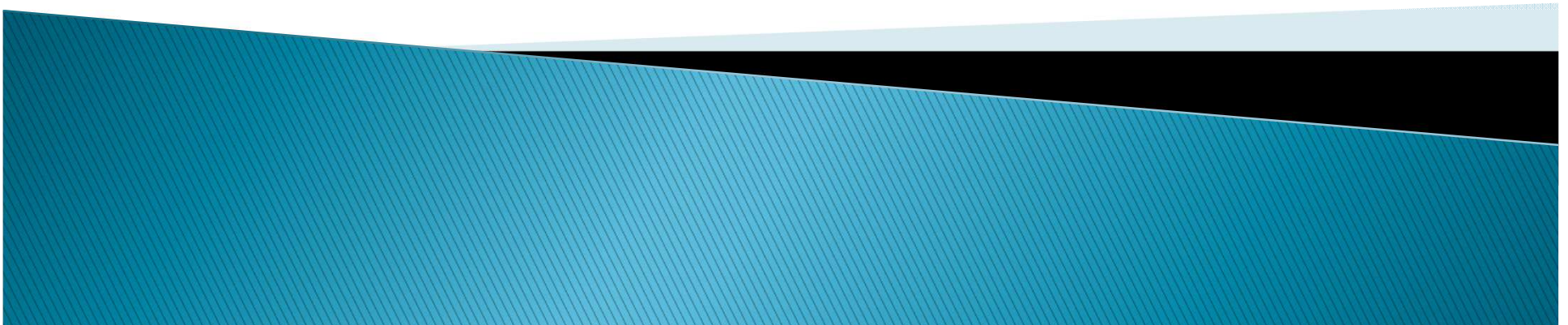


TEMA 3: PROGRAMACIÓN DE COMUNICACIONES EN RED

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**.

