que ha revolucionado los sistemas informáticos es el modelo cliente/servidor. El modelo cliente/servidor está compuesto por un servidor que ofrece una serie de servicios y unos clientes que acceden a dichos servicios a través de la red.

Por ejemplo, el servicio más importante de Internet, el WWW, utiliza el modelo cliente/servidor. Por un lado tenemos el servidor que alojan las páginas web y por otro lado los clientes que solicitan al servidor una determinada página web.



*Modelo Cliente/Servidor*

Otro modelo ampliamente utilizado son los Sistemas de Información Distribuidos. Un Sistema de Información Distribuido está compuesto por un conjunto de equipos que interactúan entre sí y pueden trabajar a la vez como cliente y servidor. Desde el punto de vista externo es igual que un sistema cliente/servidor ya que el cliente ve al Sistema de Información Distribuido como una entidad. Internamente, los equipos del Sistema



de Información Distribuido interactúan entre sí (actuando como servidores y clientes de forma simultánea) para compartir información, recursos, realizar tareas, etc.



*Modelo Sistema de Información Distribuido*

## 2. Sockets TCP

Los sockets permiten la comunicación entre procesos de diferentes equipos de una red. Un socket, es un punto de información por el cual un proceso puede recibir o enviar información.

Tal y como hemos visto anteriormente, existen dos tipos de protocolos de comunicación: TCP o UDP. En el protocolo TCP es un protocolo orientado a la conexión que permite que un flujo de bytes originado en una máquina se entregue sin errores en cualquier máquina destino. Por otro lado, el protocolo UDP, no orientado a conexión, se utiliza para comunicaciones donde se prioriza la velocidad sobre la pérdida de paquetes.

A la hora de crear un socket hay que tener claro el tipo de socket que se quiere crear (TCP o UDP). En esta sección vamos a aprender a utilizar los sockets TCP y en el siguiente punto veremos los sockets UDP.

Al utilizar sockets TCP, el servidor utiliza un puerto por el que recibe las diferentes peticiones de los clientes. Normalmente, el puerto del servidor es un puerto bajo [1-1023].



*Socket TCP - Conexión*

Cuando el cliente realiza la conexión con el servidor, a partir de ese momento se crea un nuevo socket que será el encargado de permitir el envío y recepción de datos entre el cliente/servidor. El puerto se crea de forma dinámica y se encuentra en el rango 49152 y 65535. De ésta forma, el puerto por donde se reciben las conexiones de los clientes queda libre y cada comunicación tiene su propio socket.



*TCP: transferencia de datos*

El paquete **java.net** de Java proporciona la clase **Socket** que permite la comunicación por red. De forma adicional, **java.net** incluye la clase **ServerSocket**, que permite a un servidor escuchar y recibir las peticiones de los clientes por la red. Y la clase **Socket** permite a un cliente conectarse a un servidor para enviar y recibir información.

## 2.1. Servidor

Los pasos que realiza el servidor para realizar una comunicación son:

- **Publicar puerto.** Se utiliza el comando **ServerSocket** indicando el puerto por donde se van a recibir las conexiones.
- **Esperar peticiones.** En este momento el servidor queda a la espera a que se conecte un cliente. Una vez que se conecte un cliente se crea el socket del cliente por donde se envían y reciben los datos.
- **Envío y recepción de datos** . Para poder recibir/enviar datos es necesario crear un flujo (stream) de entrada y otro de salida. Cuando el servidor recibe una petición, éste la procesa y le envía el resultado al cliente.
- Una vez finalizada la comunicación se **cierra el socket del cliente**.



*Esquema de funcionamiento interno del modelo cliente/servidor*

Para publicar el puerto del servidor se utiliza la función **ServerSocket** a la que hay que indicarle el puerto a utilizar. Su estructura es:

```
ServerSocket skServidor = new ServerSocket(Puerto);
```

Una vez publicado el puerto, el servidor utiliza la función **accept()** para esperar la conexión de un cliente. Una vez que el cliente se conecta, entonces se crea un nuevo socket por donde se van a realizar todas las comunicaciones con el cliente y servidor.

```
Socket sCliente = skServidor.accept();
```

Una vez recibida la petición del cliente el servidor se comunica con el cliente a través de streams de datos que veremos en el siguiente punto.

Finalmente, una vez terminada la comunicación se cierra el socket de la siguiente forma:

```
sCliente.close();
```

A continuación se muestra el código comentado de un servidor

```
import java.io.* ;
import java.net.* ;
class Servidor {
    static final int Puerto=2000;
    public Servidor( ) {
        try {
            // Inicio la escucha del servidor en un determinado puerto
            ServerSocket skServidor = new ServerSocket(Puerto);
            System.out.println("Escucho el puerto " + Puerto );
            // Espero a que se conecte un cliente y creo un nuevo socket para el
            // cliente
            Socket sCliente = skServidor.accept();
            // ATENDER PETICIÓN DEL CLIENTE
            // Cierro el socket
            sCliente.close();
        }
        } catch( Exception e ) { System.out.println( e.getMessage() );
    }
}
```

## 2.2. Cliente

Los pasos que realiza el cliente para realizar una comunicación son:

- **Conectarse con el servidor.** El cliente utiliza la función **Socket** para indicarse con un determinado servidor a un puerto específico. Una vez realizada la conexión se crea el socket por donde se realizará la comunicación.
- **Envío y recepción de datos.** Para poder recibir/enviar datos es necesario crear un flujo (stream) de entrada y otro de salida.
- Una vez finalizada la comunicación se **cierra el socket**.



*Esquema de funcionamiento interno del modelo cliente/servidor.*

Para conectarse a un servidor se utiliza la función **Socket** indicando el equipo y el puerto al que desea conectarse. Su sintaxis es:

```
Socket sCliente = new Socket( Host , Puerto );
```

donde Host es un string que guarda el nombre o dirección IP del servidor y Puerto es una variable del tipo **int** que guarda el puerto.

Si lo prefiere también puede realizar la conexión directamente:

```
Socket sCliente = new Socket("192.168.1.200", 1500);
```

Una vez establecida la comunicación, se crean los streams de entrada y salida para realizar las diferentes comunicaciones entre el cliente y el servidor. En el siguiente apartado veremos la creación de streams.

Finalmente, una vez terminada la comunicación se cierra el socket de la siguiente forma:

```
sCliente.close();
```

A continuación se muestra el código comentado de un cliente.

