1. Introducción

Hasta ahora hemos visto cómo diferentes aplicaciones pueden coordinarse para completar una tarea de forma conjunta (**multiprocesamiento**), así como la manera en que un programa puede fraccionar un trabajo en varias partes que se ejecutan en paralelo mediante hilos (**multihilo**). Todo ello ocurre dentro de una misma máquina, ya sea con uno o varios procesadores, bajo un solo sistema operativo y compartiendo normalmente tanto memoria como dispositivos de E/S.

En esta nueva unidad se avanza hacia el desarrollo de programas capaces de operar en sistemas distribuidos. Aunque seguimos trabajando con múltiples procesos, la diferencia con las unidades anteriores es que ya no existe una relación jerárquica entre ellos. Ahora se ejecutan en equipos separados y se comunican entre sí a través de la red empleando protocolos de comunicación.

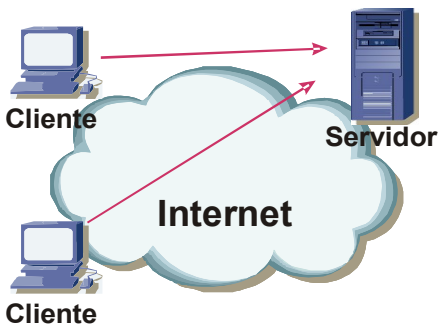
Para desarrollar aplicaciones que permita establecer conexiones y comunicarse a través de una red de equipos, no se va a partir desde cero. **Java** ofrece clases para establecer conexiones, crear servidores, enviar y recibir datos y para muchas otras operaciones utilizadas en las comunicaciones a través de una red de ordenadores.

Además, la programación en red está estrechamente relacionada con la programación multiproceso, especialmente en lo que respecta a los mecanismos de comunicación entre procesos que ya hemos estudiado.

Además, la capacidad de un servidor para atender simultáneamente a varios clientes se apoya en la utilización de hilos para repartir el trabajo.

Por todo ello, los conceptos y habilidades que hemos aprendido hasta ahora constituyen la base que nos permitirá profundizar en los contenidos de esta unidad.

1.1. Comunicación entre aplicaciones. Modelos.



La interacción entre aplicaciones es muy importante para lograr que diferentes sistemas trabajen juntos de un modo efectivo. Ya vimos en la unidad 1 que la comunicación puede establecerse usando diferentes modelos:

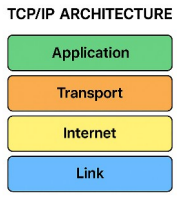
- Modelo de memoria compartida.
- Modelo basado en el intercambio de mensajes.
- Modelo basado en APIs y Web Services. Uso de APIs de REST para la comunicación entre aplicaciones mediante HTTP. Por ejemplo, una app móvil que consume datos de una API REST.
- Modelo basado en eventos. Una aplicación emite eventos y otras los consumen.
- Modelo de comunicación mediante intercambio de ficheros.

Además de estos, otros dos grandes modelos en los que los sistemas distribuidos suelen clasificarse son:

- **Cliente–Servidor**: un proceso servidor proporciona servicios a uno o más procesos clientes.
- **Punto a Punto (P2P)**: todos los procesos participan de manera equivalente, colaborando juntos sin que exista una función especializada para ninguno de ellos.

En la presente unidad, de todos estos modelos **nos centraremos en las aplicaciones cliente-servidor usando Socket de Java**.

1.2. Arquitectura TCP/IP



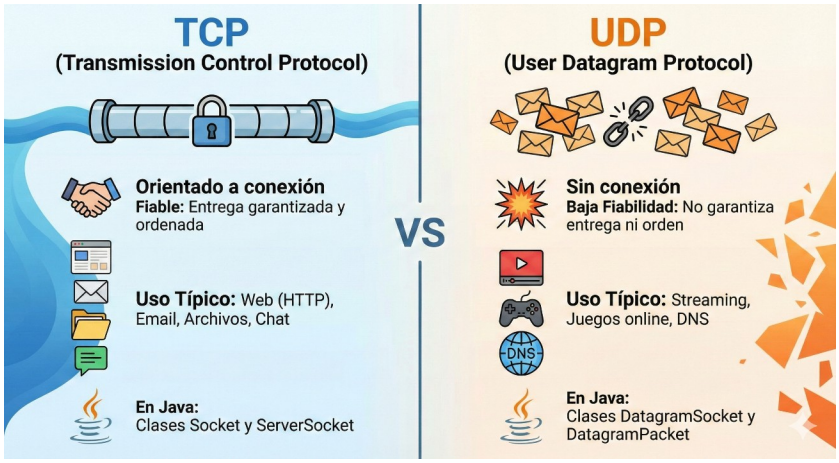
TCP/IP es una familia de protocolos desarrollados para permitir la comunicación entre cualquier par de ordenadores de cualquier red o fabricante. Consta de **cuatro capas**, cada una de las cuales, está encargada de resolver un subconjunto de los problemas específicos que encontramos cuando queremos realizar una comunicación de datos entre equipos en una red.

Cuando se desarrollan aplicaciones Java que se comunican a través de la red, se está programando en la capa de aplicación.