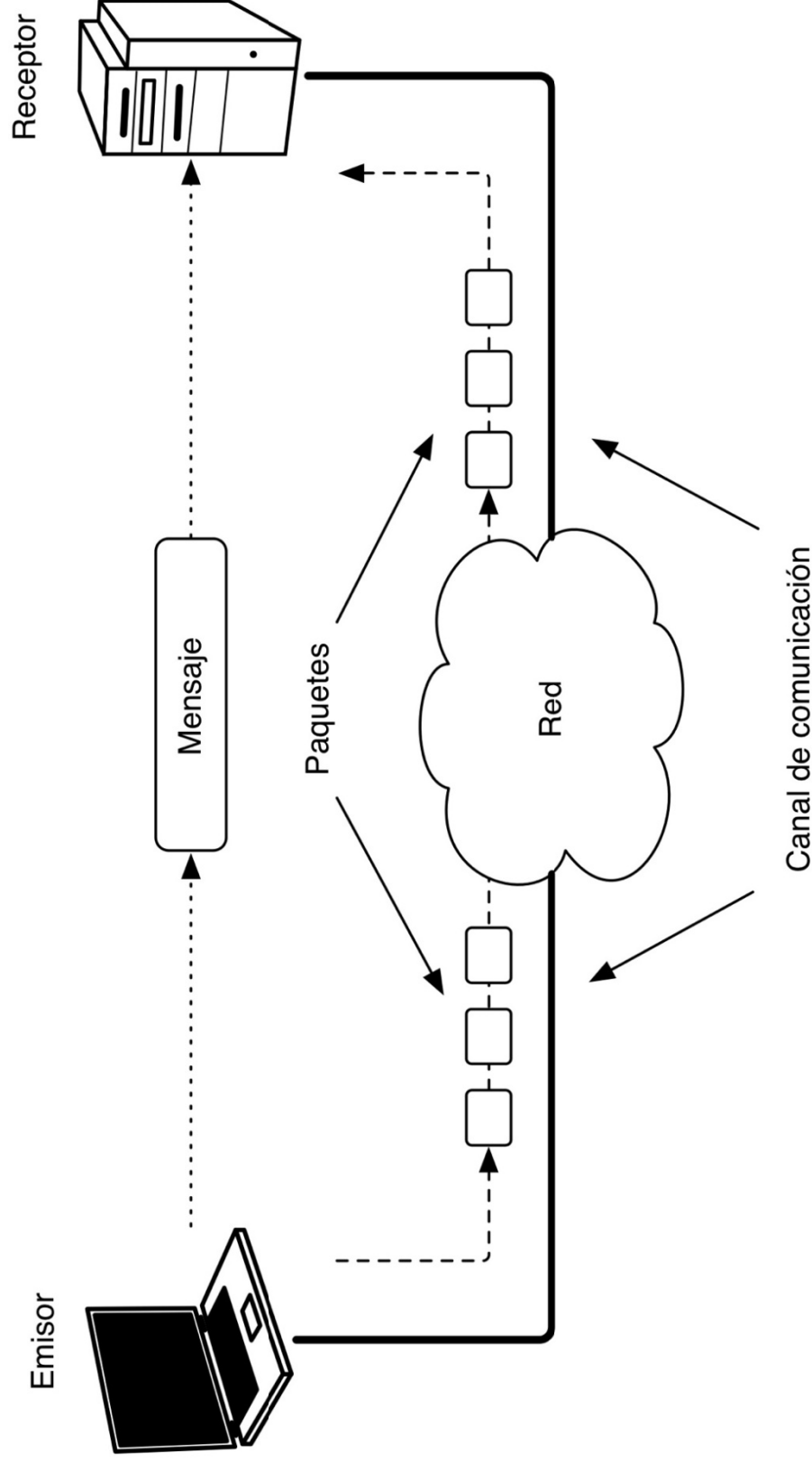


Comunicación entre aplicaciones

- ▶ 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.

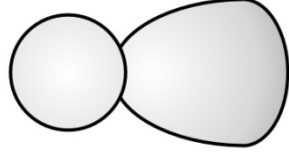


Protocolos de comunicaciones

- ▶ 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:

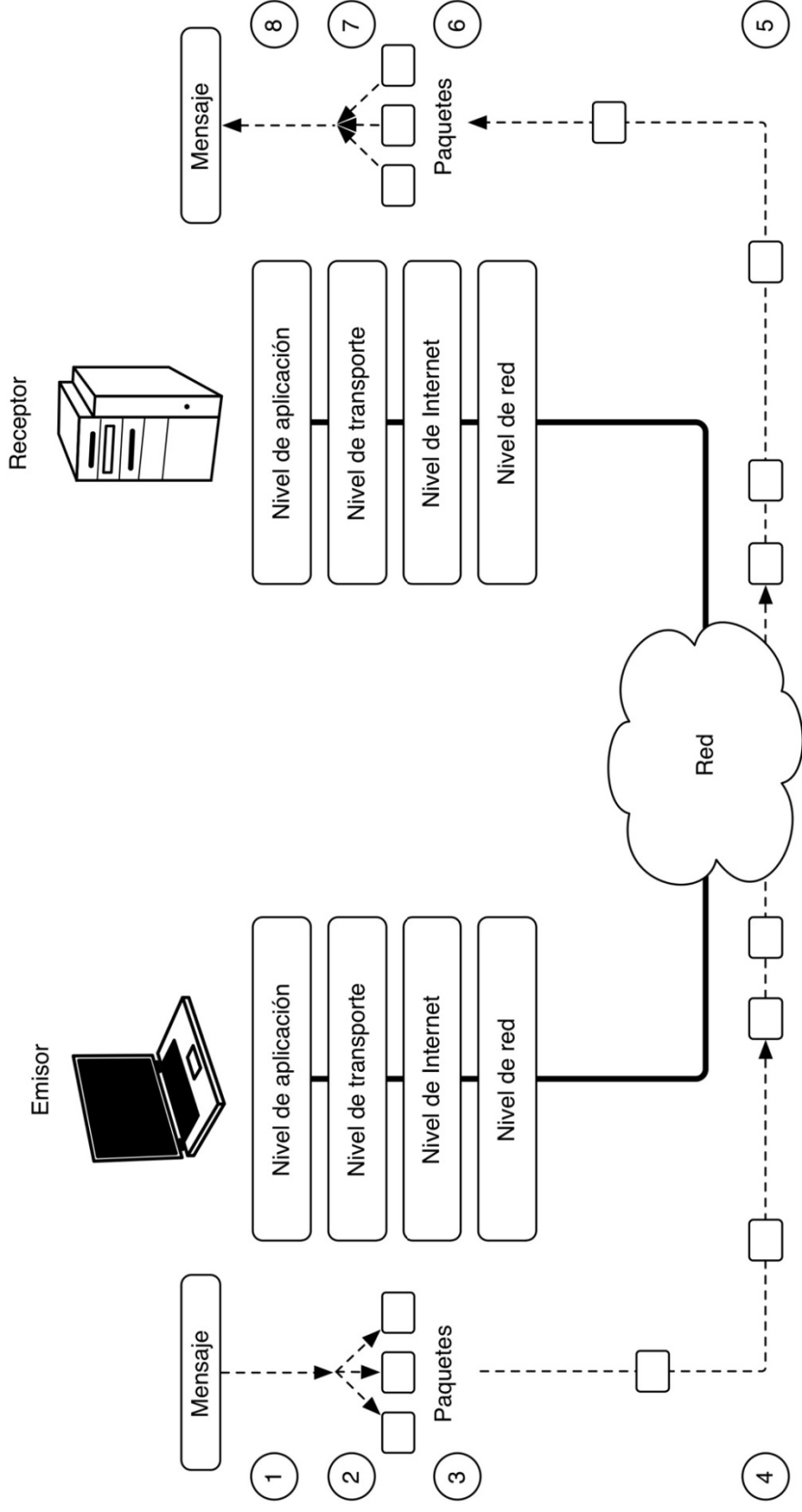
- Lo componen los elementos software cuya función es crear el canal de comunicación, descomponer el mensaje en paquetes y gestionar su transmisión entre el emisor y el receptor.
- Los dos protocolos de transporte fundamentales: TCP y UDP.

▶ 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.

