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.

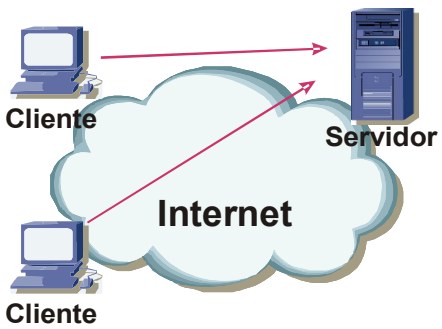
Para desarrollar aplicaciones que permita establecer conexiones y comunicarse a través de una red de equipos, no se va a partir desde cero. **Java** ofrece clases para establecer conexiones, crear servidores, enviar y recibir datos y para muchas otras operaciones utilizadas en las comunicaciones a través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

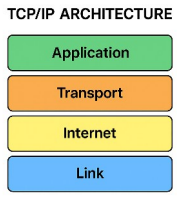
- Modelo de memoria compartida.
- Modelo basado en el intercambio de mensajes.
- Modelo basado en APIs y Web Services. Uso de APIs de REST para la comunicación entre aplicaciones mediante HTTP. Por ejemplo, una app móvil que consume datos de una API REST.
- Modelo basado en eventos. Una aplicación emite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



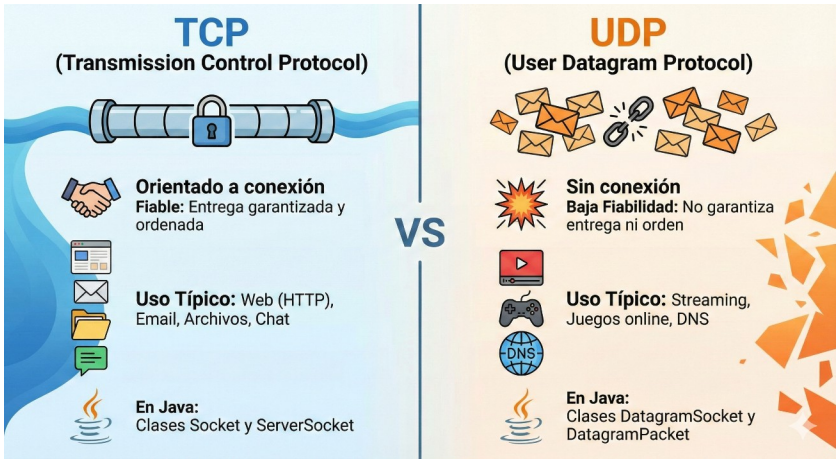
TCP/IP es una familia de protocolos desarrollados para permitir la comunicación entre cualquier par de ordenadores de cualquier red o fabricante. Consta de **cuatro capas**, cada una de las cuales, está encargada de resolver un subconjunto de los problemas específicos que encontramos cuando queremos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

4. La clase URL

Una URL, Localizador Uniforme de Recursos, representa una dirección a un recurso de la World Wide Web. Un **recurso** puede ser algo tan simple como un archivo o un directorio, o puede ser una referencia a un objeto más complicado, como una consulta a una base de datos, el resultado de la ejecución de un programa, etc.

Consideraciones sobre una URL:

- Puede referirse a sitios web, ficheros, sitios ftp, direcciones de correo electrónico, etc.
- La **estructura de una URL** se puede dividir en varias partes, tal y como puede ver en la imagen:

  - **Protocolo**. El protocolo que se usa para comunicar.
  - **Nombrehost**. Nombre del host que proprociona el servicio o servidor.
  - **Puerto**. El puerto de red en el servidor para conectarse. Si no se especifica, se utiliza el puerto por defecto para el protocolo.
  - **Ruta**. Es la ruta o path al recurso en el servidor.
  - **Referencia**. Es un fragmento que indica una parte específica dentro del recurso especificado.

La clase URL de Java es la que permite representar un URL. Utilizando la clase URL, se puede establecer una conexión con cualquier recurso que se encuentre en Internet. Una vez establecida la conexión, invocando los métodos apropiados sobre ese objeto URL se puede obtener el contenido del recurso en el cliente.

Sin embargo, los objetos de tipo URL no se van a instanciar usando **los constructores de dicha clase dado que están deprecated**. Se usará para ello la clase URI, y cuando se vaya a establecer la conexión, se convierte el objeto URI a un objeto URL usando el método toURL().

