2025

UNIDAD 3. PROGRAMACIÓN DE COMUNICACIONES EN RED

2ºDAM. PROGRAMACIÓN DE SERVICIOS Y PROCESOS

FRAN HUERTAS

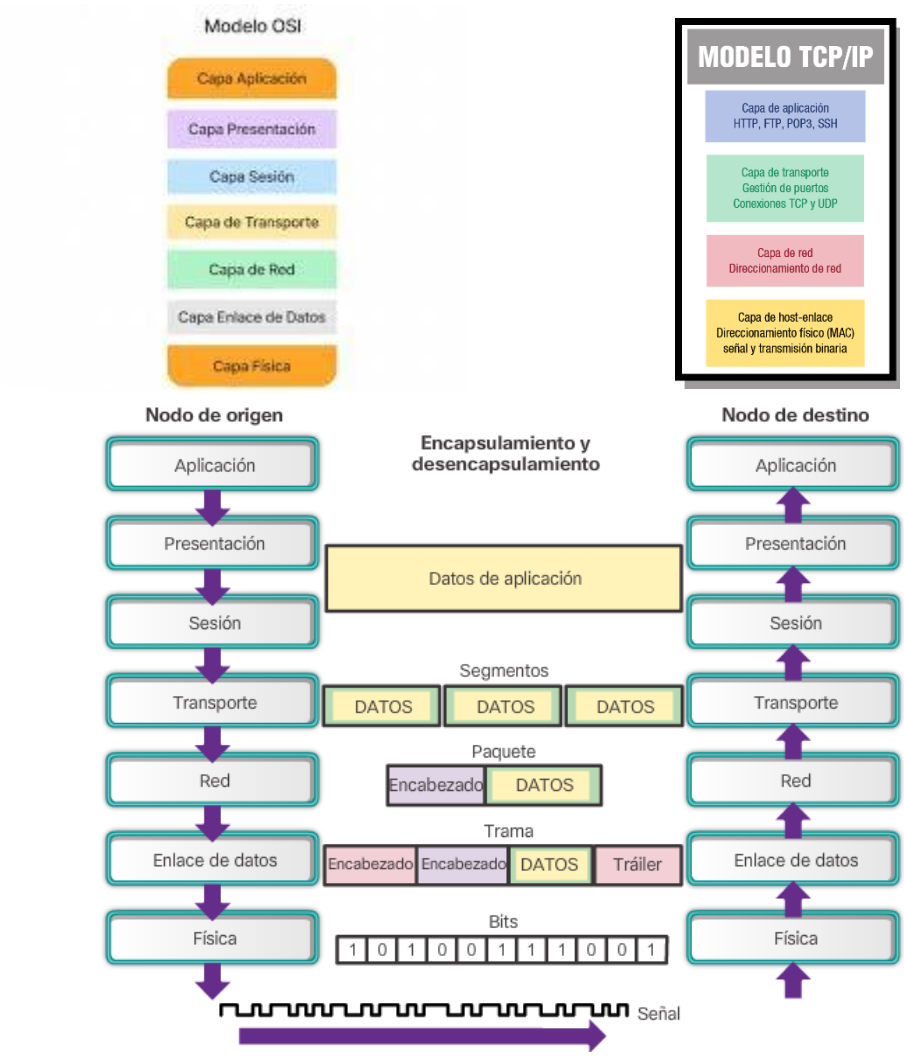
Contenido

CONCEPTOS PREVIOS

Para el desarrollo de aplicaciones que se comuniquen a través de una red debemos conocer el funcionamiento, a grandes rasgos, de ésta.

Aunque existen diferentes arquitecturas de red para comunicar ordenadores y otros dispositivos, sin duda la más extendida y la que nosotros vamos a utilizar es la que se basa en el modelo TCP/IP, que es el que usa Internet.

Para establecer una referencia consideraremos el modelo OSI (modelo teórico de la ISO) que define una arquitectura de comunicación en red divida en 7 capas cada una encargada de una parte de la comunicación. La comunicación dentro de cada nodo es entre capas adyacentes. Cada capa añade su encabezado al mensaje recibido de la capa superior. Este modelo tiene una equivalencia con el modelo TCP/IP que se reduce a 4 capas.



MODELO TCP/IP

Realmente este modelo solo define las capas 2 y 3. Vamos a ver la función de cada una de ellas.

Capa de host-enlace. Esta capa se encarga de negociar con el hardware de red, empaquetar/desempaquetar las tramas, controlar el acceso al medio y realizar el direccionamiento físco (MAC) y la tradución entre direcciones lógicas (IP) y físicas.