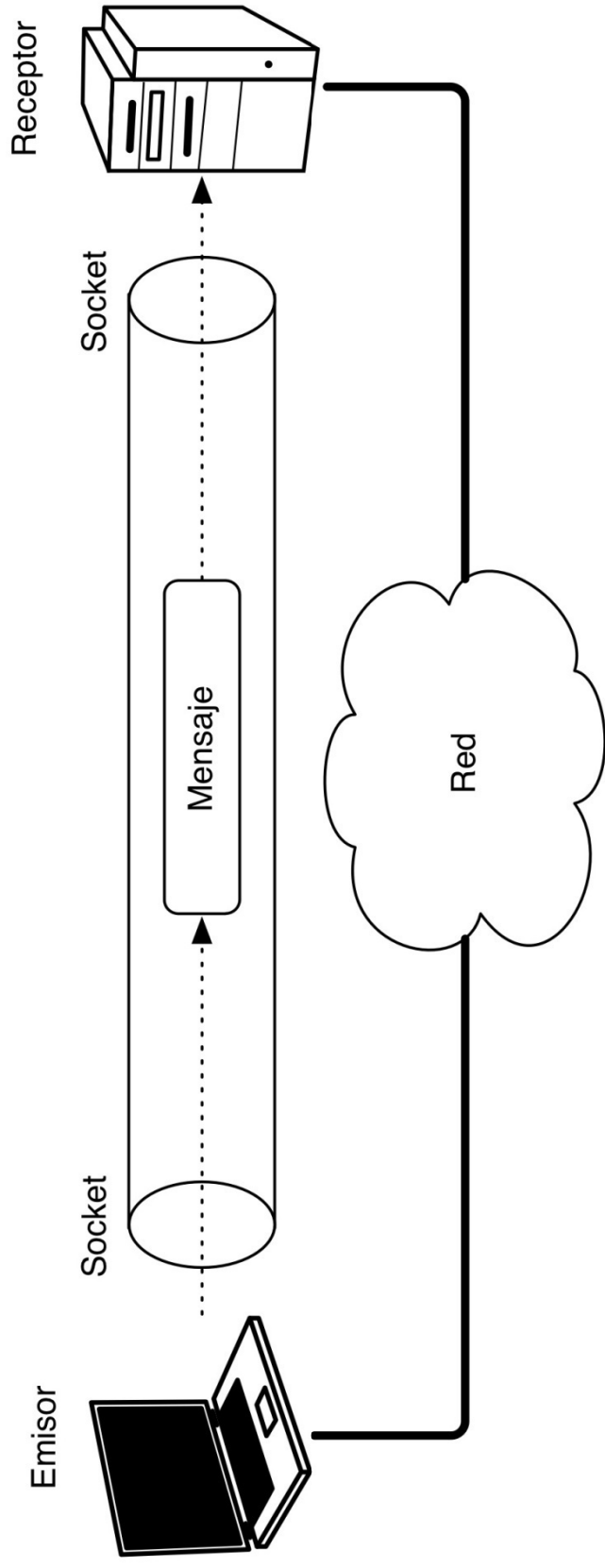


Sockets

- ▶ 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***).



