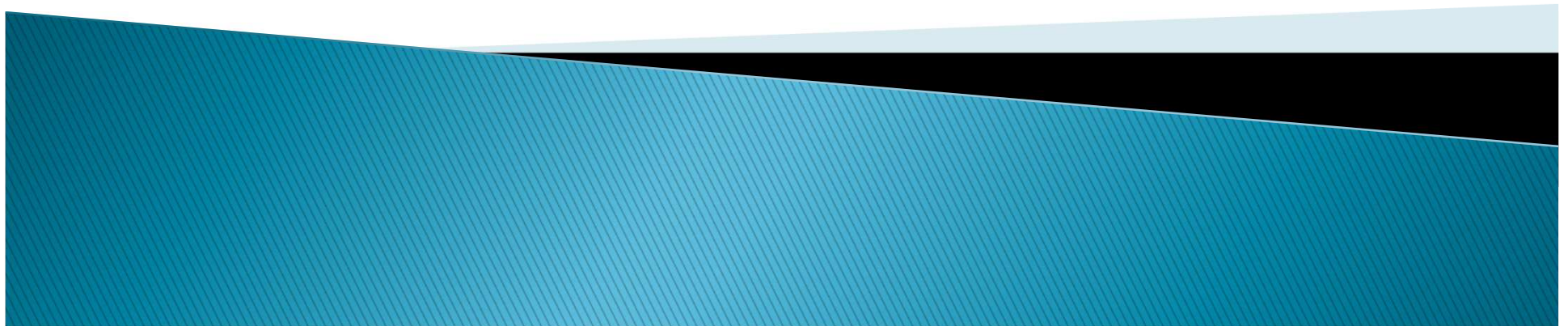


TEMA 3: PROGRAMACIÓN DE COMUNICACIONES EN RED

Programación de Servicios y Procesos

José Manuel García Sánchez



ÍNDICE

- ▶ Conceptos básicos: Comunicación entre aplicaciones
- ▶ Protocolos de comunicaciones: IP, TCP, UDP
- ▶ Sockets
- ▶ Modelos de Comunicaciones



Conceptos básicos

- ▶ Muchos sistemas computacionales de la actualidad siguen el modelo de **computación distribuida**.
- ▶ Aplicaciones a través de Internet, móviles, etc.
- ▶ La mayoría de superordenadores modernos son sistemas distribuidos.



Sistema distribuido

- ▶ Está formado por **más de un elemento computacional** distinto e independiente (un procesador dentro de una máquina, un ordenador dentro de una red, etc), que no comparte memoria con el resto.
- ▶ Los elementos que forman el sistema distribuido **no están sincronizados**: No hay reloj común.
- ▶ Los elementos que forman el sistema están **conectados a una red de comunicaciones**.

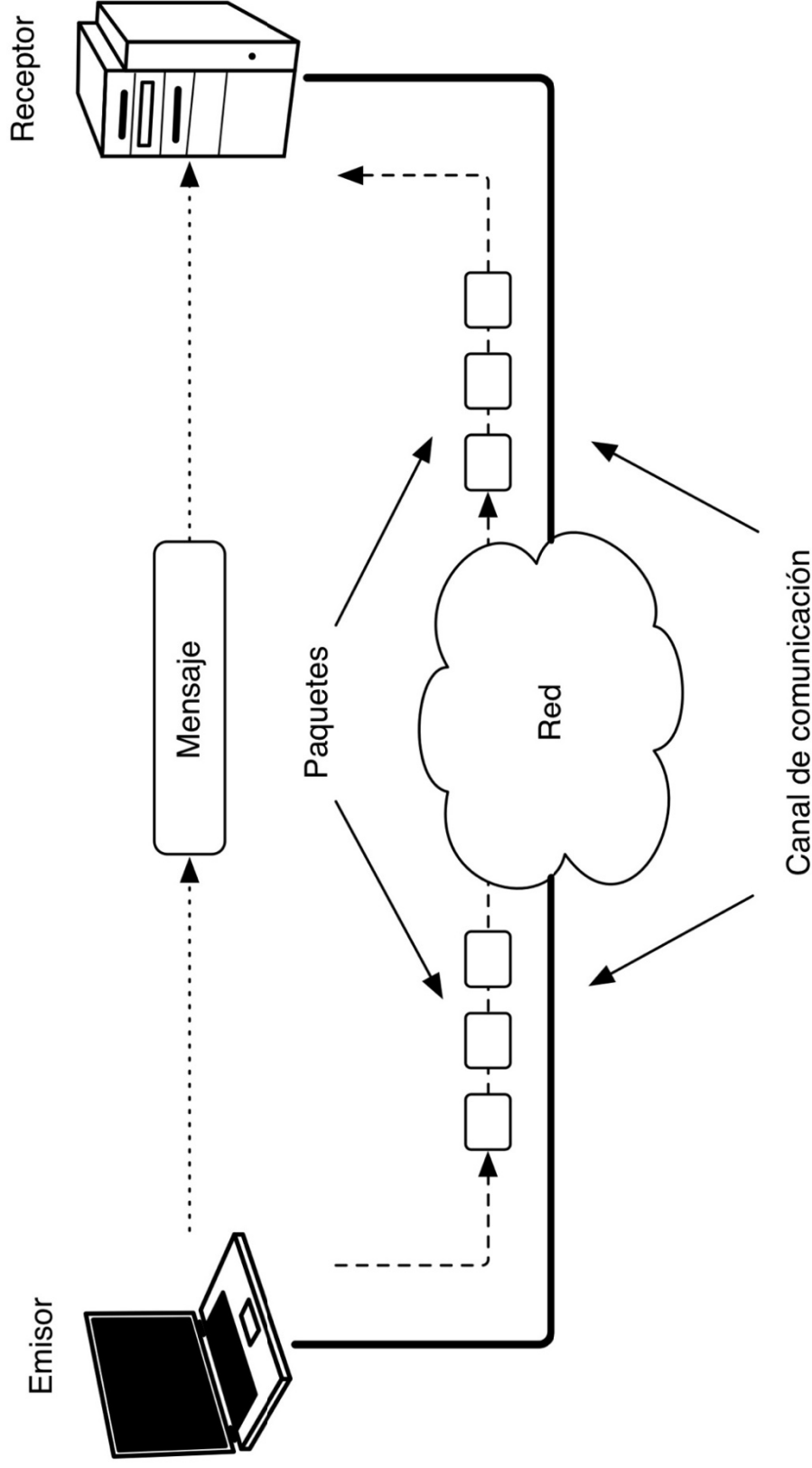


Comunicación entre aplicaciones

- ▶ Es la base del funcionamiento de todo sistema distribuido.
- ▶ En el proceso de comunicación se distingue:
 - Mensaje
 - Emisor
 - Receptor
 - Paquete
 - Canal de comunicación
 - Protocolo de comunicaciones

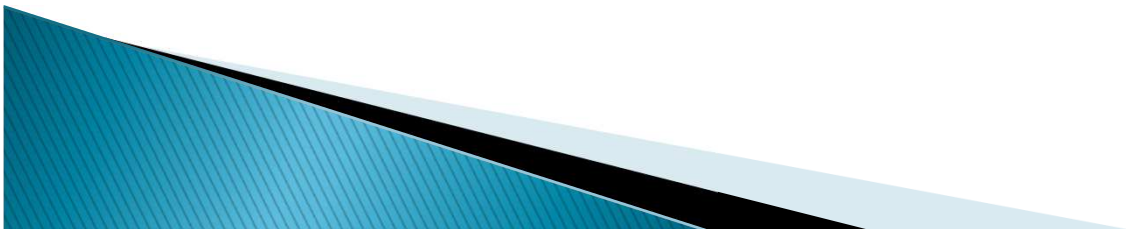


Comunicación entre aplicaciones



Comunicación entre aplicaciones

- ▶ Mensaje:
 - Es la información que se intercambia entre las aplicaciones que se comunican.
- ▶ Emisor:
 - Es la aplicación que envía el mensaje.
- ▶ Receptor:
 - Es la aplicación que recibe el mensaje.
- ▶ Paquete:
 - Es la unidad básica de información que intercambian dos dispositivos de comunicación.



Comunicación entre aplicaciones

- ▶ Canal de comunicación:
 - Es el medio por el que se transmiten los paquetes, que conecta el emisor con el receptor.
- ▶ Protocolo de comunicaciones:
 - Es el conjunto de reglas que fijan cómo se deben intercambiar paquetes entre los diferentes elementos que se comunican entre sí.

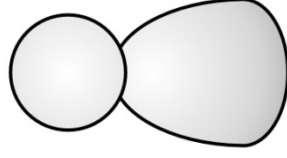


Protocolos de comunicaciones

- ▶ Para que las diferentes aplicaciones que forman un sistema distribuido puedan comunicarse, debe existir una serie de mecanismos que hagan posible esa comunicación:
 - Elementos hardware
 - Elemento software
- ▶ Todos estos componentes se organizan en lo que se denomina una **jerarquía** o **pila de protocolos**.



Pila de protocolos IP



Usuario

Nivel de aplicación

Nivel de transporte

Nivel de Internet

Nivel de red



Pila de protocolos IP

- ▶ Nivel de red:

- Lo componen los elementos hardware de comunicaciones y sus controladores básicos.
- Se encarga de transmitir los paquetes de información.

- ▶ Nivel de Internet:

- Lo componen los elementos software que se encargan de dirigir los paquetes por la red, asegurándose de que lleguen a su destino.
- También llamado **nivel IP**.