Capa de red. Su función es encaminar los paquetes hasta su destino final. Utiliza una dirección que identifica de forma única a cada dispositivo de comunicaciones llamado dirección IP.

Tenemos dos versiones: IPv4 que usa un número de 32 bits e IPv6 que usa un número de 128 bits.

Nivel de transporte. Es el encargado de establecer el canal de comunicaciones extremo a extremo. Dispone de dos protocolos: TCP y UDP.

TCP es un protocolo orientado a conexión, es decir, espera confirmación de cada unidad de datos que envía. Siendo su funcionamiento el siguiente: Establecimiento de conexión, Envío de mensajes y Cierre de conexión. Ej. HTTP(80),FTP(21), Telnet(23), SSH(22), SMTP(25),etc.

UDP. User Datagrama Protocol. Se trata de un protocolo no orientado a conexión, sin garantía de entrega del mensaje pero más rápido que TCP. El tamaño máximo del mensaje es de 64k. Ej. TFTP(21), SNMP, DNS(53), etc

Tanto TCP como UDP ofrecen sus servicios a la capa de aplicación a través de los Puntos de Acceso al Servicio, denominados puertos y que se identifican con número comprendido entre 0 y $2^{16} - 1$. De esta manera permiten multiplexar sobre un mismo dispositivo (dirección IP) múltiples servicios o conexiones (Aplicaciones).

- **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.

Nivel de aplicación. Es el nivel más alto de la pila de protocolos y lo forman las aplicaciones que ofrecen sus servicios al usuario o al sistema dónde están instaladas.

NOMBRES DE 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.iax.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.

SOCKETS TCP

El concepto de socket (enchufe) nos abstrae de la complejidad de protocolos que intervienen en el proceso de comunicación, reduciéndolo a la conexión entre extremos.

Representa el extremo de un canal de comunicaciones establecido entre un emisor y un receptor. Requiere una aplicación en cada extremo que cree su socket y se conecten entre sí. Una vez establecida la conexión se crea como una tubería por donde fluyen los mensajes de un extremo a otro.

El envío y recepción de mensajes se reduce a escribir y leer respectivamente, en el socket correspondiente.

Como ya hemos comentado anteriormente, los dispositivos, normalmente una máquina, están identificados de forma única mediante una dirección IP y las aplicaciones de comunicaciones dentro de esa máquina tienen asociado un número de puerto único. Al binomio formado por la dirección IP más el número de puerto se le conoce también como socket, se representa: **IP:puerto**.

TIPOS DE SOCKET

Sockets Stream. Orientados a conexión, por tanto, **usan el protocolo TCP**. Se utiliza para establecer una comunicación extremo a extremo y mantenerla hasta que se cierre.

Cliente	**Servidor**
• Crea un socket. Socket cliente. • Se asigna un número de puerto (el primero disponible) y la IP de la máquina.	• Crea un socket. Socket servidor. • Asigna dirección y puerto (*bind*) • Se pone en escucha de peticiones (*listen*)
• Solicita conexión al socket servidor, especificando la IP y el puerto de éste (*connect*)	• El socket servidor acepta la petición del socket cliente(*accept*)
• Se realiza el intercambio de mensajes (*write y read*)	• Se realiza el intercambio de mensajes (*read y write*)
• Cierra la conexión (*close*)	• Cierra la conexión (*close*) El Socket queda libre para otra comunicación.

Sockets datagram. No orientados a conexión, **usan UDP**. Se pueden usar para enviar mensajes a multitud de receptores. No se distingue entre cliente y servidor.

El proceso es similar al anterior, pero sin necesidad de solicitar conexión ni aceptarla.

Cliente	**Servidor**
• Crea un socket. • Asigna dirección y puerto (*bind*)	• Crea un socket. • Asigna dirección y puerto (*bind*)
• Se realiza el intercambio de mensajes (*send y receive*)	• Se realiza el intercambio de mensajes (*receive y send*)
• Cierra la conexión (*close*)	• Cierra la conexión (*close*)

SOCKETS CON JAVA

Las clases que dispone java para las comunicaciones con sockets son: **DatagramSocket, Socket, ServerSocket, DatagramPacket y MulticastSocket;** se encuentran en el paquete **java.net**.

Por otro lado, la clase **InetAddress** se utiliza para representar una dirección IP.

Para crear un socket cliente, tan solo es necesario instanciar un objeto **Socket** con el constructor por defecto.