Un aspecto importante a la hora de trabajar en esta arquitectura es si la capa de transporte a utilizar es orientada a la conexión (**Protocolo TCP**) o no (**Protocolo UDP**):

- **TCP**: Basado en la conexión, garantiza que los datos enviados (**segmentos**) desde un extremo de la conexión llegan al otro extremo y en el mismo orden en el que fueron enviados. De no ser así, se notifica un error.
- **UDP**: No basado en la conexión, por lo que los datos se envían de forma independiente, sin garantizar ni la recepción de los paquetes enviados (**datagramas**) ni el orden en el que se entregan (no es importante).

Por tanto, dado que las aplicaciones que se van a desarrollar están en la capa de aplicación, **lo primero que habrá que hacer es identificar el escenario de comunicación, eligiendo el protocolo adecuado**.



2. API de Java para comunicaciones en red.

Java proporciona un conjunto robusto de librerías para programar aplicaciones en red. El paquete principal es java.net.

El **paquete** java.net**, se puede dividir en dos niveles:**

1) **Una API de bajo nivel**, que permite representar los siguientes objetos:
   - ✔ **Direcciones**. Son los identificadores de red, esto es, las direcciones IP.
     - Clase InetAddress. Implementa una dirección IP.
   - ✔ **Sockets**. Son los mecanismos básicos de comunicación bidireccional de datos.
     - Clase Socket. Implementa un extremo de una conexión bidireccional.
     - Clase ServerSocket. Implementa un socket que los servidores pueden utilizar para escuchar y aceptar peticiones de clientes.
     - Clase DatagramSocket. Implementa un socket para el envío y recepción de datagramas.
     - Clase MulticastSocket. Representa un socket datagrama, útil para enviar paquetes multidifusión.
   - ✔ **Interfaces.** Describen las interfaces de red.
     - ✔ Clase NetworkInterface. Representa una interfaz de red compuesta por un nombre y una lista de direcciones IP asignadas a esta interfaz.
2) **Una API de alto nivel**, que se ocupa de representar los siguientes objetos, mediante las clases:
   - ✔ **URI**. Representan los identificadores de recursos universales.
     - Clase URI.
   - ✔ **URL.** Representan localizadores de recursos universales.
     - Clase URL. Representa una dirección URL.
   - ✔ **Conexiones.** Representa las conexiones con el recurso apuntado por URL.
     - Clase URLConnection. Es la superclase de todas las clases que representan un enlace de comunicaciones entre la aplicación y una URL.
     - Clase HttpURLConnection. Representa una URLConnection con soporte para HTTP y con ciertas características especiales.

3. Métodos y ejemplos de uso de InetAddress

La clase InetAddress proporciona objetos que se pueden utilizar para manipular tanto direcciones IP como nombres de dominio. También proporciona métodos para averiguar los nombres de host a partir de sus direcciones IP y viceversa.

Una instancia de InetAddress consta de una dirección IP y en algunos casos también del nombre de host asociado. Esto último depende de si se ha creado con el nombre de host o bien ya se ha aplicado la resolución de nombres.

Esta clase **no tiene constructores**, **dispone de métodos estáticos que devuelven objetos** InetAddress (patrón de diseño Factory Method). Por tanto, se usarán métodos estáticos para crear instancias ("fabricar" objetos InetAddress) y métodos de instancia para obtener información.

Para crear instancias (Métodos Estáticos):

- getByName(String host)

  Devuelve un objeto de tipo InetAddress donde el parámetro *host* tiene que ser el nombre DNS del host (iesalandalus.org), una IP válida (195.78.228.161), o de la red de área local (como documentos.servidor o 192.168.0.5). Incluso puedes poner localhost.

- getLocalHost()

  Devuelve un objeto de tipo InetAddress con los datos de direccionamiento de la máquina donde se está ejecutando la aplicación. (no del conocido localhost).

- getAllByName(String host)

  Devuelve un array de objetos de tipo InetAddress con los datos de direccionamiento del host pasado como parámetro. Recuerda que en Internet es frecuente que un mismo dominio tenga a su disposición más de una IP. Muy útil porque grandes servicios (como Google o Amazon) tienen muchas IPs asociadas a un solo nombre de dominio para balancear la carga.

Todos estos métodos pueden generar una excepción UnknownHostException si no pueden resolver el nombre pasado como parámetro.

Para obtener información (Métodos de Instancia):

- getHostAddress(). Devuelve de un objeto InetAddress la dirección IP en formato texto (String), ej: "192.168.0.10".
- getHostName(). Devuelve el nombre del host de un objeto InetAddress (si puede resolverlo).
- getAddress(). Devuelve un array formado por los grupos de bytes de la IP correspondiente.
- isReachable(int tiempo). Devuelve **true** o **false** dependiendo de si la dirección es alcanzable en el tiempo indicado en el parámetro. El tiempo es indicado en milisegundos.

¿Qué utilidad tiene InetAddress para las comunicaciones en red?:

- En el Cliente: Cuando se crea un Socket, se necesita pasar un objeto InetAddress para decirle a dónde conectarse.
- En el Servidor: Cuando se acepta una conexión se dispone de un método getInetAddress() que permite saber quién se ha conectado al servidor. Esto es vital para logs de seguridad o para filtrar accesos (ej: "solo acepto conexiones de la IP 192.168.1.50").

Ver APIInetAddress