Pila de protocolos IP

▶ Nivel de transporte:

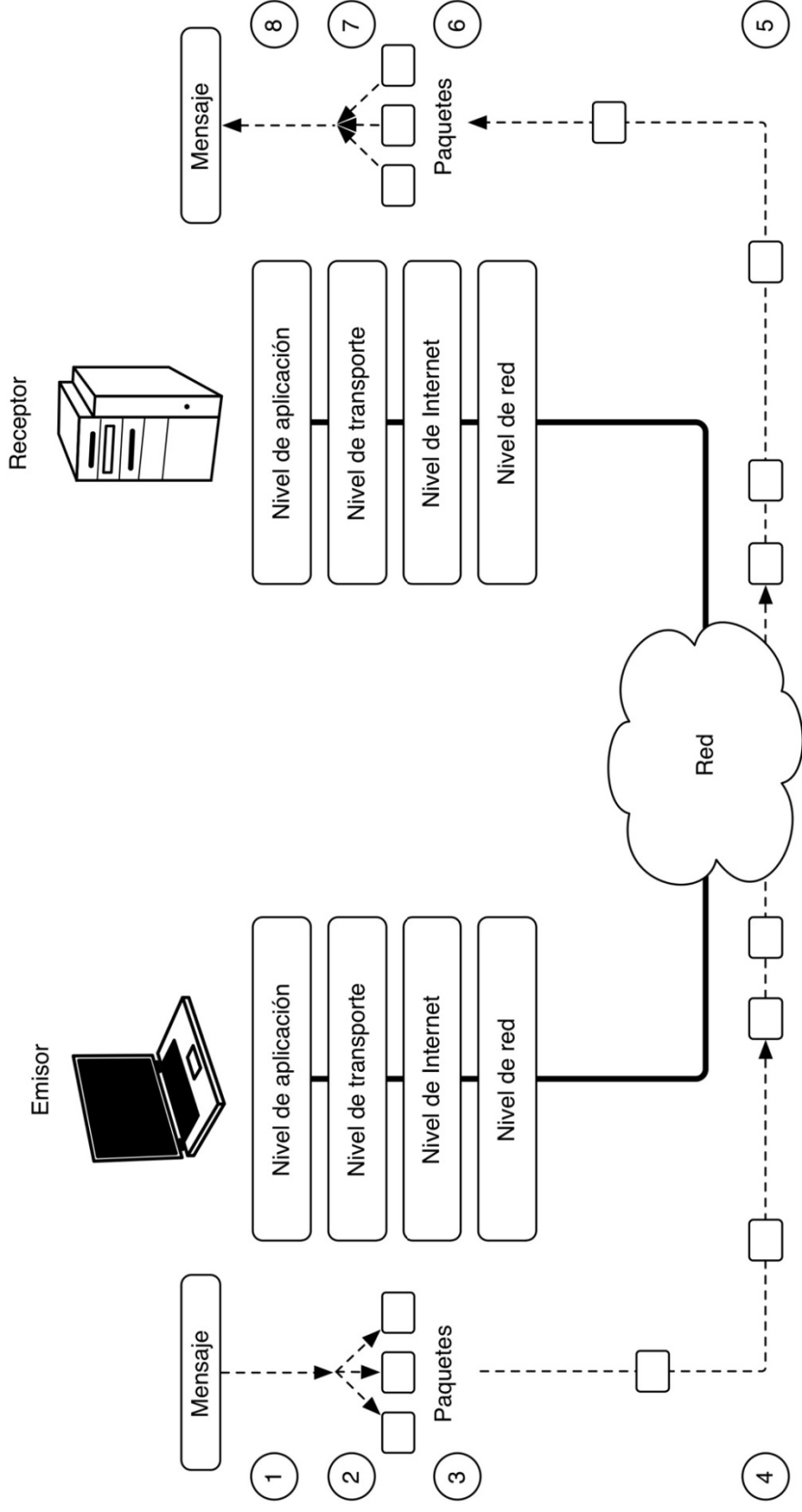
- Lo componen los elementos software cuya función es crear el canal de comunicación, descomponer el mensaje en paquetes y gestionar su transmisión entre el emisor y el receptor.
- Los dos protocolos de transporte fundamentales: TCP y UDP.

▶ Nivel de aplicación:

- Lo componen las aplicaciones que forman el sistema distribuido, que hacen uso de los niveles inferiores para poder transferir mensajes entre ellas



Funcionamiento de la pila de protocolos



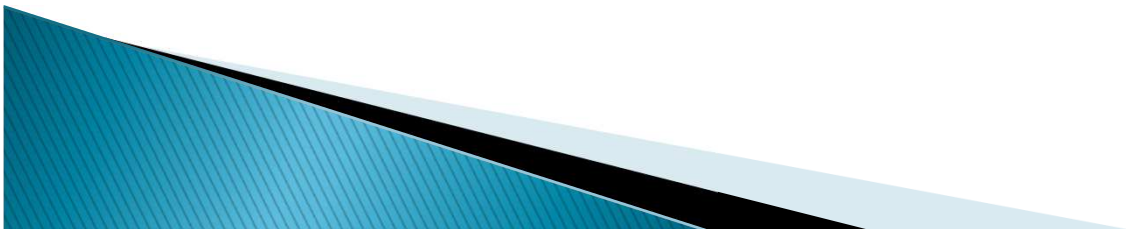
Protocolo de transporte TCP

- ▶ Garantiza que los datos no se pierden.
- ▶ Garantiza que los mensajes llegarán en orden.
- ▶ Se trata de un **protocolo orientado a conexión**.



Protocolo orientado a conexión

- ▶ Es aquel en que el canal de comunicaciones entre dos aplicaciones permanece abierto durante un cierto tiempo, permitiendo enviar múltiples mensajes de manera fiable por el mismo.
- ▶ Opera en tres fases:
 1. Establecimiento de la conexión.
 2. Envío de mensajes.
 3. Cierre de la conexión.
- ▶ El ejemplo mas habitual es el protocolo TCP.

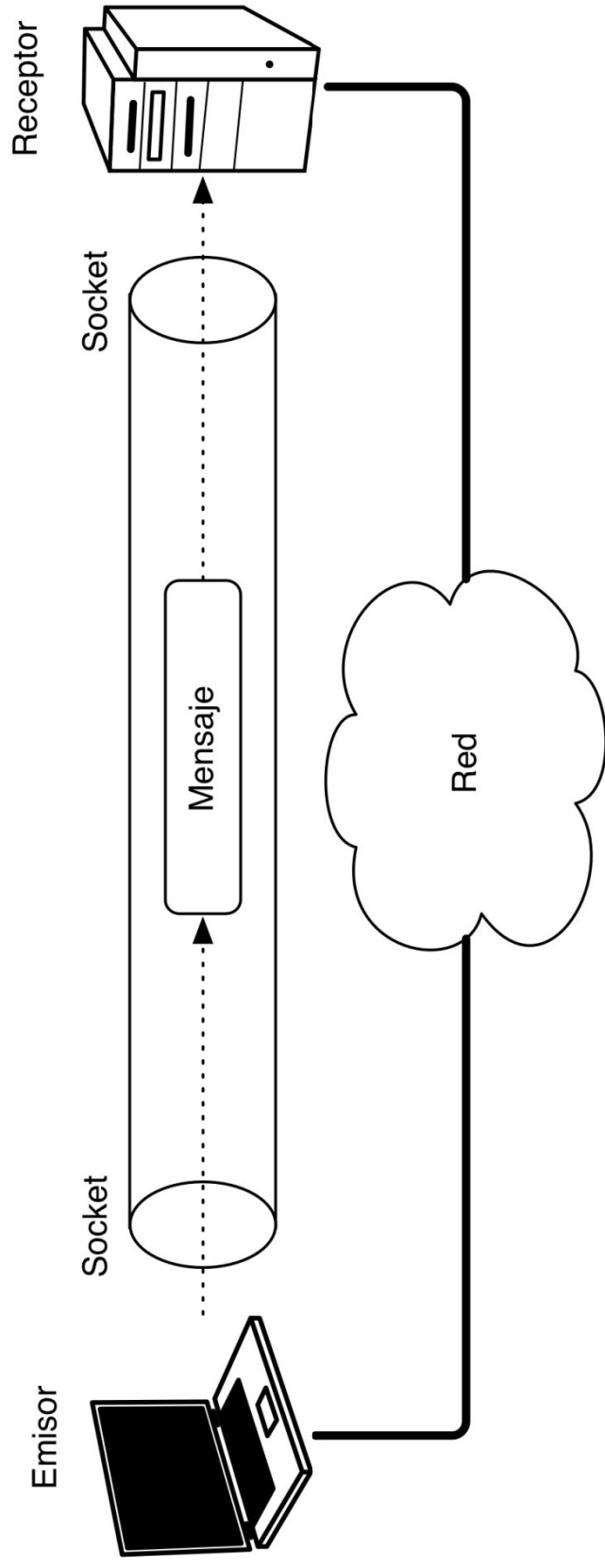


Sockets

- ▶ Los *sockets* son el mecanismo de comunicación básico fundamental que se usa para realizar transferencias de información entre aplicaciones.
- ▶ Proporcionan una abstracción de la pila de protocolos.
- ▶ Un *socket* (en inglés, literalmente, un “enchufe”) representa el extremo de un canal de comunicación establecido entre un emisor y un receptor.



Sockets



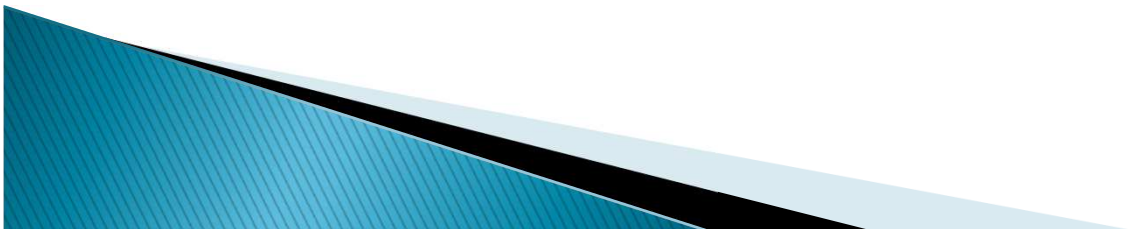
Direcciones IP y puertos

- ▶ Una dirección IP es un número que identifica de forma única a cada máquina de la red, y que sirve para comunicarse con ella.
- ▶ Un puerto es un número que identifica a un *socket* dentro de una máquina.



Sockets stream

- ▶ Son orientados a conexión.
- ▶ Cuando operan sobre IP, emplean TCP.
- ▶ Un ***socket stream*** se utiliza para comunicarse siempre con el mismo receptor, manteniendo el canal de comunicación abierto entre ambas partes hasta que se termina la conexión.
- ▶ Una parte ejerce la función de **proceso cliente** y otra de **proceso servidor**.



Sockets stream

- ▶ Proceso cliente:

1. **Creación** del socket.
2. Conexión del socket (***connect***).
3. **Envío y recepción** de mensajes.
4. Cierre de la conexión (***close***).