En definitiva, se usará URI para fabricar el objeto y validarlo, y luego convertirlo a URL solo para abrir la conexión.

4.1. Constructores de la clase URI

La clase java.net.URI tiene varios constructores, pero los más importantes son tres:

1. public URI(String str)

   Este es el más básico. Asume que la cadena de texto que se le pasa **está perfectamente formateada y codificada**.

   ```
   URI uri = new URI("https://www.ejemplo.com/ruta/recurso");
   ```

   ¿Qué hace? Analiza la cadena para ver si cumple la gramática RFC 2396.

   ¿Qué NO hace? No codifica (encode) nada. Si le pasas un espacio, una tilde o una ñ, lanzará una excepción inmediatamente.

   ```
   //VÁLIDO: La cadena es correcta
   URI uri1 = new URI("https://www.google.com/search?q=java");


   //ERROR: Contiene un espacio en "vacaciones verano"
   //Lanza URISyntaxException: Illegal character in path at index…
   URI uri2 = new URI("https://miservidor.com/fotos/vacaciones verano.jpg");
   ```

2. URI(String scheme, String ssp, String fragment)

   scheme: protocolo → "http", "https", "ftp", etc.

   ssp: scheme-specific part (todo lo que va después del :\\)

   fragment: lo que va después de #

   Por ejemplo, para https://www.ejemplo.com/ruta/recurso#seccion1

   ```
   URI uri = new URI("https", "www.ejemplo.com/ruta/recurso", "seccion1");
   ```

3. URI(String scheme, String host, String path, String fragment)

   El más común para descargar ficheros directos.

   Por ejemplo, para https://www.ejemplo.com/productos/lista#top

   ```
   URI uri = new URI("https", "www.ejemplo.com", "/productos/lista", "top");
   ```

4. URI(String scheme, String userInfo, String host, int port, String path, String query, String fragment)

   Es el más completo. Se usa si se necesita especificar puerto (ej: 8080) o parámetros de consulta (query). Si algún dato no se tiene (ej: no tienes fragmento), se pone null.

   userInfo: usuario y contraseña.
   port: puerto
   query: parámetros de consulta (?param=123)

Por ejemplo, para https://user:pass@www.ejemplo.com:8080/api/items?id=10&sort=asc

```
URI uri = new URI(
    "https",                    // scheme
    "user:pass",                // userInfo
    "www.ejemplo.com",          // host
    8080,                       // puerto
    "/api/items",               // path
    "id=10&sort=asc",           // query
    null                        // fragment
);
```

// Queremos: http://localhost:8080/api/buscar?q=programación java

```
URI compleja = new URI("http", null, "localhost", 8080, "/api/buscar", "q=programación java",
null);
// Salida automática: http://localhost:8080/api/buscar?q=programaci%C3%B3n%20java
```

Todos estos constructores pueden lanzar la excepción URISyntaxException si la URL está mal construida. No se hace ninguna verificación de que realmente exista la máquina o el recurso en la red.

4.2. Métodos de la clase URL

Para crear un objeto URL se debe usar el método toURL() de la clase URI. Este método puede lanzar la excepción MalformedURLException.

```
URI uri=new URI("http://www.iesalandalus.org");

URL url=uri.toURL();
```

Los métodos de la clase URL se pueden clasificar en dos grandes grupos:

1. **Métodos de Inspección** (getters): Para "despiezar" la dirección y ver sus partes.
2. **Métodos de Acción**: Para conectarse a internet y traer datos.

Métodos de Inspección

Vamos a poner de ejemplo la siguiente URL cargada en tu objeto:

https://tienda.com:8080/ofertas/verano?id=55#precio

Método	Descripción	Ejemplo de retorno
String getProtocol()	Devuelve el esquema de comunicación.	https
String getHost()	Devuelve el nombre del servidor o IP.	tienda.com
int getPort()	Devuelve el puerto **explícito**. **OJO:** Si la URL no tiene puerto escrito (ej: https://www.google.com/search?q=google.com), devuelve **-1**.	8080
int getDefaultPort()	Devuelve el puerto estándar del protocolo (útil cuando getPort da -1).	443 (para https)