Métodos de la clase Socket

Método	Descripción
`Socket(String host, int port)`	Crea un socket y lo conecta al host y puerto especificados
`Socket(InetAddress address, int port)`	Crea un socket y lo conecta a la dirección IP y puerto especificados
`connect(SocketAddress endpoint)`	Conecta el socket al servidor especificad
`bind(SocketAddress bindpoint)`	Vincula el socket a una dirección local
`getInputStream()`	Devuelve el flujo de entrada para leer datos del socket
`getOutputStream()`	Devuelve el flujo de salida para escribir datos en el socket
`getInetAddress()`	Devuelve la dirección a la que está conectado el socket

`getPort()`	Devuelve el puerto remoto al que está conectado el socket
`getLocalPort()`	Devuelve el puerto local al que está asociado el socket
`close()`	Cierra el socket
`isClosed()`	Comprueba si el socket está cerrado
`isConnected()`	Comprueba si el socket está conectado

Métodos de la clase ServerSocket

Método	Descripción
`ServerSocket(int port)`	Crea un ServerSocket vinculado al puerto especificado
`accept()`	Escucha y acepta una conexión de un cliente, devolviendo un objeto Socket para comunicarse con ese cliente
`close()`	Cierra el ServerSocket y libera los recursos asociados
`bind(SocketAddress endpoint)`	Vincula el ServerSocket a una dirección específica
`setSeTimeout(int timeout)`	Establece el tiempo de espera para aceptar conexiones

Métodos de la clase InetAddress

Método	Descripción
`getLocalHost()`	Devuelve un objeto InetAddress con la información de la máquina local.
`getByName(String host)`	Crea un objeto InetAddress a partir de un nombre de host o una dirección IP proporcionada como cadena
`getAllByName(String host)`	Devuelve un array de objetos InetAddress con todas las direcciones IP asociadas a un nombre de host

`getByAddress(byte[] addr)`	Crea un objeto InetAddress a partir de un array de bytes que representa una dirección IP
`getHostAddress()`	Devuelve la dirección IP como una cadena de texto.
`getHostName()`	Retorna el nombre del host asociado a la dirección IP
`getAddress()`	Devuelve la dirección IP como un array de bytes
`isReachable(int timeout)`	Comprueba si la dirección es alcanzable en el tiempo especificado
`getCanonicalHostName()`	Retorna el nombre canónico completo del host

Ejemplo con socket TCP.

```
/**
 *
 * @author Fran
 */
public class ServidorSocketDistancia {

    static final int Puerto = 2000;

    public ServidorSocketDistancia() {
        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 DataOut-
putStream(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 ServidorSocketDistancia();
    }
}
```

```
public class ClienteSocketDistancia {

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

    public ClienteSocketDistancia() {
        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 ClienteSocketDistancia();
    }
}
```

Métodos de la clase DatagramSocket

Método	**Descripción**
DatagramSocket()	Construye un objeto DatagramSocket.
DatagramSocket(SocketAddress bindaddr)	Construye un objeto DatagramSocket y lo asocia (bind) con la dirección y puerto especificado en bindaddr.
send(DatagramPacket p)	Envía un datagrama
receive(DatagramPacket p)	Recibe un datagrama
close()	Cierra el socket.

Métodos de la clase DatagramPacket

Método	Descripción
DatagramPacket(byte [] buf, int long)	Constructor para datagramas recibidos, buf especifica la cadena dónde se almacena y long la longitud.
DatagramPacket(byte [] buf,int offset, int long)	Constructor para datagramas recibidos, buf especifica la cadena dónde se almacena, long la longitud y offset el desplazamiento dentro de buff.
DatagramPacket(byte [] buf,int offset, int long, InetAddress addr, int port)	Constructor para enviar datagramas, buf especifica la cadena a enviar, long la longitud y offset el desplazamiento dentro de buff, addr es la dirección destino y port el puerto.
DatagramPacket(byte [] buf, int long, InetAddress addr, int port)	Constructor para enviar datagramas, buf especifica la cadena a enviar, long la longitud, addr es la dirección destino y port el puerto.
InetAddress getAddress()	
Byte[] getData()	
Int getLength()	
Int getPort()	
setAddress(InetAddress addr)	
setData(byte [] buff)	
setLength(int long)	
setPort(int port)	

Ejemplo con Sockets UDP.

```
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, cade-
na.length);

            System.out.println("Esperando mensajes..");
            while (true) {
                // Recibe y muestra el mensaje
                sSocket.receive(mensaje);
                String datos = new String(mensaje.getData(), 0, mensa-
je.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());
        }
    }
}
```

```
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());
        }
    }
}
```

DIRECCIONES MULTICAST.