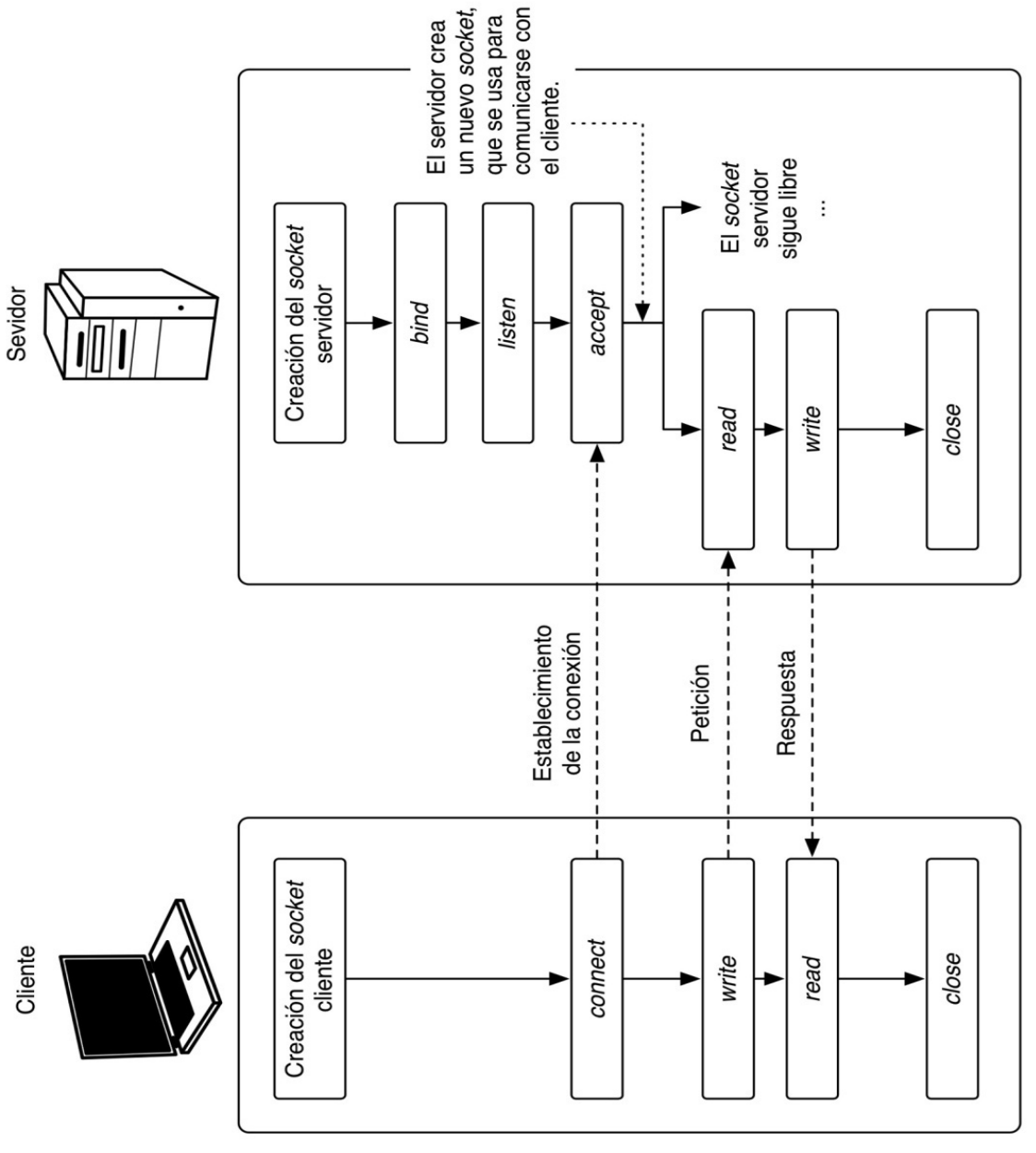
Sockets stream

► Proceso servidor:

1. **Creación** del socket.
2. Asignación de dirección y puerto (***bind***).
3. Escucha (***listen***).
4. Aceptación de conexiones (***accept***). Esta operación implica la **creación de un nuevo socket**, que se usa para comunicarse con el cliente que se ha conectado.
5. **Envío y recepción** de mensajes.
6. Cierre de la conexión (***close***).



Sockets stream



Programación con sockets

- ▶ ***java.net.Socket***,
 - para la creación de *sockets stream* cliente.
- ▶ ***java.net.ServerSocket***,
 - para la creación de *sockets stream* servidor.
- ▶ ***java.net.DatagramSocket***,
 - para la creación de *sockets datagram*.



Clase Socket

- ▶ Esta clase (`java.net.Socket`) se usa para crear y operar con sockets stream clientes.

Método	Tipo retorno	Descripción
<code>Socket()</code>	<code>Socket</code>	Constructor básico de la clase. Sirve para crear sockets stream clientes
<code>Connect(SocketAddress addr)</code>	<code>Void</code>	Establece la conexión con la dirección y puerto destino
<code>getInputStream()</code>	<code>InputStream</code>	Obtiene un objeto de clase <code>InputStream</code> que se usa para realizar operaciones de lectura (<code>read</code>)
<code>getOutputStream()</code>	<code>OutputStream</code>	Obtiene un objeto de clase <code>OutputStream</code> que se usa para realizar operaciones de escritura (<code>write</code>)
<code>Close()</code>	<code>void</code>	Cierra el socket

Sockets Stream (Cliente)

```
import java.io.*;
import java.net.*;
public class ClienteSocketStream {
    public static void main(String[] args) {
        try {
            Socket clientSocket = new Socket(); //Creando socket cliente
            //Estableciendo conexión
            InetSocketAddress addr = new InetSocketAddress("localhost", 5555);
            clientSocket.connect(addr);
            //enviando mensaje
            InputStream is = clientSocket.getInputStream();
            OutputStream os = clientSocket.getOutputStream();
            String mensaje = "mensaje desde el cliente";
            PrintWriter fsalida=new PrintWriter(os,true);
            fsalida.println(mensaje);
            //Mensaje eviado, ahora cierro el socket
            clientSocket.close();

            } catch (IOException e) { e.printStackTrace(); } } }
```



Clase ServerSocket

- ▶ La clase ServerSocket (java.net.ServerSocket) se usa para crear y operar con sockets stream servidor.

Método	Tipo retorno	Descripción
ServerSocket()	Socket	Constructor básico de la clase. Sirve para crear sockets stream servidores
ServerSocket(String host, int port)	Socket	Constructor alternativo de la clase. Se le pasan como argumento la dirección IP y el puerto que se desean asignar al socket. Este método realiza las operación de creación del socket y bind directamente
Bind(SocketAddress bindpoint)	void	Asigna al socket una dirección y número de puerto determinado (operación bind)
Accept()	Socket	Escucha por el socket servidor, esperando conexiones por parte de clientes (operación accept). Cuando llega una conexión devuelve un nuevo objeto de clase Socket, conectado al cliente
Close()	void	Cierra el socket

Gestión de Sockets TCP

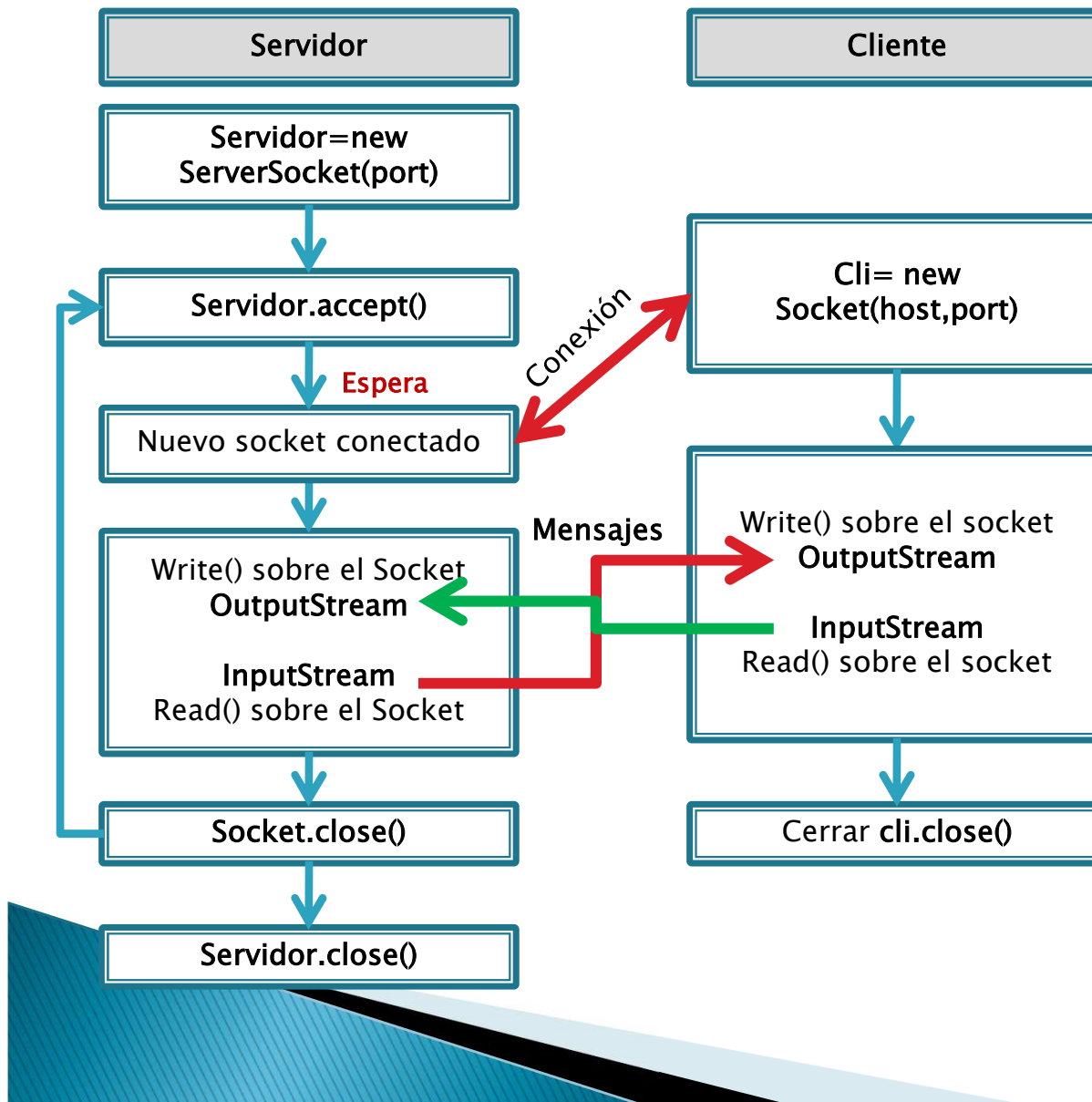
Socket Stream

El **servidor** crea un socket de servidor definiendo un puerto, mediante `ServerSocket(port)` y espera mediante el método `accept()` a que un cliente solicite conexión.

Cuando **el cliente** solicita conexión, **el servidor** abrirá la conexión al socket con el método `accept()`.

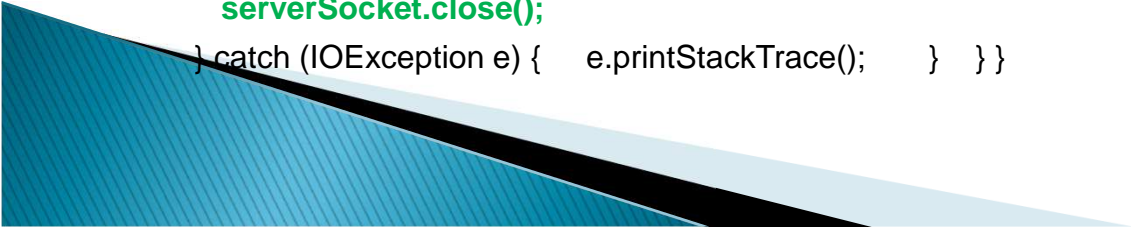
El **cliente** establece una conexión con el host a través del puerto especificado mediante el método **`Socket(host,port)`**.

Cliente y servidor se comunican mediante `InputStream` y `OutputStream`.