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

class Cliente {

    static final String Host = "localhost";
    static final int Puerto=2000;

    public Cliente( ) {
        try{

            // Me conecto al servidor en un determinado puerto Socket
            sCliente = new Socket( Host, Puerto );

            // TAREAS QUE REALIZA EL CLIENTE
            // Cierro el socket sCliente.close();

        } catch( Exception e ) {

            System.out.println( e.getMessage() );

        }

    }

    public static void main( String[] arg )
    {
        new Cliente();
    }
}
```

## 2.3. Flujo de Entrada y de Salida

Una vez establecida la conexión entre el cliente y el servidor se inicializa la variable del tipo Socket que en el ejemplo se llama sCliente. Para poder enviar o recibir datos a través del socket es necesario establecer un stream (flujo) de entrada o de salida según corresponda.

Continuando con el ejemplo anterior, a continuación se va a establecer un stream de salida llamado flujo\_salida.

```
OutputStream aux = sCliente.getOutputStream();
```

```
DataOutputStream flujo_salida= new DataOutputStream( aux );
```

o lo que es lo mismo,

```
DataOutputStream flujo_salida= new
```

```
DataOutputStream(sCliente.getOutputStream());
```

A partir de éste momento puede enviar información de la siguiente forma:

```
flujo_salida.writeUTF( "Enviar datos");
```

De forma análoga, puede establecer el stream de entrada de la siguiente forma:

```
InputStream aux = sCliente.getInputStream();
```

```
DataInputStream flujo_entrada = new DataInputStream( aux );
```

o lo que es lo mismo,

```
DataInputStream flujo_entrada = new
```

```
DataInputStream(sCliente.getInputStream());
```

A continuación se muestra una forma cómoda de recibir información

```
String datos=new String();
```

```
Datos=flujo_entrada.readUTF();
```



## 2.4. Ejemplo

Para continuar con el ejemplo anterior y poder utilizar los sockets para enviar información vamos a realizar un ejemplo muy sencillo en el que el servidor va a aceptar tres clientes (de forma secuencial no concurrente) y le va a indicar el número de cliente que es.

### Servidor.java

```
import java.io.* ;
import java.net.* ;
class Servidor {
    static final int Puerto=2000;
    public Servidor( ) {
        try {
            ServerSocket skServidor = new ServerSocket(Puerto);
            System.out.println("Escucho el puerto " + Puerto );
            for ( int nCli = 0; nCli < 3; nCli++) {
                Socket sCliente = skServidor.accept();
                System.out.println("Sirvo al cliente " + nCli);
                OutputStream aux = sCliente.getOutputStream();
                DataOutputStream flujo_salida= new DataOutputStream( aux );
                flujo_salida.writeUTF( "Hola cliente " + nCli );
                sCliente.close();
            }
            System.out.println("Ya se han atendido los 3 clientes");
        } catch( Exception e ) {
            System.out.println( e.getMessage() );
        }
    }
    public static void main( String[] arg ) {
        new Servidor();
    }
}
```

### 2.4.1. Ejemplo (II)

De la misma forma, creamos el cliente con el siguiente código:

#### **Cliente.java**

```
import java.io.*;
import java.net.*;
class Cliente {
    static final String HOST = "localhost";
    static final int Puerto=2000;
    public Cliente( ) {
        try{
            Socket sCliente = new Socket( HOST , Puerto );
            InputStream aux = sCliente.getInputStream();
            DataInputStream flujo_entrada = new DataInputStream( aux );
            System.out.println( flujo_entrada.readUTF() );
            sCliente.close();
        } catch( Exception e ) {
            System.out.println( e.getMessage() );
        }
    }
    public static void main( String[] arg ) {
        new Cliente();
    }
}
```

Al ejecutar el cliente éste se conecta al servidor y muestra el mensaje que le envía el servidor.

## 3. Sockets UDP

En el caso de utilizar sockets UDP no se crea una conexión (como es el caso de socket TCP) y básicamente permite enviar y recibir mensajes a través de una dirección IP y un puerto. Estos mensajes se gestionan de forma individual y no se garantiza la recepción o envío del mensaje como si ocurre en TCP.

Para utilizar sockets UDP en java tenemos la clase **DatagramSocket** y para recibir o enviar los mensajes se utiliza clase **DatagramPacket**. Cuando se recibe o envía un paquete se hace con la siguiente información: mensaje, longitud del mensaje, equipo y puerto.

### 3.1. Receptor

En el caso de querer iniciar el socket en un determinado puerto se realiza de la siguiente forma:

```
DatagramSocket sSocket = new DatagramSocket(puerto);
```

Una vez iniciado el socket ya estamos en disposición de recibir mensajes utilizando la clase **DatagramPacket**. Cuando se recibe o envía un paquete se hace con la siguiente información: mensaje, longitud del mensaje, equipo y puerto.

A continuación se muestra un código de ejemplo para recibir un mensaje:

```
byte [] cadena = new byte[1000] ;  
  
DatagramPacket mensaje = new DatagramPacket(cadena, cadena.length);  
  
sSocket.receive(mensaje);
```

Una vez recibido el mensaje puede mostrar su contenido de la siguiente forma:

```
String datos=new String(mensaje.getData(),0,mensaje.getLength());  
  
System.out.println("\tMensaje Recibido: " +datos);
```

Finalmente, una vez terminado el programa cerramos el socket:

```
sSocket.close();
```

### 3.2. Emisor

Por otro lado, para realizar una aplicación emisora de mensajes UDP debe inicializar primero la estructura **DatagramSocket**.