String getPath()	La ruta al recurso (carpetas y archivo)	/ofertas/verano
String getQuery()	La cadena de consulta (parámetros)	id=55
String getRef()	La referencia o ancla (lo que va tras el #)	precio
String getFile()	Devuelve Path + Query	/ofertas/verano?id=55
String getAuthority()	Devuelve la autoridad del objeto URL	tienda.com
String getUserInfo()	Devuelve el user y password utilizados	null

Ver: UriUrlInspeccion

Métodos de Acción

Método	Descripción	Cuándo usarlo	Ejemplo
InputStream openStream()	Abre una conexión, redirige, y te devuelve directamente un **InputStream** para leer los datos brutos (bytes).	Cuando solo quieres **descargar** el contenido (leer el HTML de una web, bajar una imagen) y no te importa nada más.	`InputStream entrada = miUrl.openStream();`
URLConnection openConnection()	Devuelve un objeto URLConnection (o HttpURLConnection), **todavía no conecta**.	Necesitas saber el **tamaño** del fichero antes de bajarlo. Necesitas leer **cabeceras** (cookies, tipo MIME). Necesitas **enviar** datos (POST) o escribir en un formulario.	`URLConnection conexion = miUrl.openConnection();`

Ver UriUrlAccion

4.3. Clase URLConnection

Una vez que se tiene un objeto de la clase URL, se invoca al método OpenConnection() para realizar la comunicación con el objeto y si la conexión se establece correctamente, se tiene una instancia de un objeto URLConnection:

```
URI uri=new URI("http://www.iesalandalus.org");

URL url=uri.toURL();

URLConnection urlCon=URL.openConnection();
```

URLConnection es una clase abstracta que contiene métodos que permiten la comunicación entre la aplicación implementada y la URL.

Las instancias de este objeto se pueden usar tanto para leer como para escribir al recurso referenciado por la URL. Tener en cuenta que se puede lanzar una excepción IOException.

Método	Retorno	Descripción
connect()	void	Establece la conexión con el recurso remoto si tal conexión no se ha establecido ya.
getInputStream()	InputStream	Devuelve un objeto InputStream para leer datos de esta conexión.
setDoInput(boolean doinput)	void	Habilita lectura desde la conexión. Permite que el usuario reciba datos desde la URL si el parámetro es true (por defecto está establecido a true).
getDoInput()	boolean	Indica si la lectura está habilitada
getOutputStream()	OutputStream	Devuelve un objeto OuputStream para escribir datos en esta conexión.
setDoOutput(boolean dooutput)	void	Habilita escritura hacia la conexión. Permite que el usuario envíe datos si el parámetro es true (no está establecido al principio).
getDoOutput()	boolean	Indica si la escritura está habilitada
getContent()	Object	Obtiene el contenido del recurso
getContent(Class[] classes)	Object	Devuelve el contenido como una de las clases indicadas
getContentLength()	int	Tamaño del contenido en bytes. Devuelve el valor del campo de cabecera content-lenght o -1 si no está definido.
getContentLengthLong()	long	Tamaño del contenido (long)
getContentType()	String	Tipo MIME del contenido. Devuelve el valor del campo de cabecera content-type o null si no está definido.
getContentEncoding()	String	Codificación del contenido
getExpiration()	long	Fecha de expiración del recurso
getDate()	long	Fecha del recurso. Devuelve el valor del campo de cabecera date o 0 si no está definido.
getLastModified()	long	Última modificación del recurso. Devuelve el valor del campo de cabecera last-modified.
getHeaderField(int n)	String	Devuelve el valor del header en la posición **n.** Devuelve el valor del campo enésimo campo de cabecera especificado o null si no está definido.
getHeaderField(String name)	String	Devuelve el valor del header por nombre
getHeaderFieldKey(int n)	String	Devuelve el nombre del header en la posición **n**