Sockets Stream (Server)

```
import java.io.*;
import java.net.*;
public class ServeridorSocketStream {
    public static void main(String[] args) {
        try {
            ServerSocket serverSocket = new ServerSocket(); //nuevo socket servidor
            //Realizando bind
            InetSocketAddress addr = new InetSocketAddress("localhost", 5555);
            serverSocket.bind(addr);
            //Aceptando conexiones
            Socket newSocket = serverSocket.accept();
            //Recibiendo mensaje
            InputStream is = newSocket.getInputStream();
            OutputStream os = newSocket.getOutputStream();
            BufferedReader fentrada=new BufferedReader(new InputStreamReader(is));
            String mensaje=fentrada.readLine();
            System.out.println( mensaje);
            //cerrando el nuevo socket
            newSocket.close();
            //cerrando el socket servidor
            serverSocket.close();
        } catch (IOException e) { e.printStackTrace(); } } }
```



Flujo de datos. Stream y Socket

- ▶ **PrintWriter (OutputStream out, boolean autoFlush)**
 - `PrintWriter(cliente.getOutputStream, true)`
 - Implementa el flujo de salida de caracteres sobre el socket y fuerza la escritura de datos con `Flush` a `true`.
- ▶ **BufferedReader(InputStreamReader in)**
 - `InputStreamReader in=new InputStreamReader(cliente.getInputStream());`
 - `BufferedReader(in);`
 - Implementa el flujo de entrada de caracteres sobre el socket.
- ▶ **Implementación**
 - `OutputStream out = cliente.getOutputStream();`
 - `PrintWriter fsalida=new PrintWriter(out,true);`
 - `fsalida.println(cadena); //envio de la cadena al servidor`

 - `InputStream in=cliente.getInputStream();`
 - `BufferedReader fentrada= new BufferedReader(new InputStreamReader(in);`
 - `String mensajeIn= fentrada.readLine(); //recibir la cadena del servidor`

Ejercicio:

Programa cliente que envía texto desde teclado al servidor y este lo lee y devuelve de nuevo al cliente el texto recibido.

El cliente lee del socket el mensaje recibido y lo muestra por pantalla.

El programa server finaliza cuando el cliente termina la entrada por teclado o cuando recibe como cadena un “*”.

El cliente finaliza cuando se detenga la entrada de datos mediante CTRL+C o CTRL+Z.



Documento de
icrosoft Office Wo



Ej. Cliente. Socket Stream

```
import java.io.IOException; import java.io.InputStream; import java.io.OutputStream;
import java.net.InetAddress; import java.net.InetSocketAddress; import java.net.Socket;
public class socketcliente {
    public static void main(String[] args) throws IOException {
        String Host ="localhost";
        Int Puerto = 6000; // Puerto remote
        Socket Cliente = new Socket (Host, Puerto);
        // Crear Flujo de Salida al Servidor
        PrintWriter fsalida = new PrintWriter ( Cliente.getOutputStream(), true);
        // Crear Flujo de Entrada al Servidor
        BufferedReader fentrada= new BufferedReader(new InputStreamReader(Cliente.getInputStream()));
        //Flujo para Entada estandar
        BufferedReader in=new BufferedReader (new InputStreamReader(System.in));
        String cadena, eco="";
        System.out.print("Introduce cadena; ");
        Cadena=in.readLine(); //lectura por teclado
        While(cadena!=null){
            fsalida.println(cadena); //envio de la cadena al servidor
            Eco= fentrada.readLine(); //recibir la cadena del servidor
            System.out.println(" =>ECO; " + eco);
            System.out.println("Introduce cadena; ");
            Cadena= in.readLine(); //lectura por teclado }
        fsalida.close();
        fentrada.close();
        System.out.println("Fin del envio...");
        in.close();
        Cliente.close();  }}
```


Ej. Servidor. Socket Stream

```
import java.io.IOException; import java.io.InputStream; import java.io.OutputStream;
import java.net.InetAddress; import java.net.InetSocketAddress; import java.net.Socket; import java.net.ServerSocket;
public class socketserver {
    public static void main(String[] args) throws IOException{
        int numeroPuerto=6000; //Puerto
        ServerSocket servidor=new ServerSocket(numeroPuerto);
        String cad="";
        System.out.println("Esperando conexión....");
        Socket clienteConectado=servidor.accept();
        System.out.println("Cliente conectado...");
        //Creando el flujo de salida al Cliente
        PrintWriter fsalida=new PrintWriter(clienteConectado.getOutputStream(),true);
        //Creando el flujo de entrada del cliente
        BufferedReader fentrada=new BufferedReader(new
        InputStreamReader(clienteConectado.getInputStream()));
        While (( cad=fentrada.readLine())!=null) { //recibida cadena del cliente
            fsalida.println(cad); //Envío de cadena a un cliente
            System.out.println (" Recibiendo: " + cad);
            if(cad.equals("*")) break;
        }
        //Cierre de Streams y Sockets
        System.out.println("Cerrando conexión....");
        fentrada.close();
        fsalida.close();
        clienteConectado.close();
        servidor.close(); }}
```


Ejercicio Sockets Stream. Hilos

- ▶ Escribir un programa Servidor que cree un hilo para cada conexión cliente a través del cual intercambia mensajes. Al cerrar la conexión, se cierra el hilo.

```
while (true) {  
    Socket nuevosocket = s.accept();  
    Thread t = new Hilonuevocliente(nuevosocket);  
    t.start();  
}
```

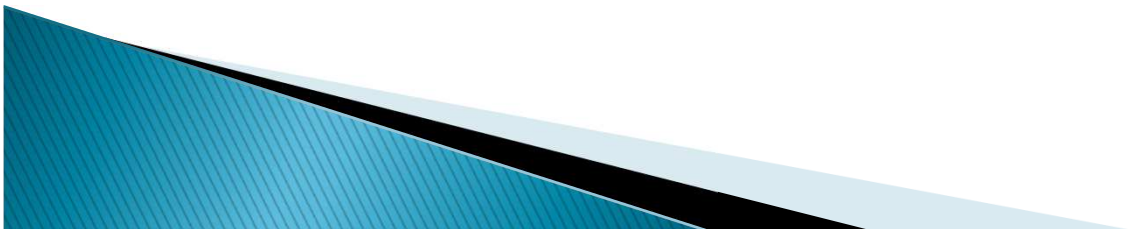
```
class Hilonuevocliente extends Thread {  
    ...  
    public void run() {  
        try {  
            // Establecer los flujos de entrada/salida para el socket /  
            / Procesar las entradas y salidas según el protocolo  
            // cerrar el socket }  
        catch (Exception e) { // manipular las excepciones } } }
```

Conexión de múltiples clientes. Hilos



Ej Ap Cliente-Servidor

- ▶ Escribir la aplicación anterior para el Servidor cree un hilo por cada conexión y admita hasta 4 conexiones concurrentes.



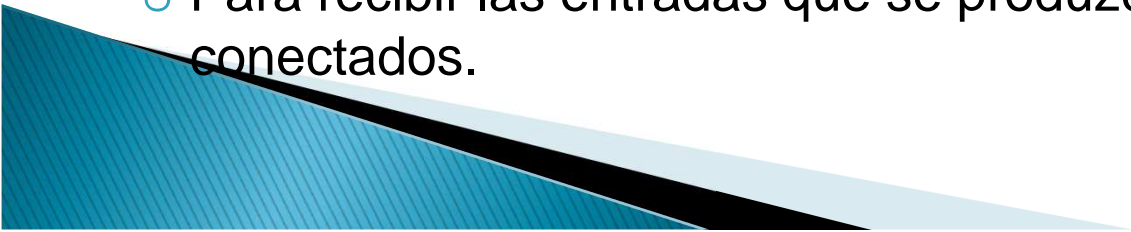
Manejo de Streams. Entrada

► **DataInputStream**

► En el Cliente:

- `DataInputStream` entrada;
- `entrada = new DataInputStream(miCliente.getInputStream());`
- La clase **`DataInputStream`** permite la lectura de líneas de texto y tipos de datos primitivos de Java de un modo altamente portable; dispone de métodos para leer todos esos tipos como: *`read()`*, *`readChar()`*, *`readInt()`*, *`readDouble()`* y *`readLine()`*. Deberemos utilizar la función que creamos necesaria dependiendo del tipo de dato que esperemos recibir del servidor. **Ej. `entrada.readInt()`**

► En el Servidor

- `DataInputStream` entrada;
 - `entrada = new DataInputStream(socketServicio.getInputStream());`
 - Para recibir las entradas que se produzcan de los clientes conectados.
- 

Manejo de Streams. Salida

▶ **PrintStream o DataOutputStream**

▶ En el Cliente

- `PrintStream salida;`
- `salida = new PrintStream(miCliente.getOutputStream());`
- La clase **PrintStream** tiene métodos para la representación textual de todos los datos primitivos de Java. Métodos *write* y *println()*
- `DataOutputStream salida;`
- `salida = new DataOutputStream(miCliente.getOutputStream());`
- La clase **DataOutputStream** permite escribir cualquiera de los tipos primitivos de Java. Ej. `Salida.writeInt()`

▶ En el Servidor

- `PrintStream salida;`
- `salida = new PrintStream(socketServicio.getOutputStream());`
- Pero también podemos utilizar la clase **DataOutputStream** como en el caso de envío de información desde el cliente.



Protocolo de transporte UDP

- ▶ **Protocolo NO orientado a conexión.** Esto lo hace más rápido que TCP, ya que no es necesario establecer conexiones, etc.
- ▶ No garantiza que los mensajes lleguen siempre.
- ▶ No garantiza que los mensajes lleguen en el mismo orden que fueron enviados.
- ▶ Permite enviar mensajes de 64 KB **como máximo.**
- ▶ En UDP, los mensajes se denominan “datagramas” (*datagrams* en ingles).