```
DatagramSocket sSocket = new DatagramSocket();
```



Ahora debe crear el mensaje del tipo **DatagramPacket** al que debe indicar:

- Mensaje a enviar.
- Longitud del mensaje.
- Equipo al que se le envía el mensaje.
- Puerto destino.

A continuación se muestra un ejemplo para crear un mensaje:

```
DatagramPacket mensaje = new DatagramPacket(mensaje,longitud_mensaje, Equipo, Puerto);
```

Para obtener la dirección del equipo al que se le envía el mensaje a través de su nombre se utiliza la función `getByName` de la clase **InetAddress** de la siguiente forma

```
InetAddress Equipo = InetAddress.getByName("localhost");
```

Una vez creado el mensaje lo enviamos con la función **send()**:

```
sSocket.send(mensaje);
```

Finalmente, una vez terminado el programa cerramos el socket:

```
sSocket.close();
```

### 3.3. Ejemplo

A continuación, para aprender a programar Sockets UDP se va a realizar un ejemplo sencillo donde intervienen dos procesos:

- **ReceptorUDP**. Inicia el puerto 1500 y muestra en pantalla todos los mensajes que llegan a él.
- **EmisorUDP**. Permite enviar por líneas de comandos mensajes al receptor por el puerto 1500.

#### ReceptorUDP.java

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

public class ReceptorUDP {
    public static void main(String args [] ) {
        // Sin argumentos
        if (args.length != 0) {
            System.err.println("Uso: java ReceptorUDP");
        }
        else try{
            // Crea el socket
            DatagramSocket sSocket = new DatagramSocket(1500);

            // Crea el espacio para los mensajes
```

```
byte [] cadena = new byte[1000] ;
DatagramPacket mensaje = new DatagramPacket(cadena, cadena.length);

System.out.println("Esperando mensajes..");
while(true){
    // Recibe y muestra el mensaje
    sSocket.receive(mensaje);
    String datos=new String(mensaje.getData(),0,mensaje.getLength());
    System.out.println("\tMensaje Recibido: " +datos);
}
} catch(SocketException e) {
System.err.println("Socket: " + e.getMessage());
} catch(IOException e) {
    System.err.println("E/S: " + e.getMessage()); }
}
}
```

### 3.3.1. Ejemplo (II)

#### EmisorUDP.java

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

public class EmisorUDP {
    public static void main(String args [] ) {
        // Comprueba los argumentos
        if (args.length != 2) {
            System.err.println("Uso: java EmisorUDP maquina mensaje");
        }
        else try{
            // Crea el socket
            DatagramSocket sSocket = new DatagramSocket();
            // Construye la dirección del socket del receptor
            InetAddress maquina = InetAddress.getByName(args[0]);
            int Puerto = 1500;

            // Crea el mensaje
            byte [] cadena = args[1].getBytes();
            DatagramPacket mensaje = new DatagramPacket(cadena,args[1].length(), maquina,
                                                         Puerto);

            // Envía el mensaje
            sSocket.send(mensaje);