Se usan para enviar un mismo paquete a varios equipos simultáneamente (Ej. Cuando restauramos una imagen). Para esta operación existe un bloque de IPs reservado (224.0.0.0 – 239.255.255.255) En Java tenemos una clase basada en datagramas que permite implementar esta función.

La clase MulticastSocket.

Se usa para enviar paquetes a múltiples destinos simultáneamente. Para esto es necesario establecer un grupo *multicast*, que es un grupo de direcciones IP que comparten el mismo número de puerto. Al enviar un mensaje *multicast*, todos los que pertenezcan al grupo recibirán el mensaje; la pertenencia al grupo es transparente para el emisor.

Métodos de la clase MulticastSocket

Método	Descripción
MulticastSocket()	Constructor por defecto.
MulticastSocket(int port)	Construye un multicast socket y lo conecta al puerto especificado
joinGroup(InetAddress mcaddress)	Une el socket multicast al grupo especificado
joinGroup(SocketAddress mcastaddr, NetworkInterface netIf)	Método actualizado más seguro para unirse a un grupo.
leaveGroup(InetAddress mcaaddress)	Abandona el grupo
send(DatagramPacket p)	Envía un datagrama al grupo
receive(DatagramPacket p)	Recibe del datagrama de un grupo.

El servidor **Multicast** envía paquetes de la siguiente forma:

1- Se crea el socket multicast

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

2- Se crea el grupo *multicast*

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

```
int puerto=1234

String mensaje = "Hola";
```

3- Se crea el datagrama

```
DatagramPacket paquete = new DatagramPacket(mensaje.getBytes(),

            mensajes.length(), grupo, puerto);
```

4- Se envía.

```
ms.send(paquete);
```

El cliente de *multicast* debe unirse al grupo para recibir los mensajes que se envían a este.

1- Se crea el socket *multicast* especificando el puerto de escucha.

```
MulticastSocket ms = new MulticastSocket(1234);
```

2- Se configura la IP del grupo *multicast*

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

3- Se une al grupo

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

El método joinGroup(InetAddress mcastaddr) ya no se recomienda porque no especifica la interfaz de red, lo cual causa problemas en equipos modernos con múltiples tarjetas de red (Wi-Fi + Ethernet + Virtuales).

```
NetworkInterface netIf = NetworkInterface.getByName("eth0"); //
o la interfaz que corresponda

InetSocketAddress group = new
InetSocketAddress(InetAddress.getByName("225.0.0.1"), 1234);

ms.joinGroup(group, netIf);
```

4- Recibe el paquete enviado

```
byte[] buff = new byte[1024];

DatagramPacket recibido = new DatagramPacket(buff, buff.length);

ms.receive(recibido);
```

5- Abandonamos el *multicast*

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

ENVIAR OBJETOS A TRAVÉS DE UNA CONEXIÓN SOCKET STREAM.

Lo único que hay que tener en cuenta es que la clase del objeto que queremos enviar ha de implementar la interfaz **Serializable** y las clases que debemos usar para el envío/recepción deben ser: **ObjectOutput** e **ObjectInputStream**, con sus métodos **writeObject** y **readObject** respectivamente. Una vez recibido el objeto lo podemos castear con su clase u obtener esta con el método **getClass**.

```
class Alumno implements Serializable{
    String nombre;
    Alumno(String n){
        nombre=n;
    }
    public String toString(){
        return nombre;
    }
}
ObjectOutput os = new ObjectOutputStream(clienteSocket.getOutputStream());
os.writeObject(new Alumno("Pepe");
ObjectInputStream is = new ObjectInputStream(s.getInputStream());
System.out.println("Mensaje " + is.readObject());
```

La serialización nativa de Java es considerada hoy en día una **vulnerabilidad de seguridad crítica** y es ineficiente. Oracle planea eliminarla a largo plazo. En el mundo real, **nadie** comunica dos aplicaciones modernas usando `ObjectOutputStream` debido a problemas de compatibilidad entre versiones y riesgos de ataques de ejecución remota de código. En su lugar se utiliza el envío de datos en formato **JSON**.

APLICACIONES CLIENTE/SERVIDOR

En los apartados anteriores hemos visto cómo establecer la comunicación entre dos aplicaciones a través de una red. En los ejemplos una aplicación tenía el rol de servidor y la otra el de cliente.

En este contexto, lo habitual es que haya una aplicación que haga de servidor ofreciendo un servicio a muchas aplicaciones que juegan el papel de clientes. A esta forma de computación se le conoce como modelo Cliente/Servidor, siendo una arquitectura distribuida que permite a los usuarios finales obtener acceso a recursos de forma transparente en entornos multiplataforma.