Protocolo de transporte UDP

- ▶ Un paquete datagrama está formado por los siguientes campos:
 - Cadena de BYTES. Contenido del mensaje
 - Longitud del mensaje
 - Dirección IP Destino
 - N° de Puerto Destino

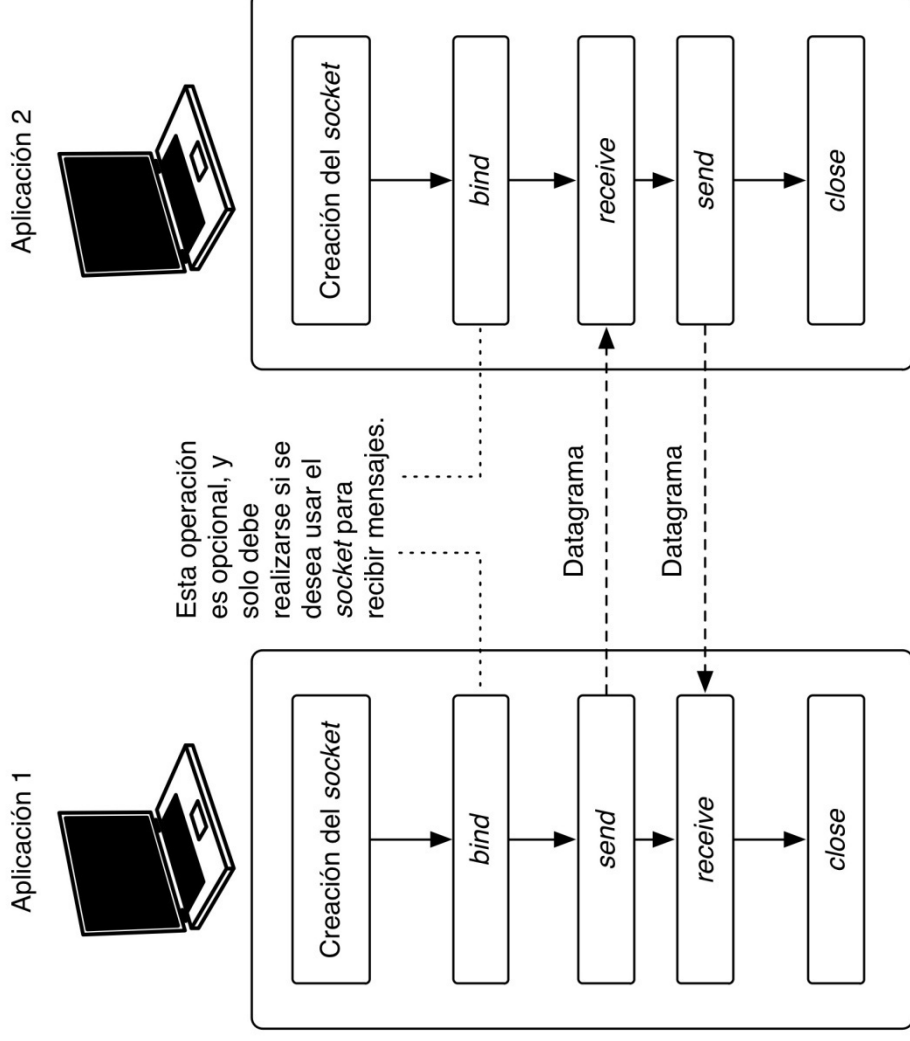


Sockets datagram

- ▶ Son no orientados a conexión.
- ▶ Cuando operan sobre IP, emplean UDP.
- ▶ Cuando se usan *sockets datagram* no existe diferencia entre proceso servidor y proceso cliente.
- ▶ Pasos para enviar mensajes:
 - **Creación** del socket.
 - Asignación de dirección y puerto (***bind***). Solo necesaria para poder recibir mensajes.
 - **Envío** y/o **recepción** de mensajes.
 - **Cierre** del socket.



Sockets datagram



Programación con sockets

- ▶ ***java.net.Socket***
 - para la creación de *sockets stream* cliente.
- ▶ ***java.net.ServerSocket***
 - para la creación de *sockets stream* servidor.
- ▶ ***java.net.DatagramSocket***
 - para la creación de *sockets datagram*.
- ▶ ***java.net.DatagramPacket***
 - Para crear instancias a partir de los datagramas recibidos



Clase DatagramPacket

Método	Descripción
DatagramPacket(byte[] buf, int length)	Constructor para datagramas recibidos . Se especifica la cadena de bytes en la que alojar el mensaje (buf) y la longitud (length) de la misma.
DatagramPacket(byte[] buf, int length, InetAddress address, int port)	Constructor para el envío de datagramas. Se especifica la cadena de bytes a enviar (buf), la longitud (length), el número de puerto de destino (port) y el host especificado en la dirección address
InetAddress getAddress()	Devuelve la dirección IP del host al cual se le envía el datagrama o del que se recibió el datagrama
Byte[] getData()	Devuelve el mensaje contenido en el datagrama tanto recibido como enviado
Int getLength()	Devuelve la longitud de los datos a enviar o recibir
Int getPort()	Devuelve el número de puerto de la máquina remota a la que se va a enviar el datagrama o del que se recibió el datagrama
setAddress(InetAddress addr)	Establece la dirección IP de la máquina a la que se envía el datagrama
setData(byte[buf])	Establece el búfer de datos para este paquete
setLength(int length)	Ajusta la longitud del paquete
setPort(int Port)	Establece el número de puerto del host remoto al que se envía el datagrama.

Ejemplo DatagramPacket

- ▶ Envío de un datagrama


```
Int port=12345; //puerto de envío  
InetAddress destino= InetAddress.getLocalHost(); // IP a la que envío  
byte[] mensaje=new byte[1024]; //matriz de bytes  
String Saludo = "Enviando saludos !!";  
mensaje = Saludo.getBytes(); //Se codifica a bytes para enviarlo  
//Construcción del datagrama para envío  
DatagramPacket envio= new DatagramPacket (mensaje, mensaje.length, destino,  
    port);
```

- ▶ Recibir un mensaje de un datagrama

```
byte[] bufer= new byte[1024];  
DatagramPacket recibido=new DatagramPacket(bufer, bufer.length);
```

- ▶ Obtener longitud de un datagrama recibido y pasarlo a String

```
Int bytesRec= recibido.getLength(); //obtener longitud del mensaje  
String paquete = new String (recibido.getData()); //obtener el mensaje  
System.out.println ("Puerto origen del mensaje; " + recibido.getPort());  
System.out.println ("IP de origen; " + recibido.getAddress().getHostAddress());
```



Clase DatagramSocket

- ▶ La clase DatagramSocket (java.net.DatagramSocket) se utiliza para crear y operar con sockets datagram.

Método	Descripción
DatagramSocket()	Constructor básico de la clase. Sirve para crear sockets datagram. El sistema elige un puerto de los que estén libres
DatagramSocket(SocketAddress bindaddr)	Constructor de la clase con operación bind incluida. Sirve para crear sockets datagrama asociados a una dirección y puerto especificado
DatagramSocket(int port, InetAddress ip)	Constructor de socket datagrama y lo conecta al puerto y dirección local asociada
Send (DatagramPacket p)	Envía un datagrama a través del socket. El argumento contiene el mensaje y su destino. Puede lanzar IOException
Receive (DatagramPacket p)	Recibe un datagrama del socket. Llena p con los datos recibidos (mensaje, longitud y origen). Puede lanzar IOException
Close()	Cierra el socket
Connect(InetAddress address, int Port)	Conecta el socket a un puerto remoto y una IP concreta. El socket solo podrá enviar y recibir mensajes desde esa dirección
setSoTimeout(int timeout)	Permite establecer un tiempo de espera límite. Provoca que el método receive() se bloquee durante el tiempo fijado. Si no se reciben datos se lanza la excepción InterruptedException

Ejemplo DatagramSocket

- ▶ Envío del Datagrama Packet usando DatagramSocket

//Construcción del datagrama a enviar indicando el host destino y el puerto.

DatagramPacket **envio** = new DatagramPacket (mensaje, mensaje.length, destino, port);

DatagramSocket socket= new DatagramSocket(34567);

socket.send(envio); //envío datagrama a destino y puerto

- ▶ Recepción del datagrama mediante DatagramSocket.

DatagramSocket socket= new DatagramSocket(12345);

//construcción del datagrama a recibir

DatagramPacket **recibido** = new DatagramPacket(bufer, bufer.length);

socket.receive(recibido); //recepción del datagrama

Int bytesRec = recibido.getLength(); //Obtención del número de bytes

String **paquete** = new String (**recibido.getData()**); //obtener el String

System.out.println("Núm de Bytes recibidos: " + bytesRec);

System.out.println("Contenido del paquete; " + paquete.trim());

System.out.println("Puerto Origen del mensaje; " + **recibido.getPort()**);

System.out.println("IP de Origen: " + **recibido.getAddress().getHostAddress()**);

System.out.println("Puerto destino del mensaje; " + **socket.getLocalPort()**);

Socket.close(); //Cierre del socket



Gestión de Sockets UDP

Socket Datagram

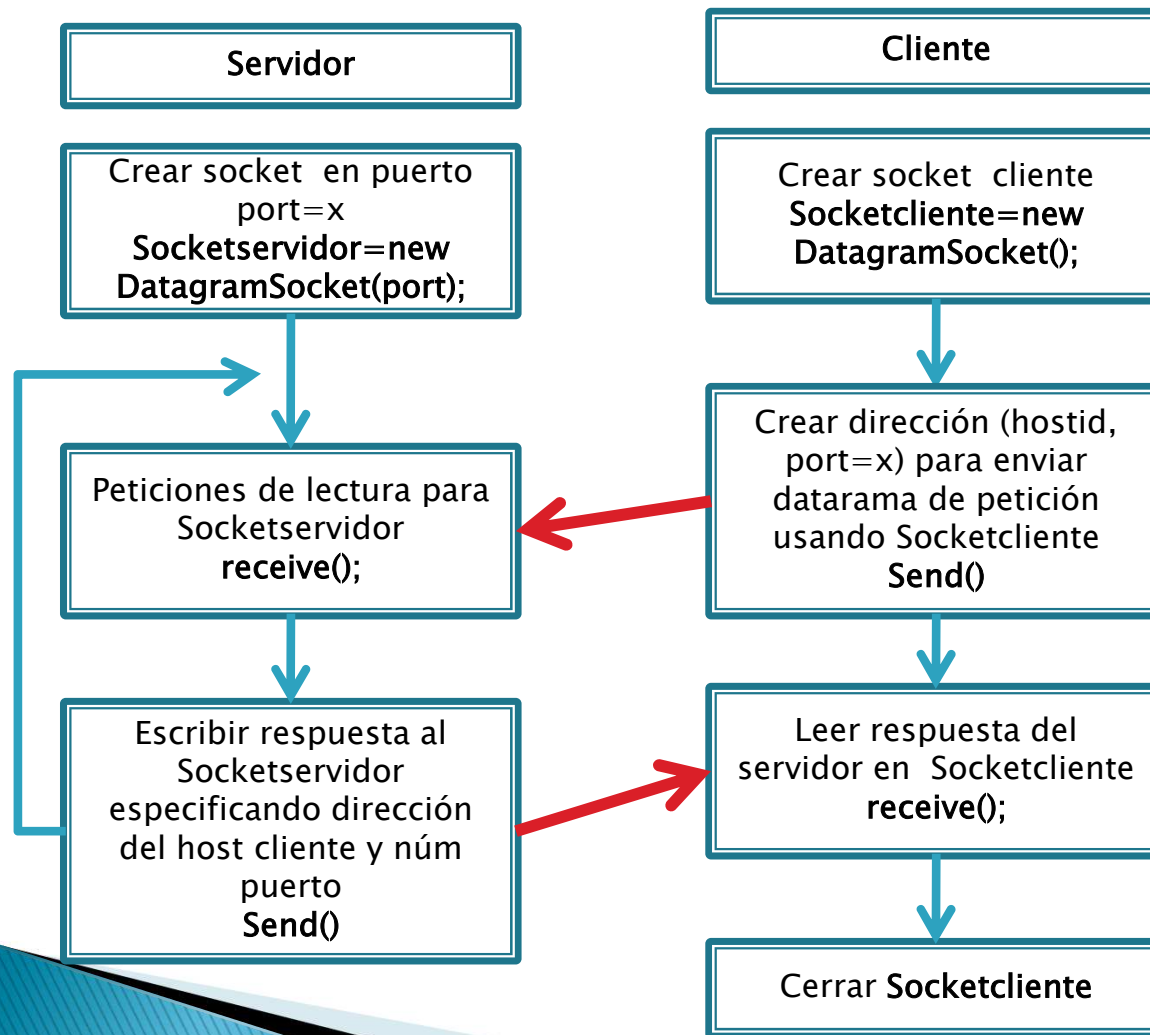
El **servidor** crea un socket asociado a un puerto local para escuchar peticiones de clientes. Permanece a la espera.