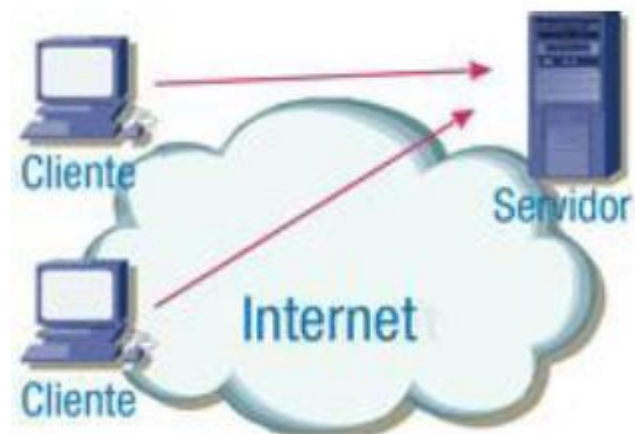
            // Cierra el socket
            sSocket.close();
        } catch(UnknownHostException e) {
            System.err.println("Desconocido: " + e.getMessage());
        } catch(SocketException e) {
            System.err.println("Socket: " + e.getMessage());
        } catch(IOException e) {
            System.err.println("E/S: " + e.getMessage());
        }
    }
}
```

## 4. Paradigma cliente/servidor

- “Sistema en el cual múltiples procesadores autónomos, posiblemente de diferente tipo, están interconectados por una subred de comunicación para interactuar de una manera cooperativa en el logro de un objetivo global”. [Lelann 1981]
- “Conjunto de ordenadores independientes que se muestran al usuario como un sistema único coherente”. [Tanenbaum 2001]

Sin duda alguna, el modelo de comunicación que ha revolucionado los sistemas informáticos en red es el modelo cliente/servidor. El modelo cliente/servidor está compuesto por un servidor que ofrece una serie de servicios y unos clientes que acceden a dichos servicios a través de la red.

Por ejemplo, el servicio más importante de Internet, el WWW, utiliza el modelo cliente/servidor. Por un lado tenemos el servidor que alojan las páginas web y por otro lado los clientes que solicitan al servidor una determinada página web.



Modelo Cliente/Servidor

**En el mundo de las comunicaciones entre equipos el modelo de comunicación** más utilizado es el modelo Cliente/servidor ya que ofrece una gran flexibilidad, interoperabilidad y estabilidad para acceder a recursos de forma centralizada.

El término modelo Cliente/Servidor se acuñó por primera vez en los años 80 para explicar un sencillo paradigma: un equipo cliente requiere un servicio de un equipo

servidor.

Desde el punto de vista funcional, se puede definir el modelo Cliente/Servidor como una arquitectura distribuida que permite a los usuarios finales obtener acceso a recursos de forma transparente en entornos multiplataforma.

Normalmente, los recursos que suele ofrecer el servidor son datos, pero también puede permitir acceso a dispositivos hardware, tiempo de procesamiento, etc.

Los elementos que componen el modelo son:

- **Cliente.** Es el proceso que permite interactuar con el usuario, realizar las peticiones, enviarlas al servidor y mostrar los datos al cliente. En definitiva, se comporta como la interfaz (front-end) que utiliza el usuario para interactuar con el servidor. Las funciones que lleva a cabo el proceso cliente se resumen en los siguientes puntos:
  - Interactuar con el usuario.
  - Procesar las peticiones para ver si son válidas y evitar peticiones maliciosas al servidor.
  - Recibir los resultados del servidor.
  - Formatear y mostrar los resultados.
- **Servidor.** Es el proceso encargado de recibir y procesar las peticiones de los clientes para permitir el acceso a algún recurso (back-end). Las funciones del servidor son:
  - Aceptar las peticiones de los clientes.
  - Procesar las peticiones.
  - Formatear y enviar el resultado a los clientes.
  - Procesar la lógica de la aplicación y realizar validaciones de datos.
  - Asegurar la consistencia de la información.
  - Evitar que las peticiones de los clientes interfieran entre sí.
  - Mantener la seguridad del sistema.

La idea es tratar el servidor como una entidad que realiza un determinado conjunto de tareas y que las ofrece como servicio a los clientes.

La forma más habitual de utilizar el modelo cliente/servidor es mediante la utilización de equipos a través de interfaces gráficas; mientras que la administración de datos y su seguridad e integridad se deja a cargo del servidor. Normalmente, el trabajo pesado lo realiza el servidor y los procesos clientes sólo se encargan de interactuar con el usuario. En otras palabras, el modelo Cliente/Servidor es una extensión de programación modular en la que se divide la funcionalidad del software en dos módulos con el fin de hacer más fácil el desarrollo y mejorar su mantenimiento

## 4.1. Características básicas.

Las características básicas de una arquitectura Cliente/Servidor son:

- Combinación de un cliente que interactúa con el usuario, y un servidor que interactúa con los recursos compartidos. El proceso del cliente proporciona la interfaz de usuario y el proceso del servidor permite el acceso al recurso compartido.
- Las tareas del cliente y del servidor tienen diferentes requerimientos en cuanto al procesamiento; todo el trabajo de procesamiento lo realiza el servidor y mientras que el cliente interactúa con el usuario.
- Se establece una relación entre distintos procesos, las cuales se pueden ejecutar en uno o varios equipos distribuidos a lo largo de la red.
- Existe una clara distinción de funciones basada en el concepto de "servicio", que se establece entre clientes y servidores.
- La relación establecida puede ser de muchos a uno, en la que un servidor puede dar servicio a muchos clientes, regulando el acceso a los recursos compartidos.
- Los clientes corresponden a procesos activos ya que realizan las peticiones de servicios a los servidores. Estos últimos tienen un carácter pasivo ya que esperan las peticiones de los clientes.
- Las comunicaciones se realizan estrictamente a través del intercambio de mensajes.
- Los clientes pueden utilizar sistemas heterogéneos ya que permite conectar clientes y servidores independientemente de sus plataformas.

## 4.2. Ventajas y desventajas.

Entre las principales ventajas del esquema Cliente/Servidor destacan:

- Utilización de clientes ligeros (con pocos requisitos hardware) ya que el servidor es quien realmente realiza todo el procesamiento de la información.
- Facilita la integración entre sistemas diferentes y comparte información permitiendo interfaces amigables al usuario.
- Se favorece la utilización de interfaces gráficas interactivas para los clientes para interactuar con el servidor. El uso de interfaces gráficas en el modelo Cliente/Servidor presenta la ventaja, con respecto a un sistema centralizado, de que normalmente sólo transmite los datos por lo que se aprovecha mejor el ancho de banda de la red.

- El mantenimiento y desarrollo de aplicaciones resulta rápido utilizando las herramientas existentes.
- La estructura inherentemente modular facilita además la integración de nuevas tecnologías y el crecimiento de la infraestructura computacional, favoreciendo así la escalabilidad de las soluciones.
- Contribuye a proporcionar a los diferentes departamentos de una organización, soluciones locales, pero permitiendo la integración de la información relevante a nivel global.
- El acceso a los recursos se encuentra centralizado.
- Los clientes acceden de forma simultánea a los datos compartiendo información entre sí.

Entre las principales desventajas del esquema Cliente/Servidor destacan:

- Mantenimiento de sistemas más difícil pues implica la interacción de diferentes partes de hardware y de software lo cual dificulta el diagnóstico de errores.
- Hay que tener estrategias para el manejo de errores del sistema.
- Es importante mantener la seguridad del sistema.
- Hay que garantizar la consistencia de la información. Como es posible que varios clientes operen con los mismos datos de forma simultánea, es necesario utilizar mecanismos de sincronización para evitar que un cliente modifique datos sin que lo sepan los demás clientes.

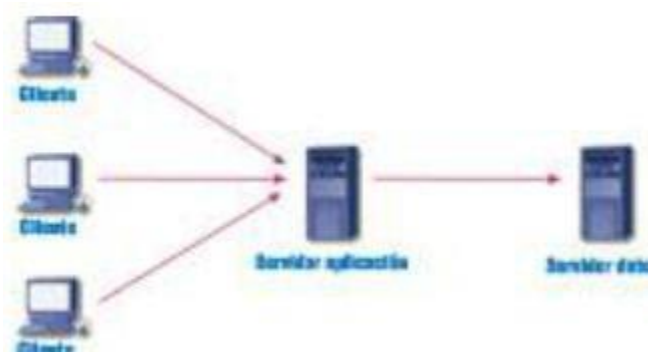
### 4.3. Modelos.

La principal forma de clasificar los modelos Cliente/Servidor es a partir del número de capas (tiers) que tiene la infraestructura del sistema. De ésta forma podemos tener los siguientes modelos:

- **1 capa (1-tier).** El proceso cliente/servidor se encuentra en el mismo equipo y realmente no se considera un modelo cliente/servidor ya que no se realizan comunicaciones por la red.
- **2 capas (2-tiers).** Es el modelo tradicional en el que existe un servidor y unos clientes bien diferenciados. El principal problema de éste modelo es que no permite escalabilidad del sistema y puede sobrecargarse con un número alto de peticiones por parte de los clientes.



- **3 capas (3-tiers).** Para mejorar el rendimiento del sistema en el modelo de dos capas se añade una nueva capa de servidores. En este caso se dispone de:
  - **Servidor de aplicación.** Es el encargado de interactuar con los diferentes clientes y enviar las peticiones de procesamiento al servidor de datos.
  - **Servidor de datos.** Recibe las peticiones del servidor de aplicación, las procesa y le devuelve su resultado al servidor de aplicación para que éste los envíe al cliente. Para mejorar el rendimiento del sistema, es posible añadir los servidores de datos que sean necesarios.



- **n capas (n-tiers).** A partir del modelo anterior, se pueden añadir capas adicionales de servidores con el objetivo de separar la funcionalidad de cada servidor y de mejorar el rendimiento del sistema.



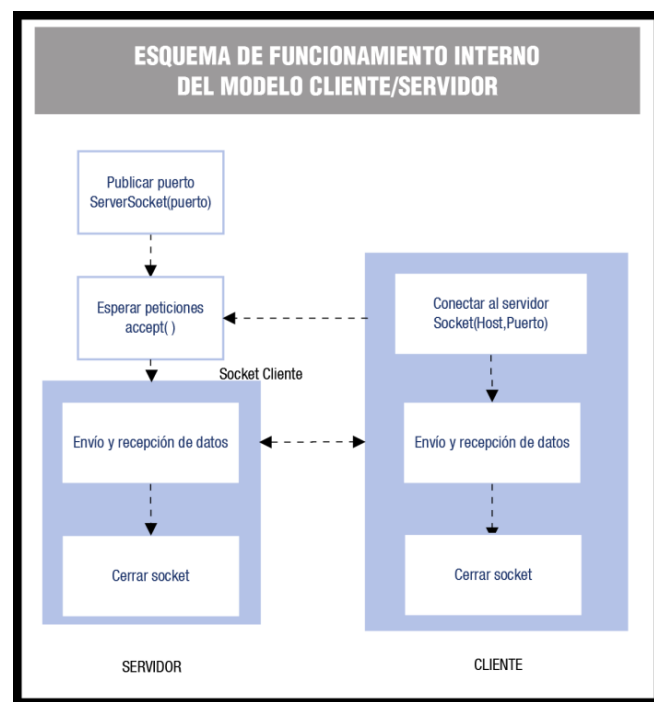
## 4.4. Programación

De forma interna, los pasos que realiza el servidor para realizar una comunicación son:

- **Publicar puerto.** Publica el puerto por donde se van a recibir las conexiones.
- **Esperar peticiones.** En este momento el servidor queda a la espera a que se conecte un cliente. Una vez que se conecte un cliente se crea el socket del cliente por donde se envían y reciben los datos.
- **Envío y recepción de datos.** Para poder recibir/enviar datos es necesario crear un flujo (stream) de entrada y otro de salida. Cuando el servidor recibe una petición, éste la procesa y le envía el resultado al cliente.
- Una vez finalizada la comunicación **se cierra el socket del cliente.**

Los pasos que realiza el cliente para realizar una comunicación son:

- **Conectarse con el servidor.** El cliente se conecta con un determinado servidor a un puerto específico. Una vez realizada la conexión se crea el socket por donde se realizará la comunicación.
- **Envío y recepción de datos.** Para poder recibir/enviar datos es necesario crear un flujo (stream) de entrada y otro de salida.
- Una vez finalizada la comunicación se **cierra el socket.**



## 5. Optimización de sockets

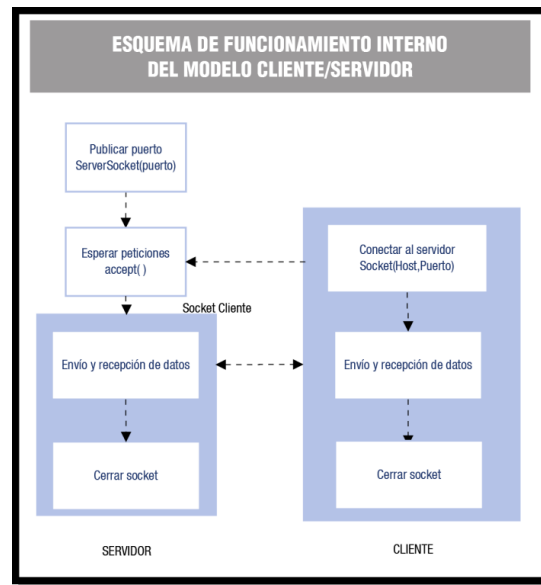
A la hora de utilizar los sockets es muy importante optimizar su funcionamiento y garantizar la seguridad del sistema. Como la información reside en el servidor y existen múltiples clientes que realizan peticiones es totalmente indispensable permitir que la aplicación cliente/servidor cuente con las siguientes características que veremos más adelante:

- **Atender múltiples peticiones simultáneamente.** El servidor debe permitir el acceso de forma simultánea al servidor para acceder a los recursos o servicios que éste ofrece.
- **Seguridad.** Para asegurar el sistema, como mínimo, el servidor debe ser capaz de evitar la pérdida de información, filtrar las peticiones de los clientes para asegurar que éstas están bien formadas y llevar un control sobre las diferentes transacciones de los clientes.
- Por último, es necesario dotar a nuestro sistema de mecanismos para **monitorizar los tiempos de respuesta** de los clientes para ver el comportamiento del sistema.

### 5.1. Atender múltiples peticiones simultáneamente

Si observamos la siguiente figura, cuando un servidor recibe la conexión del cliente (accept) se crea el socket del cliente, se realiza el envío y recepción de datos y se cierra el socket del cliente finalizando la ejecución del servidor.

Como el objetivo es permitir que múltiples clientes utilicen el servidor de forma simultánea es necesario que la parte que atiende al cliente (zona coloreada de azul) se atienda de forma independiente para cada uno de los clientes.



Para ello, en vez de ejecutar todo el código del servidor de forma secuencial, vamos a tener un bucle while para que cada vez que se realice la conexión de un cliente se cree una hebra de ejecución (thread) que será la encargada de atender al cliente. De ésta forma, tendremos tantas hebras de ejecución como clientes se conecten a nuestro servidor de forma simultánea.

De forma resumida, el código necesario es:

```

while(true){

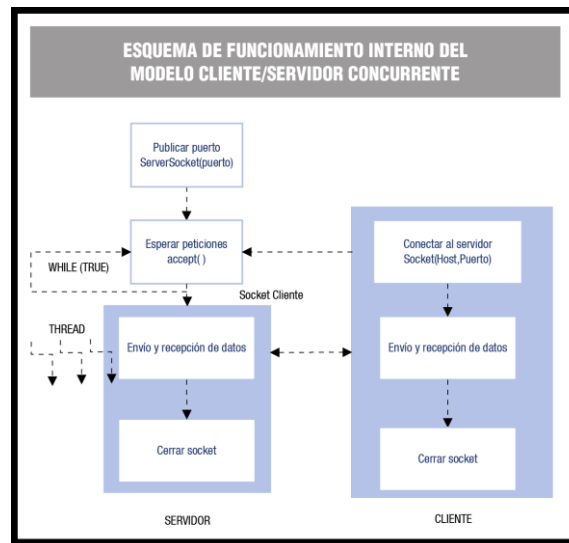
    // Se conecta un cliente

    Socket skCliente = skServidor.accept();
    System.out.println("Cliente conectado");

    // Atiendo al cliente mediante un thread
    new Servidor(skCliente).start();

}
  
```

Y a continuación puede ver su representación de forma gráfica:



## 5.2. Threads

Para crear una `haya` que defina la clase que extienda de `Threads`:

```

class Servidor extends Thread{

    public Servidor() {

        // Inicialización de la hebra
    }

    public static void main( String[] arg ) {
        new Servidor().start();
    }

    public void run(){

        //tarefas que realiza la hebra
    }
}

```

donde:

- La función `public Servidor` permite inicializar los valores iniciales que recibe la hebra.
- La función `run()` es la encargada de realizar las tareas de la hebra.
- Para iniciar la hebra se crea el objeto `Servidor` y se inicia:  
`new Servidor().start ();`

### 5.3. Ejemplo

Si añade al código anterior la utilización de sockets, tal y como se ha visto anteriormente, por parte del servidor obtiene un servidor que permite atender múltiples peticiones de forma concurrente:

#### **Servidor.java**

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

class Servidor extends Thread{
    Socket skCliente;

    static final int Puerto=2000;

    public Servidor(Socket sCliente) {
        skCliente=sCliente;
    }

    public static void main( String[] arg ) {
        try{

            // Inicio el servidor en el puerto

            ServerSocket skServidor = new ServerSocket(Puerto);

            System.out.println("Escucho el puerto " + Puerto );

            while(true){

                // Se conecta un cliente
                Socket skCliente = skServidor.accept();
                System.out.println("Cliente conectado");
```

## CFGs Desarrollo de Aplicaciones Multiplataforma Programación de Procesos y Servicios

```
// Atiendo al cliente mediante un thread
new Servidor(skCliente).start();

    }

    } catch (Exception e) {}

}

public void run(){

    try {

        // Creo los flujos de entrada y salida

        DataInputStream flujo_entrada = new
        DataInputStream(skCliente.getInputStream());

        DataOutputStream flujo_salida= new
        DataOutputStream(skCliente.getOutputStream());

        // ATENDER PETICIÓN DEL CLIENTE

        flujo_salida.writeUTF("Se ha conectado el cliente de forma correcta");

        // Se cierra la conexión
        skCliente.close();
        System.out.println("Cliente desconectado");

        } catch( Exception e ) {
            System.out.println( e.getMessage() );
        }

    }

}
```

Lógicamente, el funcionamiento del cliente no cambia ya que la concurrencia la realiza el servidor. A continuación puede ver un ejemplo básico de un cliente.

## Cliente.java

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

class Cliente {
```

```
static final String HOST = "localhost";
static final int Puerto=2000;

public Cliente( ) {
    try{

        Socket sCliente = new Socket( HOST , Puerto );

        // Creo los flujos de entrada y salida
        DataInputStream flujo_entrada = new
DataInputStream(sCliente.getInputStream());

        DataOutputStream flujo_salida= new
DataOutputStream(sCliente.getOutputStream());

        // TAREAS QUE REALIZA EL CLIENTE

        String datos=flujo_entrada.readUTF();

        System.out.println(datos);

        sCliente.close();

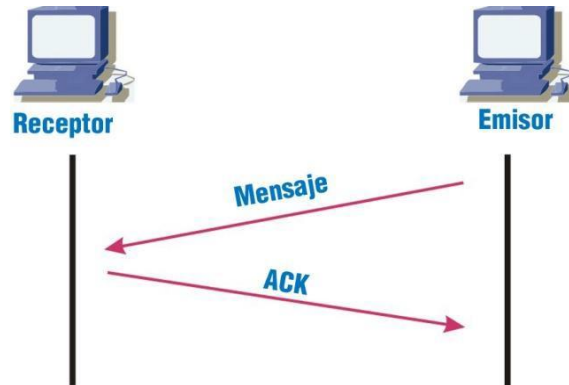
    } catch( Exception e ) {
        System.out.println( e.getMessage() );
    }
}