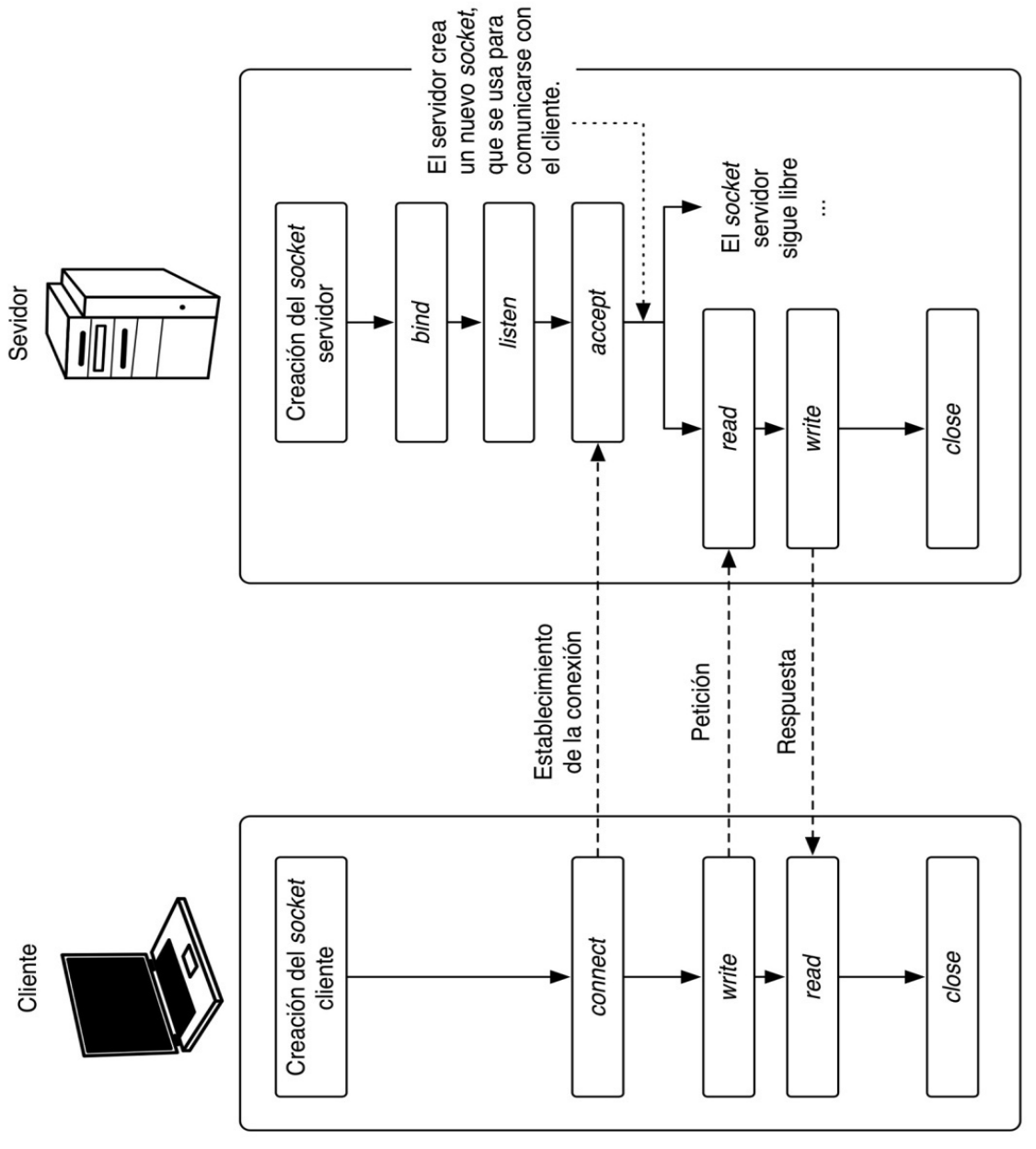
Sockets stream

► 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

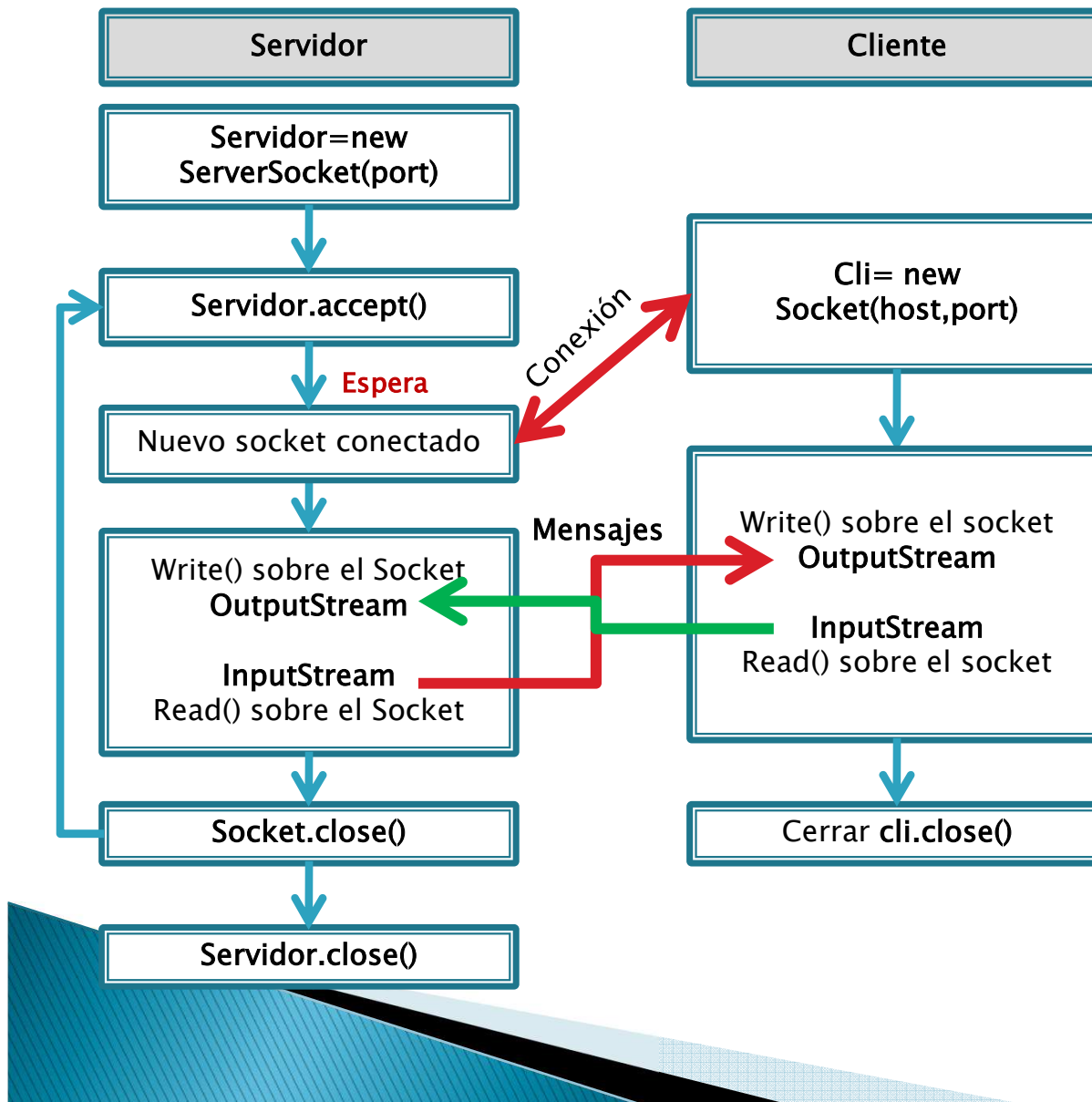
Socket Stream

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()`.

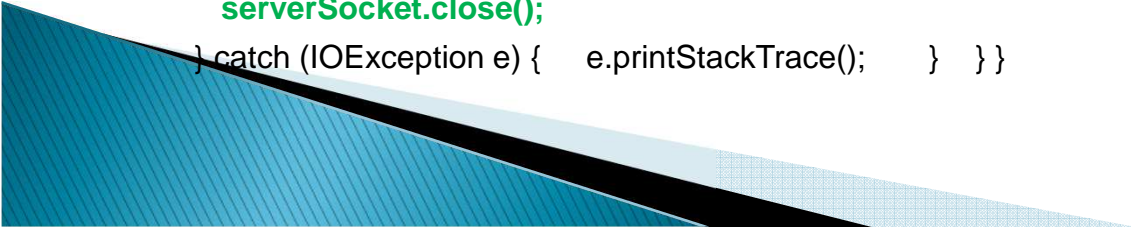
El **cliente** establece una conexión con el host a través del puerto especificado mediante el método **`Socket(host,port)`**.

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));
            Stream 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.

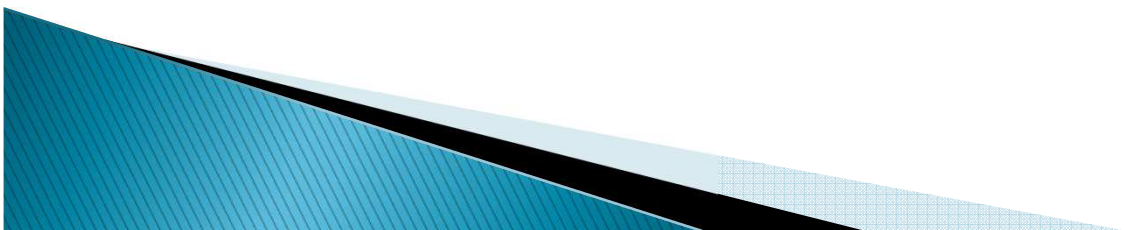
El cliente lee del socket el mensaje recibido y lo muestra por pantalla.

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.