El **cliente** crea un socket para comunicarse con el y necesita conocer su IP y su puerto de escucha. Usa send para enviar una petición al servidor

El **servidor** recibe por el socket y mediante send() del socket envía respuesta

El **cliente** recibe la respuesta mediante receive().

El **servidor** continua a la espera de más peticiones



Ejemplo sockets datagram

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

public class EmisorDatagram {

    public static void main(String[] args){
        try {
            DatagramSocket datagramSocket = new DatagramSocket();
            String mensaje = "mensaje desde el emisor";
            InetAddress addr = InetAddress.getByName("localhost");
            DatagramPacket datagrama = new DatagramPacket(mensaje.getBytes(),
                                                            mensaje.getBytes().length, addr, 5555);

            datagramSocket.send(datagrama);
            datagramSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```



Ej Proceso que recibe un mss y lo envía usando sockets datagram

//Programa que requiere un cliente que le envíe mensajes

```
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.InetSocketAddress;
```

```
public class socketdatagrama {
```

```
    public static void main(String[] args) {
```

```
        try{
```

```
            System.out.println("Creando socket datagrama");
```

```
            InetSocketAddress addr = new InetSocketAddress("localhost", 5555);
```

```
            DatagramSocket datagramSocket = new DatagramSocket(addr);
```

```
            System.out.println("Recibiendo mensaje");
```

```
            byte[] mensaje = new byte[25];
```

```
            DatagramPacket datagrama1 = new DatagramPacket(mensaje, 25);
```

```
            datagramSocket.receive(datagrama1);
```

```
            System.out.println("Mensaje recibido: " + new String(mensaje));
```

```
            System.out.println("Enviando mensaje");
```

```
            InetAddress addr2 = InetAddress.getByName("localhost");
```

```
            DatagramPacket datagrama2 = new DatagramPacket(mensaje, mensaje.length, addr2, 5556);
```

```
            datagramSocket.send(datagrama2);
```

```
            System.out.println("Mensaje enviado");
```

```
            System.out.println("Cerrando el socket datagrama");
```

```
            datagramSocket.close();
```

```
            System.out.println("Terminado");
```

```
        } catch (IOException e) { e.printStackTrace(); } }
```



Documento de
icrosoft Office Wo

Ej. Continuación. Socket Datagram Cliente

//Envía un mensaje al Proceso servidor que lo espera

```
package socketdatagramcliente;
```

```
import java.io.IOException;
```

```
import java.net.DatagramPacket;
```

```
import java.net.DatagramSocket;
```

```
import java.net.InetAddress;
```

```
public class socketdatagramcliente {
```

```
    public static void main(String[] args) {
```

```
        try{
```

```
            System.out.println("Creando socket datagram");
```

```
            DatagramSocket datagramSocket = new DatagramSocket();
```

```
            System.out.println("Enviando mensaje");
```

```
            String mensaje = "mss desde el emisor";
```

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

```
            DatagramPacket datagrama=new DatagramPacket (mensaje.getBytes(), mensaje.getBytes().length, addr, 5555);
```

```
            datagramSocket.send(datagrama);
```

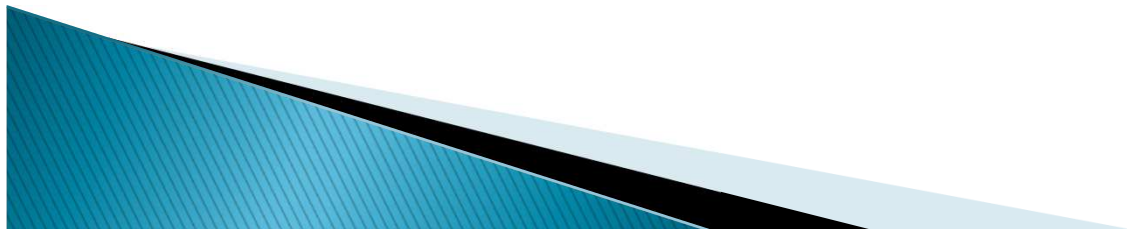
```
            System.out.println("Mensaje enviado");
```

```
            System.out.println ("Cerrando el socket datagrama");
```

```
            datagramSocket.close();
```

```
            System.out.println("Terminado");
```

```
        } catch (IOException e) { e.printStackTrace();}}
```



Modelos de comunicaciones

- ▶ Los **sockets** son una herramienta básica para enviar y recibir mensajes.
- ▶ A la hora de desarrollar aplicaciones distribuidas debemos tener en cuenta aspectos de más alto nivel.
- ▶ Dependiendo de cuál sea el propósito de nuestra aplicación, y cómo vaya a funcionar internamente, deberemos escoger un modelo de comunicaciones distinto.



Modelos de comunicaciones

- ▶ Un **modelo de comunicaciones** es una arquitectura general que especifica cómo se comunican entre sí los diferentes elementos de una aplicación distribuida.
- ▶ Un modelo de comunicaciones normalmente define aspectos como cuántos elementos tiene el sistema, qué función realiza cada uno, etc.
- ▶ Los modelos más usados en la actualidad son:
 - *Cliente/servidor.*
 - *Comunicación en grupo.*

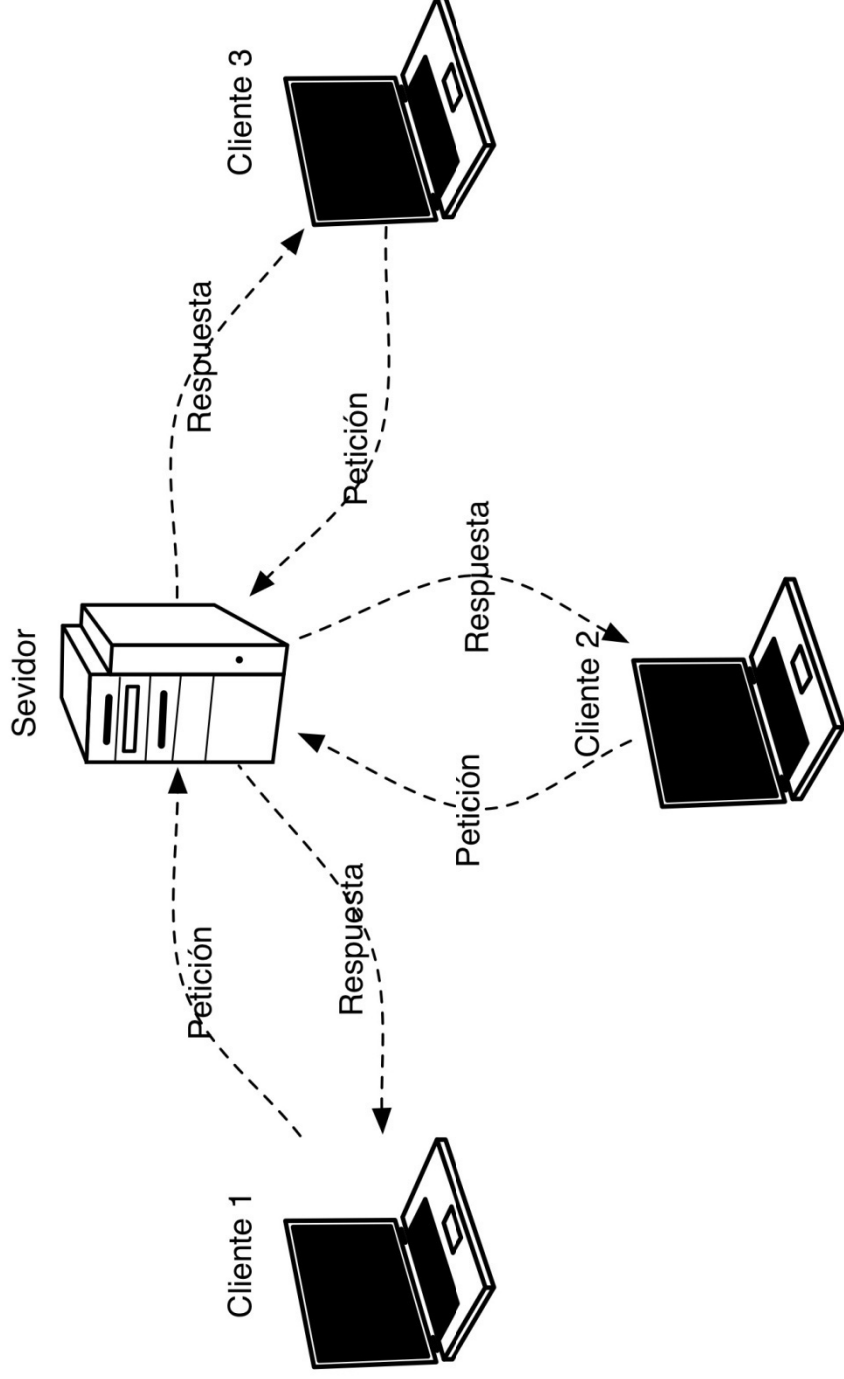


Modelo cliente/servidor

- ▶ El más sencillo de los comúnmente usados en la actualidad.
- ▶ En este modelo, un proceso central, llamado *servidor*, ofrece una serie de servicios a uno o más procesos *cliente*.
- ▶ El proceso *servidor* debe estar alojado en una máquina fácilmente accesible en la red, y conocida por los *clientes*.
- ▶ Cuando un *cliente* requiere sus servicios, se conecta con el *servidor*, iniciando el proceso de comunicación.