public static void main( String[] arg ) {
    new Cliente();
}
}
```

## 5.4. Pérdida de información

La pérdida de paquetes en las comunicaciones de red es un factor muy importante que hay que tener en cuenta ya que, por ejemplo, si se envía un fichero la pérdida de un único paquete produce que el fichero no se reciba correctamente.

Para evitar la pérdida de paquetes en las comunicaciones, cada vez que se envía un paquete el receptor envía al emisor un paquete de confirmación ACK (acknowledgement).

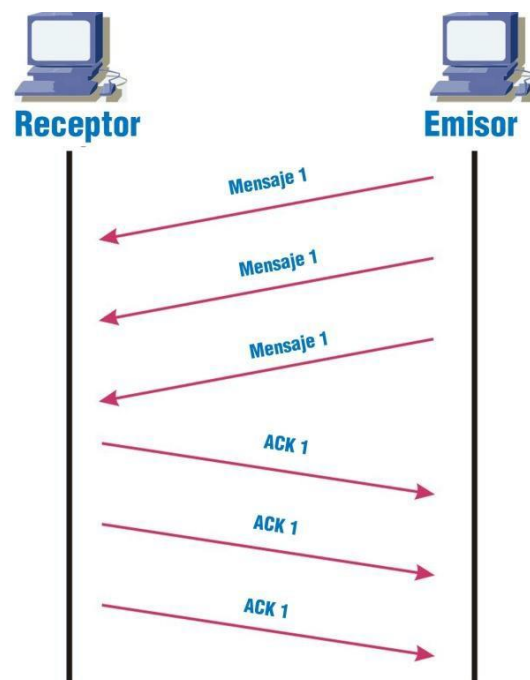


En el caso que el mensaje no llegue correctamente al receptor el paquete de confirmación no se envía nunca. El emisor cuando transcurre un determinado tiempo considera que el paquete se ha producido un error y vuelve a enviar el paquete.

Este método, aunque efectivo, resulta bastante lento ya que para enviar un nuevo paquete debe esperar el ACK del paquete anterior por lo que se produce un retardo en las comunicaciones.

Una mejora importante del método anterior consiste en permitir al emisor que envíe múltiples paquetes de forma sin necesidad de esperar los paquetes de confirmación. De esta forma el emisor puede enviar N paquetes de forma simultánea y así mejorar las comunicaciones.





Como una de las características de las redes es que es posible que los paquetes no lleguen ordenados, ni los paquetes ACK lleguen ordenados o, simplemente que se pierda algún mensaje en el camino, es necesario llevar un control sobre los paquetes enviados y los confirmados.

Para llevar un control de los paquetes enviados se utiliza un vector en el que se indica si un determinado paquete se ha enviado correctamente o no. Lógicamente, como los paquetes pueden llegar de forma desordenada, perderse paquetes,... es posible encontrar en el vector de configuración múltiples combinaciones como la siguiente:

Vector de ACK (estado inicial)

**Mensaje** 0 1 2 3 4 5 6 7 8 9

**ACK** 1 1 0 0 1 1 0 0 0 0

Como puede ver en el ejemplo anterior, los mensajes 0, 1, 4, y 5 han llegado correctamente. Por lo tanto para poder retransmitir más mensajes se desplaza el vector de derecha a izquierda con los mensajes enviados correctamente hasta llegar al primer mensaje no enviado correctamente (en el ejemplo el 2). De esta forma siguiendo el ejemplo propuesto el vector queda de la siguiente forma:

Vector de ACK (desplazado)  
**Mensaje** 2 3 4 5 6 7 8 9 10 11  
**ACK**      0 0 1 1 0 0 0 0 0 0

Ahora el sistema ya puede enviar los mensajes 10 y 11 mientras que espera la confirmación de los demás mensajes.

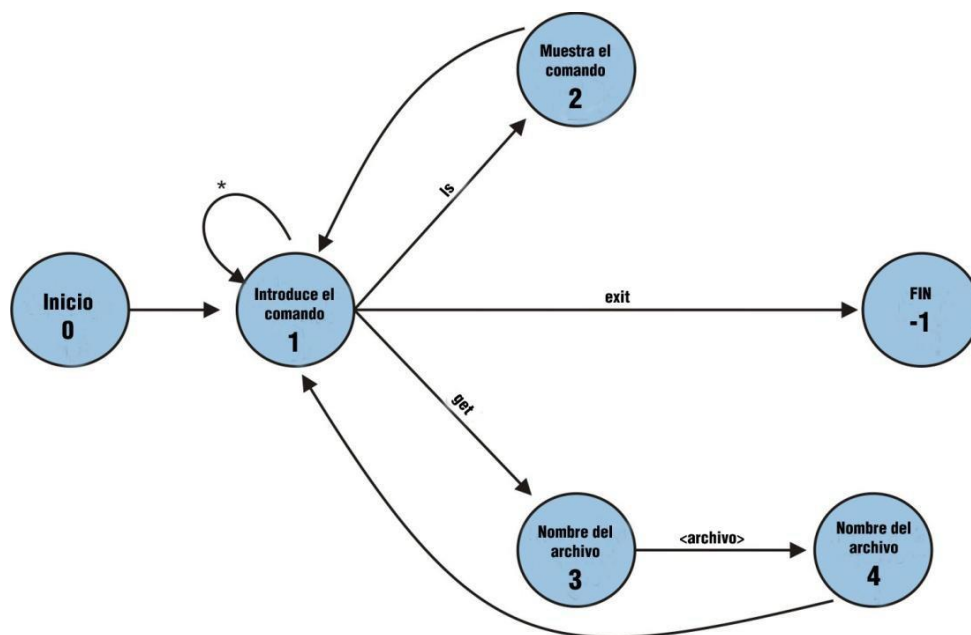
Como ha podido observar, el tamaño del vector influye muy estrechamente en el rendimiento del sistema ya que cuando mayor sea el vector más mensajes se pueden enviar de forma concurrente. Lógicamente, existe la limitación de la memoria RAM que ocupa el vector.

## 5.5. Transacciones

Uno de los principales fallos de seguridad que se producen en la programas clientes/servidor es que el cliente pueda realizar:

- **Operaciones no autorizadas.** El servidor no puede procesar una orden a la que el cliente no tiene acceso. Por ejemplo, que el cliente realice una solicitud de información a la que no tiene acceso.
- **Mensajes mal formados.** Es posible que el cliente envíe al servidor mensajes mal formados o con información incompleta que produzca un error de procesamiento del sistema llegando incluso a dejar "colgado" el servidor.

Para evitar cualquier problema de seguridad es muy importante modelar el flujo de información y el comportamiento del servidor con un **diagrama de estados o autómatas**. Por ejemplo, en la siguiente figura



puede ver que el servidor se inicia en el estado 0 y directamente envía al cliente el mensaje Introduce el comando. El cliente puede enviar los comandos:

- ls que va al estado 2 mostrando el contenido del directorio y vuelve automáticamente al estado 1.
- get que le lleva al estado 3 donde le solicita al cliente el nombre del archivo a mostrar. Al introducir el nombre del archivo se desplaza al estado 4 donde muestra el contenido del archivo y vuelve automáticamente al estado 1.
- exit que le lleva directamente al estado donde finaliza la conexión del cliente (estado -1).
- Cualquier otro comando hace que vuelva al estado 1 solicitándole al cliente que introduzca un comando válido.

Para poder seguir el comportamiento del autómata el servidor tiene que definir dos variables estado y comando. La variable estado almacena la posición en la que se encuentra y la variable comando es el comando que recibe el servidor y el que permite la transición de un estado a otro. Cuando se utilizan autómatas muy sencillos como es el caso del ejemplo, es posible modelar el comportamiento del autómata utilizando estructuras case e if. En el caso de utilizar autómatas grandes la mejor forma de modelar su comportamiento es mediante una tabla cuyas filas son los diferentes estados del autómata y la columna las diferentes entradas del sistema.

## 5.6. Monitorizar tiempos de respuesta

Un aspecto muy importante para ver el comportamiento de nuestra aplicación Cliente/Servidor son los tiempos de respuesta del servidor. Desde que el cliente realiza una petición hasta que recibe su resultado intervienen dos tiempos:

- **Tiempo de procesamiento.** Es el tiempo que el servidor necesita para procesar la petición del cliente y enviar los datos.
- **Tiempo de transmisión.** Es el tiempo que transcurre para que los mensajes viajen a través de los diferentes dispositivos de la red hasta llegar a su destino.

Para medir el tiempo de procesamiento tan sólo se necesita medir el tiempo que transcurre en que el servidor procese la solicitud del cliente. Para medir el tiempo en milisegundos necesario para procesar la petición de un cliente puede utilizar el siguiente código:

```
import java.util.Date;

long tiempo1=(new Date()).getTime();

// Procesar la petición del cliente

long tiempo2=(new Date()).getTime();