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.

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.
- 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:

- **El mantenimiento de los sistemas es 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.

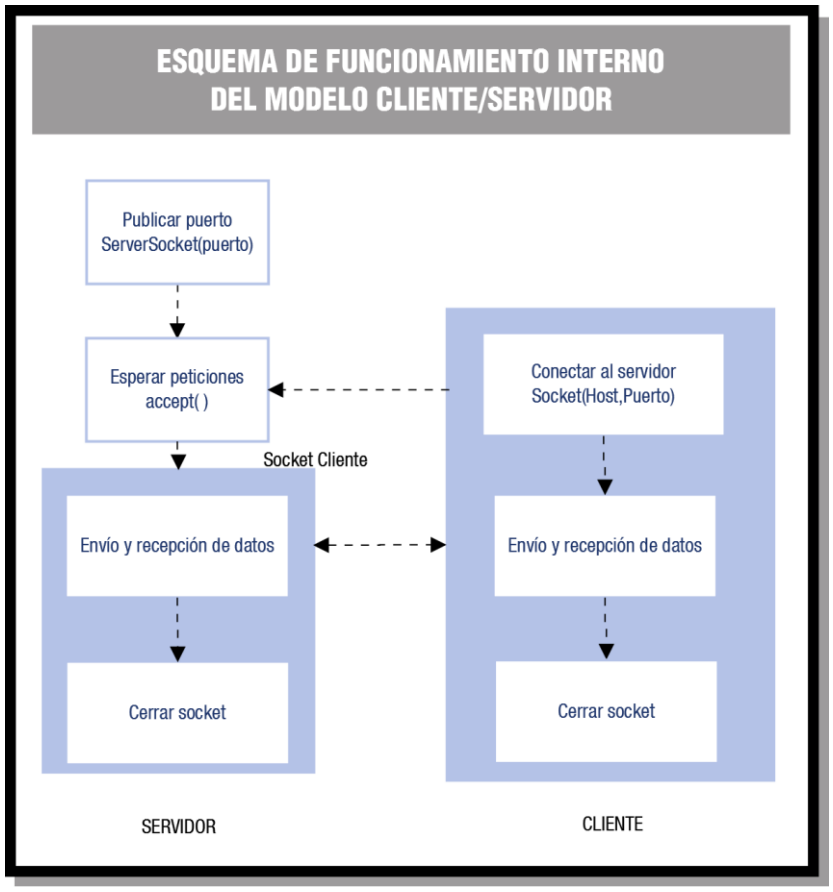
PROGRAMACIÓN DE APLICACIONES CLIENTE/SERVIDOR

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**.



COMUNICACIONES DE ALTO NIVEL CON HTTP

Hasta ahora hemos trabajado con Sockets (TCP/UDP), que son la base de las comunicaciones. Sin embargo, en el desarrollo de aplicaciones modernas, rara vez bajamos a ese nivel. La mayoría de las aplicaciones se comunican utilizando el protocolo **HTTP/HTTPS** para consumir **APIs REST** (servicios web que devuelven datos, generalmente en formato JSON).

Hasta la llegada de **Java 11**, realizar una petición HTTP en Java era tedioso (usando la clase obsoleta HttpURLConnection) o requería librerías externas (como Apache HttpClient). Afortunadamente, Java introdujo un cliente HTTP moderno, potente y estandarizado en el paquete java.net.http.

LAS CLASES IMPLICADAS

El nuevo API se basa en tres componentes principales que funcionan en conjunto:

1. **HttpClient (El Cliente):** Es el "motor" que envía las peticiones. Puede ser configurado con políticas de redirección, tiempos de espera (timeouts), autenticación y versiones de HTTP (1.1 o 2).
    - *Cómo se crea:* Usando HttpClient.newHttpClient() (configuración por defecto) o mediante HttpClient.newBuilder()...build().
2. **HttpRequest (La Petición):** Representa lo que queremos pedir al servidor. Define la URL (URI), el método HTTP (GET, POST, PUT, DELETE) y las cabeceras (Headers). Se construye utilizando el patrón *Builder* (Constructor).
    - *Cómo se crea:* HttpRequest.newBuilder().uri(...).GET().build().

3. **HttpResponse<T> (La Respuesta):** Contiene lo que el servidor nos devuelve: el código de estado (ej. 200 OK, 404 Not Found), las cabeceras de respuesta y el cuerpo del mensaje (el contenido).
    - *El BodyHandler:* Al enviar la petición, debemos decirle a Java cómo tratar el cuerpo de la respuesta (¿lo queremos como un String? ¿como un fichero? ¿como un array de bytes?). Lo más común para texto/JSON es usar HttpResponse.BodyHandlers.ofString().