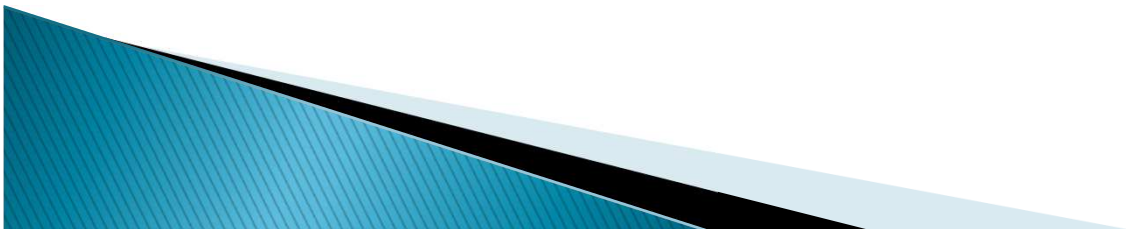


Modelo cliente/servidor

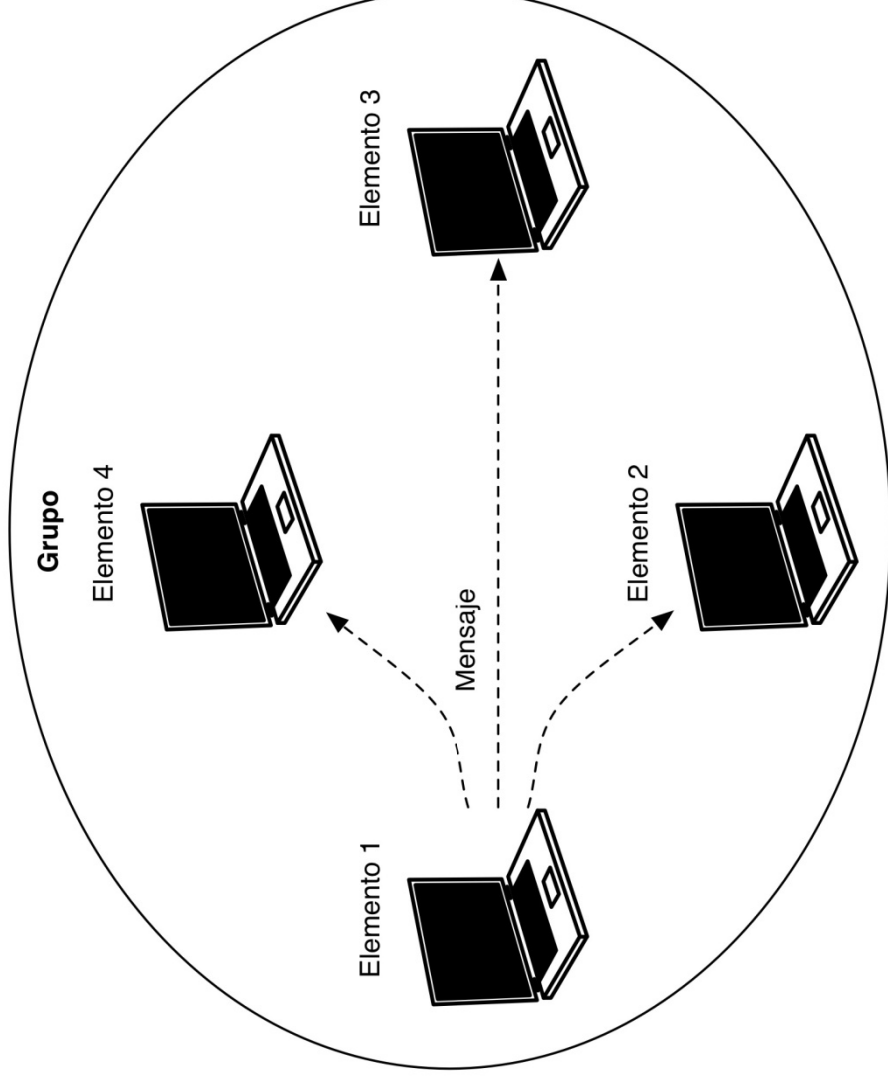


Modelo de comunicación en grupo

- ▶ Es la alternativa más común al modelo cliente/servidor.
- ▶ En este modelo no existen roles diferenciados.
- ▶ En la comunicación en grupo existe un conjunto de dos o más elementos (procesos, aplicaciones, etc.) que cooperan en un trabajo común.
- ▶ A este conjunto se le llama *grupo*, y los elementos que lo forman se consideran todos iguales, sin roles ni jerarquías definidas.
- ▶ Los mensajes se transmiten mediante *radiado*.



Modelo de comunicación en grupo



Modelos híbridos

- ▶ Las aplicaciones distribuidas más avanzadas suelen tener requisitos de comunicaciones muy complejos, que requieren de modelos de comunicaciones sofisticados.
- ▶ En muchos casos, los modelos de comunicaciones reales implementados en estas aplicaciones mezclan conceptos del modelo cliente/servidor y la comunicación en grupo, dando lugar a enfoques híbridos, como las redes *peer-to-peer* (P2P).

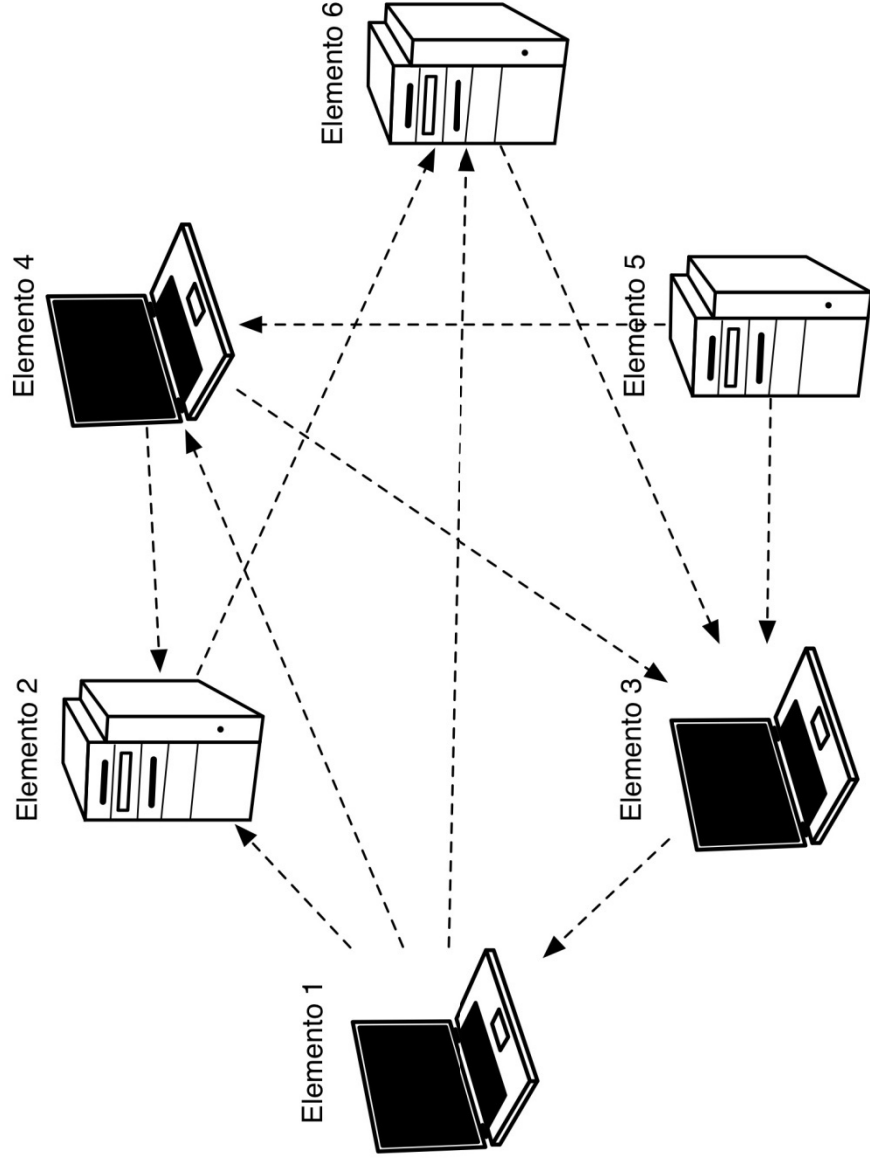


Redes *peer-to-peer*

- ▶ Una red P2P está formada por un grupo de elementos distribuidos que colaboran con un objetivo común.
- ▶ Cualquier elemento puede desempeñar los roles de *servidor* o *cliente*, como si de un modelo cliente/servidor se tratase.
- ▶ Las redes P2P puedan ofrecer servicios de forma similar al modelo cliente/servidor.
- ▶ Cualquier aplicación puede conectarse a la red como un *cliente*, localizar un *servidor* y enviarle una petición.
- ▶ Si permanece en la red P2P, con el tiempo ese mismo *cliente* puede hacer a su vez de *servidor* para otros elementos de la red.



Redes *peer-to-peer*



Multicast. Comunicación en grupo

- ▶ El mensaje es replicado y entregado a cada uno de los destinatarios.
- ▶ Permite la comunicación en grupo de forma muy eficiente.
- ▶ Supone riesgo para la seguridad de la red, ya que cualquiera puede enviar cientos de mensajes y saturar a todos los miembros de un grupo.
- ▶ El tráfico multicast está restringido en Internet y se usa en redes de área local.
- ▶ Se suelen usar bibliotecas para su implementación. Ej JGroups de java.



MulticastSocket


- ▶ La clase **MulticastSocket** es útil para enviar paquetes a múltiples destinos simultáneamente.
- ▶ Para poder recibir estos paquetes es necesario establecer un grupo multicast (grupo de direcciones IP, que comparten el mismo puerto).
- ▶ El emisor no conoce el número de miembros del grupo ni sus direcciones IP.
- ▶ Grupo Multicast: Dirección IP de Clase D y un puerto UDP Estándar.
 - Desde la 224.0.0.0 hasta 239.255.255.255
 - 224.0.0.0 está reservada y no debe ser usada



MulticastSocket. Constructores y métodos

CONSTRUCTOR	DESCRIPCION
<code>MulticastSocket()</code>	Construye un multicast socket dejando al sistema que elija el puerto de los que están libres
<code>MulticastSocket(int port)</code>	Construye un multicast socket y lo conecta al puerto local especificado.