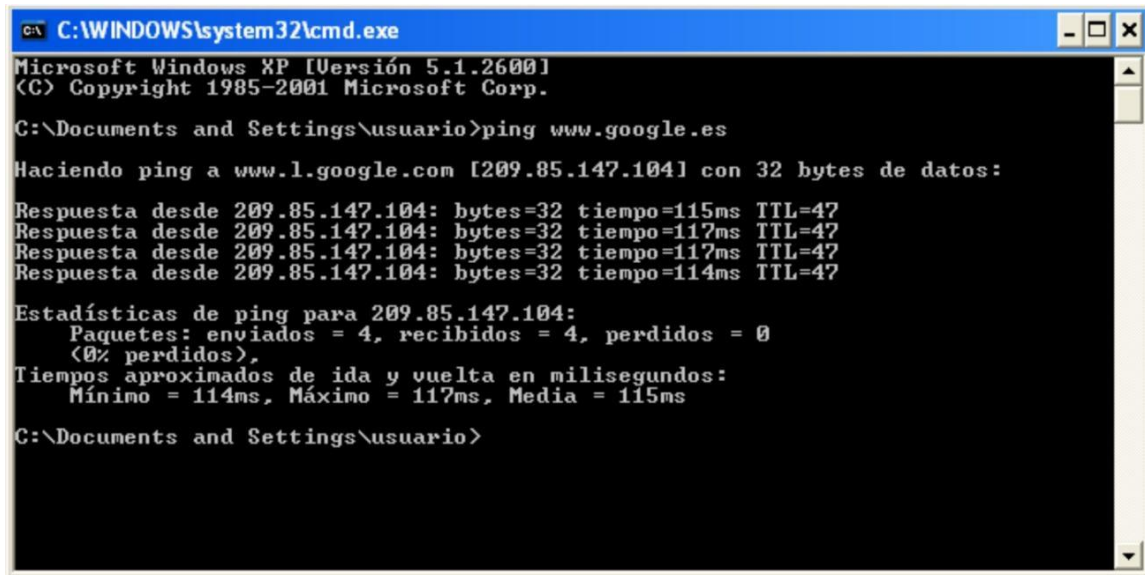
System.out.println("\t Tiempo = "+(tiempo2-tiempo1)+" ms");
```

Para medir el tiempo de transmisión es necesario enviar a través de un mensaje el tiempo del sistema y el receptor comparar su tiempo de respuesta con el que hay dentro del mensaje. Lógicamente, para poder comparar los tiempos de respuesta de dos equipos es totalmente necesario que los relojes del sistema estén sincronizados a través de cualquier servicio de tiempo (**NTP**). En equipos Windows la sincronización de los relojes se realiza automáticamente y en equipos GNU/Linux se realiza ejecutando el siguiente comando:

```
/usr/sbin/ntpdate -u 0.centos.pool.ntp.org
```

Otra forma de calcular el tiempo de transmisión es utilizar el comando **ping**. Como puede ver en la siguiente figura el tiempo para que el cliente origen acceda al

servidor [www.google.es](http://www.google.es) es de unos 115 milisegundos de media.



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Versión 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\usuario>ping www.google.es

Haciendo ping a www.l.google.com [209.85.147.104] con 32 bytes de datos:

Respuesta desde 209.85.147.104: bytes=32 tiempo=115ms TTL=47
Respuesta desde 209.85.147.104: bytes=32 tiempo=117ms TTL=47
Respuesta desde 209.85.147.104: bytes=32 tiempo=117ms TTL=47
Respuesta desde 209.85.147.104: bytes=32 tiempo=114ms TTL=47

Estadísticas de ping para 209.85.147.104:
    Paquetes: enviados = 4, recibidos = 4, perdidos = 0
    (0% perdidos),
    Tiempos aproximados de ida y vuelta en milisegundos:
        Mínimo = 114ms, Máximo = 117ms, Media = 115ms

C:\Documents and Settings\usuario>
```

A continuación vamos a ver un ejemplo en el que se calcula el tiempo de transmisión de datos entre una aplicación Cliente y Servidor. Para ello, el servidor le va a enviar al cliente un mensaje con el tiempo del sistema en milisegundos y el cliente cuando reciba el mensaje calculará la diferencia entre el tiempo de su sistema y el del mensaje.

## Servidor.java

```
import java.io.* ;
import java.net.* ;
import java.util.Date;
class Servidor {
    static final int Puerto=2000;

    public Servidor( ) {

        try {

            // Inicio el servidor en el puerto
            ServerSocket sServidor = new ServerSocket(Puerto);
            System.out.println("Escucho el puerto " + Puerto );

            // Se conecta un cliente
            Socket sCliente = sServidor.accept(); // Crea objeto
            System.out.println("Cliente conectado");

            // Creo los flujos de entrada y salida
            DataInputStream flujo_entrada = new DataInputStream(
sCliente.getInputStream());
            DataOutputStream flujo_salida= new
DataOutputStream(sCliente.getOutputStream());

            // CUERPO DEL ALGORITMO
            long tiempo1=(new Date()).getTime();
            flujo_salida.writeUTF(Long.toString(tiempo1));

            // Se cierra la conexión
            sCliente.close();
            System.out.println("Cliente desconectado");

            } catch( Exception e ) {
                System.out.println( e.getMessage() );
            }
        }

    public static void main( String[] arg ) {
        new Servidor();
    }
}
```

## Cliente.java

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

class Cliente {
    static final String HOST = "localhost";
    static final int Puerto=2000;

    public Cliente( ) {
        String datos=new String(); String
        num_cliente=new String();

        // para leer del teclado
        BufferedReader reader=new BufferedReader(new
        InputStreamReader(System.in));

        try{
            // Me conecto al puerto
            Socket sCliente = new Socket( HOST , Puerto );

            // Creo los flujos de entrada y salida DataInputStream
            flujo_entrada = new
DataInputStream(sCliente.getInputStream());
            DataOutputStream flujo_salida= new
DataOutputStream(sCliente.getOutputStream());

            // CUERPO DEL ALGORITMO
            datos=flujo_entrada.readUTF();
            long tiempo1=Long.valueOf(datos);
            long tiempo2=(new Date()).getTime(); System.out.println("\n El
            tiempo es:"+(tiempo2-tiempo1));

            // Se cierra la conexión sCliente.close();
        } catch( Exception e ) { System.out.println(
            e.getMessage() );
        }
    }

    public static void main( String[] arg ) { new
        Cliente();
    }
}
```