EJEMPLO PRÁCTICO: CONSUMIENDO UNA API REST

Vamos a crear un ejemplo que realiza una petición **GET** a una API pública gratuita llamada **JSONPlaceholder**, que simula ser una base de datos de usuarios.

Objetivo: Descargar la información del usuario con ID 1 y mostrarla por consola.

```
public class ClienteHttpModerno {

    public static void main(String[] args) {
        // 1. Crear el Cliente HTTP (El navegador virtual)
        // Configuramos un timeout de 10 segundos para no bloquear el programa indefinidamente
        HttpClient cliente = HttpClient.newBuilder()
                .version(HttpClient.Version.HTTP_2)
                .connectTimeout(Duration.ofSeconds(10))
                .build();

        // 2. Construir la Petición (Request)
        // Vamos a pedir datos a una API de prueba
        String url = "https://jsonplaceholder.typicode.com/users/1";

        HttpRequest peticion = HttpRequest.newBuilder()
                .uri(URI.create(url))
                .GET() // Definimos el método (por defecto es GET, pero es mejor ser explícito)
                .header("Content-Type", "application/json") // Indicamos que esperamos JSON
                .build();

        try {
            System.out.println("⏳ Enviando petición a: " + url);

            // 3. Enviar la petición y recibir la Respuesta
            // Usamos send() para modo síncrono (el programa espera aquí a la respuesta)
            // BodyHandlers.ofString() convierte el contenido recibido a texto
            HttpResponse<String> respuesta = cliente.send(peticion, HttpRespon-
se.BodyHandlers.ofString());

            // 4. Procesar la Respuesta
            System.out.println("Estado: " + respuesta.statusCode()); // Ej: 200

            if (respuesta.statusCode() == 200) {
```

```
                System.out.println("✅ Respuesta recibida con éxito:");
                System.out.println(respuesta.body()); // Aquí está el JSON en formato String
            } else {
                System.out.println("❌ Error en la petición: " + respuesta.statusCode());
            }

        } catch (IOException | InterruptedException e) {
            // Manejo de errores de red o interrupción del hilo
            System.err.println("Error durante la comunicación: " + e.getMessage());
            Thread.currentThread().interrupt(); // Buena práctica al capturar InterruptedExcep-
tion
        }
    }
}
```

1 **Configuración del Timeout:** Observa el uso de .connectTimeout(Duration.ofSeconds(10)). En un entorno real de producción, esto es vital. Si la red falla o el servidor remoto se "cuelga", nuestro programa no se quedará congelado para siempre; lanzará una excepción pasados 10 segundos.

2 **URI:** A diferencia de las versiones antiguas que usaban URL, aquí usamos URI (Uniform Resource Identifier), que es más seguro y moderno.

3 **Síncrono vs Asíncrono:** En este ejemplo hemos usado el método .send(), que bloquea la ejecución hasta que llega la respuesta. Para aplicaciones de alto rendimiento o interfaces gráficas que no se deban congelar, HttpClient ofrece el método .sendAsync(), que devuelve un CompletableFuture (promesas), permitiendo que el programa siga funcionando mientras llegan los datos.

El siguiente paso: Procesar el JSON

Si ejecutas el código anterior, verás que respuesta.body() te devuelve una cadena de texto (String) que parece un objeto de JavaScript.

```
{
  "id": 1,
  "name": "Leanne Graham",
  "username": "Bret",
  "email": "Sincere@april.biz",
  ...
}
```