METODO	DESCRIPCION
<code>joinGroup (InetAddress mcastaddr)</code>	Permite al socket multicast unirse al grupo de multicast
<code>leaveGroup (InetAddress mcastaddr)</code>	El Socket multicast abandona el grupo de multicast
<code>Send (DatagramPacket P)</code>	Envía el datagrama a todos los miembros del grupo multicast
<code>Receive (DatagramPacket p)</code>	Recibe el datagrama de un grupo multicast



MulticastSocket. Servidor Multicast

```
//Se crea el socket multicast. No hace falta especificar puerto  
MulticastSocket ms = new MulticastSocket();
```

```
// se define el puerto multicast  
Int Puerto = 12345;
```

```
// se crea el grupo multicast  
InetAddress grupo = InetAddress.getByName("225.0.0.1");  
String msg = "Bienvenidos!!";
```

```
//se crea el datagrama  
DatagramPacket paquete = new DatagramPacket(msg.getBytes(), msg.length(), grupo, Puerto);
```

```
//se envía el paquete al grupo;  
ms.send(paquete);
```

```
//se cierra el socket  
ms.close();
```



MulticastSocket. Cliente Multicast

//se crea un socket multicast en el puerto 12345

```
MulticastSocket ms = new MulticastSocket(12345);
```

//se configura la IP del grupo al que nos conectaremos

```
InetAddress grupo = InetAddress.getByName("225.0.0.1");
```

// se une al grupo

```
ms.joinGroup(grupo);
```

//recibe el paquete del servidor multicast

```
Byte[] buf = new byte[1000];
```

```
DatagramPacket recibido = new DatagramPacket(buf,buf.length);
```

```
ms.receive(recibido);
```

//salir del grupo multicast

```
ms.leaveGroup (grupo);
```

//se cierra el socket


```
ms.close();
```



Multicast. Ej Servidor multicast

Servidor que lee datos de teclado y los envía a todos los clientes que pertenezcan al grupo multicast, el proceso finaliza cuando se introduzca un *

```
import java.io.*;
import java.net.*;
public class servidorMC1 {
    public static void main(String rgs[]) throws Exception {
        //flujo para la entrada estandar
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        //se crea el socket multicast
        MulticastSocket ms = new MulticastSocket();
        int Puerto = 12345; //puerto multicast
        InetAddress grupo = InetAddress.getByName("225.0.0.1"); // Grupo
        String cadena="";
        while(!cadena.trim().equals("*")) {
            System.out.print("Datos a enviar al grupo ");
            cadena = in.readLine();
            // Enviando al grupo
            DatagramPacket paquete = new DatagramPacket ( cadena.getBytes(),
cadena.length(), grupo, Puerto);
            ms.send(paquete);
        }
        ms.close(); // se cierra socket
        System.out.println (" Socket cerrado...");
    }
}
```



Multicast. Ej. Cliente multicast

Visualiza el paquete que recibe del servidor, su proceso finaliza cuando recibe un asterisco

```
import java.io.*;
import java.net.*;
public class clienteMC1 {
    public static void main(String args[]) throws Exception {
        // se crea el socket multicast
        int Puerto = 12345; // Puerto multicast
        MulticastSocket ms = new MulticastSocket(Puerto);
        InetAddress grupo = InetAddress.getByName("225.0.0.1"); // Grupo
        // Unión al grupo
        ms.joinGroup(grupo);
        String msg=" ";
        byte[] buf = new byte[1000];
        while(!msg.trim().equals("*")) {
            //recibe el paquete del servidor multicast
            DatagramPacket paquete = new DatagramPacket(buf, buf.length);
            ms.receive(paquete);
            msg = new String(paquete.getData());
            System.out.println("Recibido: " + msg.trim());
        }
        ms.leaveGroup (grupo); // abandonar el grupo
        ms.close(); // cierra socket
        System.out.println ("Socket cerrado...");
    }
}
```

Ejecutar servidor en una consola
Y diferentes instancias del cliente en otras diferentes

Creación de un Chat. TCP

- ▶ Chat TCP. Atiende múltiples clientes. Multihilo.



Documento de
icrosoft Office Wo



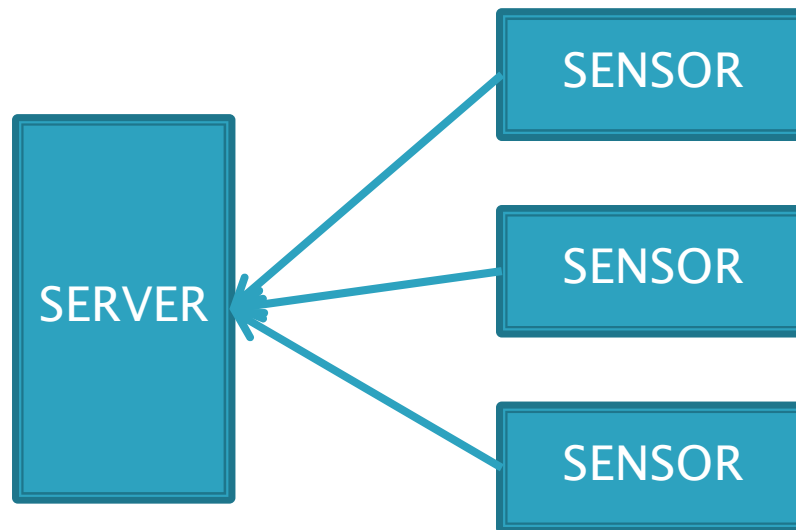
Ejercicios Propuestos

- ▶ **Calculadora.** El servidor recibe 2 números y una operación y devuelve el cálculo al cliente.
- ▶ **Examen ON LINE** tipo test. El servidor plantea una pregunta y 4 posibles respuestas al cliente de una lista de 20 posibles. Una vez recibida la respuesta. Devuelve al cliente un mensaje de si es correcta o no.



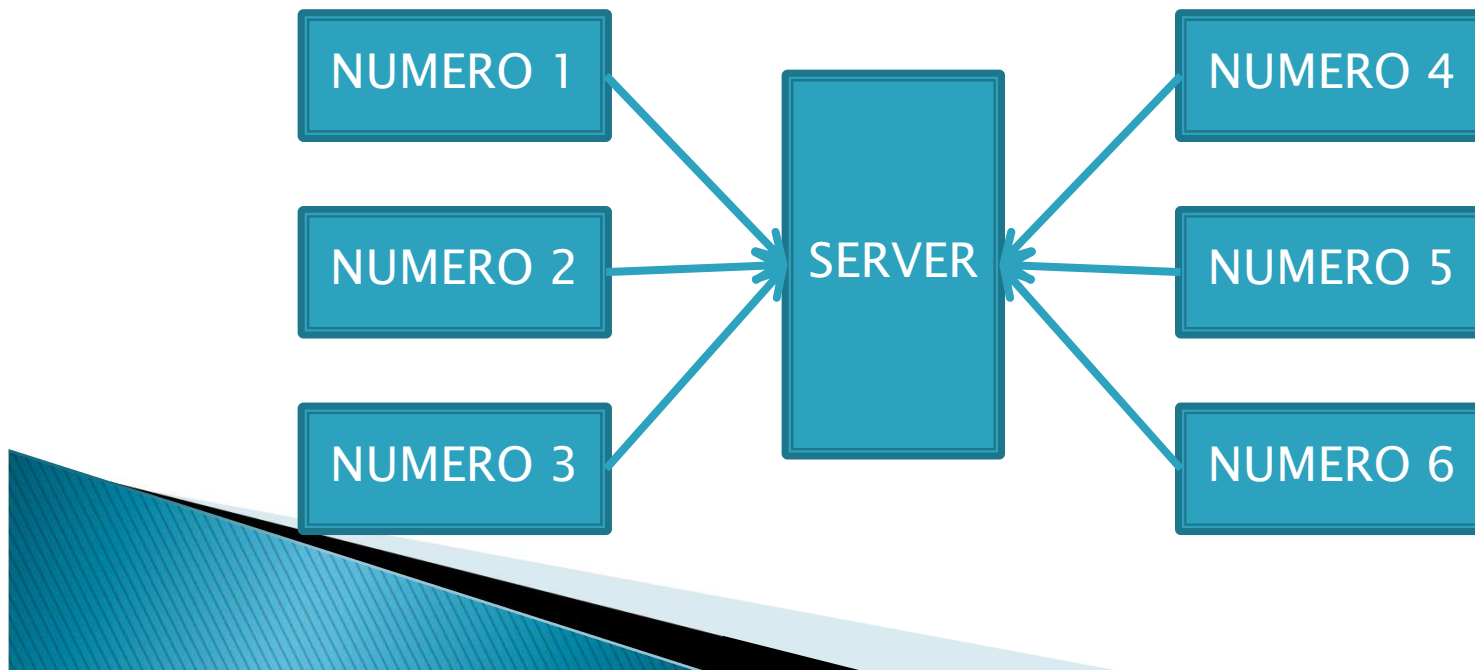
Ejercicios Propuestos

- ▶ **Estación Meteorológica.** Servidor recibe datos de temperatura de diferentes sensores clientes. El servidor va presentando datos estadísticos (temperaturas medias, máximas y mínimas) (UTP y TCP)



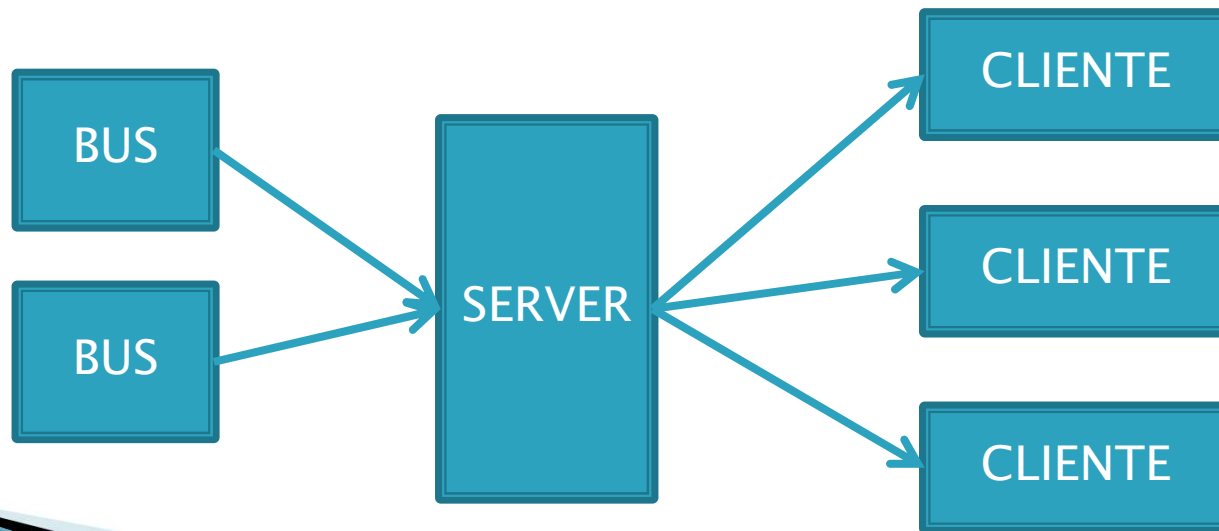
Ejercicios Propuestos

- ▶ **QUINIELA:** Cada Cliente envía un número y se hace la quiniela. El servidor devuelve a cada cliente la quiniela completa una vez recibidos todos los números (**TCP y UDP**).



Ejercicios Propuestos

- ▶ **EL BUS.** Un autobús emite a un servidor cada X tiempo. La posición donde se encuentran y el tiempo estimado que tardará en llegar a la siguiente parada. Al servidor se conectan clientes que reciben esta información . (TCP Y UDP)



Ejercicios Propuestos

- ▶ **Varios clientes escriben en el mismo fichero de texto de un servidor.**
- ▶ Implementar un **servicio de Chat** con varios clientes (TCP Y UDP).
- ▶ **Lector de ficheros de texto on-line.** El servidor tiene un libro en formato texto, lo va leyendo y se lo va enviando al cliente que lo va leyendo. Opcional: Almacena el lugar de lectura.