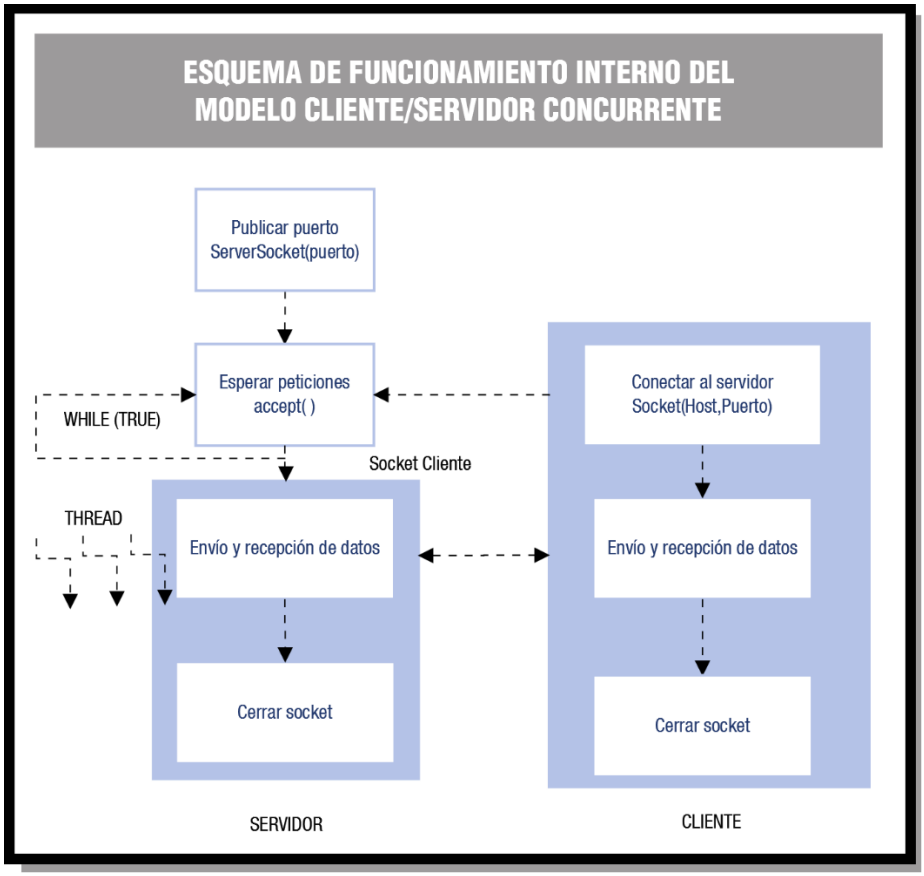
Para trabajar con estos datos en Java (por ejemplo, para hacer usuario.getName()), necesitamos **deserializar** o **parsear** ese String. Como vimos en las recomendaciones de la unidad, **no** se debe hacer manual. Lo profesional es usar una librería como **Jackson** o **Gson** para convertir ese texto automáticamente en un objeto Java (POJO).

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.

En los ejemplos anteriores y en el diagrama anterior el servidor atiende a los clientes de uno en uno por lo que su ejecución es secuencial. Sin embargo, esta ejecución no es adecuada para escenarios donde existan muchos clientes, ya que el tiempo de espera de cada uno de los clientes puede ser muy alto. La solución pasa por atender a cada uno de los clientes en un hilo tal y como se muestra en la siguiente figura.



Para implementar un servicio concurrente, como el descrito, es muy útil usar un pool de *threads*. En Java tenemos la interface **ExecutorService** que nos facilita esta implementación.

Para crear una instancia de **ExecutorService** usaremos:

```
// Instancia el pool de threads
ExecutorService executor = Executors.newCachedThreadPool();
```

Para añadir los *threads* que atienden a los clientes usaremos:

```
// Invoca al método de atención al cliente en un thread de pool de
// threads. Este ejemplo está hecho con una expresión Lambda.
executor.execute(() -> handleClient (clientSocket));

// Invoca al método de atención al cliente en un thread de pool de
// threads.
executor.execute(new HiloClienteSocket(clientSocket));
```

El método **execute** se encarga de poner en funcionamiento el hilo.

Por otro lado, el objeto **executor** dispone de métodos para finalizar los hilos y liberar recursos con diferentes comportamientos:

- `shutdown` inicia un cierre controlado del pool de *threads*. No será posible crear nuevos *threads*, pero se esperará a que los *threads* que actualmente están ejecutándose finalicen.
- `shutdownNow` realiza un cierre forzado, liberando el pool y forzando a que todos los *threads* en ejecución terminen.
- `awaitTermination` realiza un cierre ordenado con temporizador. Si se supera la cantidad de tiempo especificada y aún hay *threads* en ejecución, realiza una finalización forzada de los mismos.

Un esquema típico de un servidor concurrente que gestiona el pool de *threads* mediante un objeto **ExecutorService** sería el siguiente:

```
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Server {
    final static int LOCAL_PORT = 50000;
    final static int BACKLOG = 5000;

    public static void main(String[] args) throws Exception {
        InetAddress loopback = InetAddress.getLoopbackAddress();
        ExecutorService executor = Executors.newCachedThreadPool();
        try {
            //La clase ServerSocket implementa la interface AutoClosable y por tanto se puede
usar
            //en un bloque try-whith-resource ya que invocará de forma automática el método clo-
se.
            try (ServerSocket serverSocket = new ServerSocket(LOCAL_PORT, BACKLOG, loopback)) {
                while (true) {
```

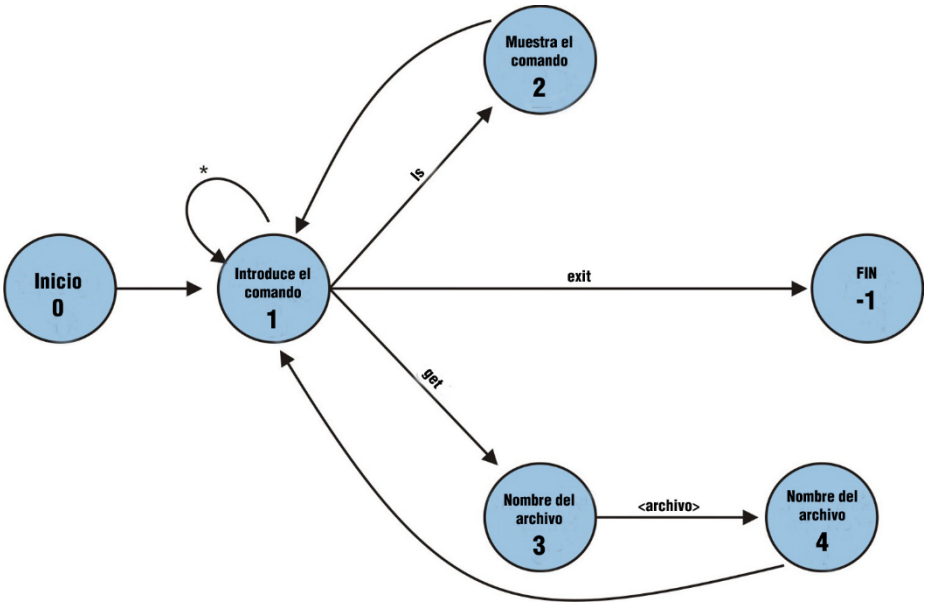
```
                    Socket clientSocket = serverSocket.accept();
                    executor.execute(() -> handleClient(clientSocket));
                }
            }
        } finally {
            if (executor != null)
                executor.shutdown();
        }
    }

    private static void handleClient(Socket clientSocket) {
        // [...] diálogo con el cliente
    }
}
```

TRANSICIONES

Uno de los principales fallos de seguridad que se producen en los 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 autómata**. 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.

Su codificación en Java podría ser así:

```
int estado = 1;

do {
    switch (estado) {

        case 1:
            flujo_salida.writeUTF("Introduce comando (ls/get/exit)");
            comando = flujo_entrada.readUTF();

            if (comando.equals("ls")) {
                System.out.println("\tEl cliente quiere ver el contenido del directorio");
                // Muestro el directorio

                estado = 1;
                break;
            } else if (comando.equals("get")) {
                // Voy al estado 3 para mostrar el fichero
                estado = 3;
                break;
            } else
                estado = 1;
            break;

        case 3:// voy a mostrar el archivo
            flujo_salida.writeUTF("Introduce el nombre del archivo");
            String fichero = flujo_entrada.readUTF();
            // Muestor el fichero
```

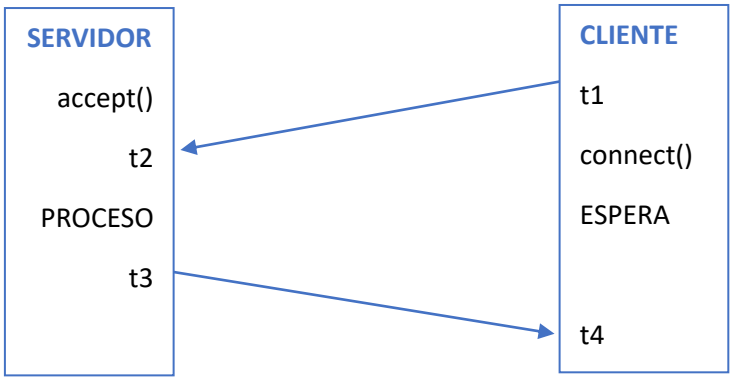
```
                estado = 1;
                break;
            }

            if (comando.equals("exit"))
                estado = -1;
        } while (estado != -1);
```

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 necesitar 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 se deben tomar los tiempos tanto en el cliente como en el servidor usando el **System.currentTimeMillis()**



Tiempo de transmisión petición: t2-t1

Tiempo de procesamiento: t3-t1

Tiempo de transmisión respuesta: t4-t3

Tiempo de respuesta: t4-t1

Lógicamente, cliente y servidor deben estar sincronizados con la misma hora.

BIBLIOGRAFÍA