Método	Retorno	Descripción
getHeaderFields()	Map<String, List<String>>	Devuelve todos los headers. Devuelve una estructura Map con los campos de la cabecera. Las claves son cadenas que representan los nombres de los campos de la cabecera y los valores son cadenas que representan los valores de los campos correspondientes.
getPermission()	Permission	Devuelve los permisos necesarios para la conexión
getURL()	URL	Devuelve la URL asociada
setAllowUserInteraction(boolean allow)	void	Permite interacción con el usuario
getAllowUserInteraction()	boolean	Indica si se permite interacción
setUseCaches(boolean usecaches)	void	Habilita el uso de caché
getUseCaches()	boolean	Indica si se usa caché
setDefaultUseCaches(boolean defaultusecaches)	void	Define uso de caché por defecto
getDefaultUseCaches()	boolean	Devuelve el uso de caché por defecto
setIfModifiedSince(long ifmodifiedsince)	void	Envía fecha de modificación para validación
getIfModifiedSince()	long	Devuelve la fecha enviada
setRequestProperty(String key, String value)	void	Añade una propiedad de request
addRequestProperty(String key, String value)	void	Añade sin reemplazar valores existentes
getRequestProperty(String key)	String	Devuelve una propiedad del request
getRequestProperties()	Map<String, List<String>>	Devuelve todas las propiedades del request
getFileNameMap()	FileNameMap	Devuelve el mapa de tipos MIME

Ver: URLConnectionTest1; URLConnectionTest2

Desde Java y usando la clase URLConnection se puede interactuar con scripts del lado del servidor para enviar valores a los campos del script sin necesidad de abrir un formulario HTML, enviando los datos al script a través del objeto URLConnection. Por tanto, la aplicación deberá hacer los siguiente:

1. Crear el objeto URL al script con el que se quiere interactuar.
2. Abrir una conexión con la URL usando el método openConnection().
3. Configurar la conexión para que se puedan enviar datos usando el método setDoOutput().
4. Obtener un stream de salida sobre la conexión usando el método getOutputStream():

```
PrinterWriter output=new PrinterWriter(conexion.getOutputStream());
```

5. Escribir en el stream de salida para enviar los parámetros junto con los valores. En este caso se manda una cadena con los datos que necesita el script:

   ```
   output.write(cadena);
   ```

   donde cadena tiene el siguiente formato: parametro=valor. Si el script recibe varios parámetros se deben separar por el carácter &: parametro1=valor1&parametro2=valor2…

6. Cerrar el strem de salida: `output.close();`

Normalmente cuando se pasa información a algún script PHP, éste realiza una acción y envía información devuelta por la misma URL. Por tanto, si se quiere ver lo que devuelve será necesario leer desde la URL. Por tal motivo, se debe abrir un stream de entrada sobre esa conexión mediante el método getInputStream() y después realizar la lectura para obtener los resultados devueltos por el script.

Ver: URLConnectionTest3

5. Sockets

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

Los protocolos TCP y UDP usan el concepto de sockets para proporcionar los extremos de una comunicación entre aplicaciones o procesos. La comunicación entre procesos consiste en la transmisión de un mensaje entre un conector de un proceso y un conector de otro proceso, a este conector es a lo que se le llama socket.

Por lo tanto, la conexión puede identificarse con una tupla formada por cuatro números: la dirección IP de origen, la dirección IP de destino, el puerto de origen y el puerto de destino.

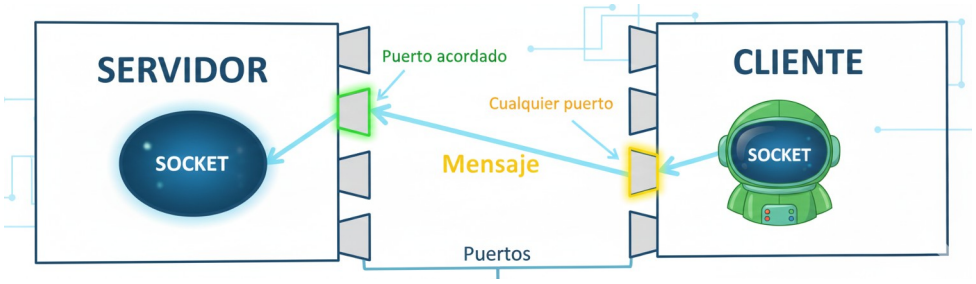
Para los procesos receptores de mensajes, su conector debe tener asociado dos campos:

1. La dirección IP del host en el que la aplicación está corriendo.
2. El puerto local a través del cual la aplicación se comunica y que identifica el proceso.

Esto permite que múltiples aplicaciones cliente ejecutándose en máquinas diferentes puedan conectarse al mismo socket de destino en el servidor. Tampoco hay confusión de a qué equipo se debe enviar una respuesta, incluso si el puerto de origen y destino es el mismo.

Usando sockets también es posible tener varias aplicaciones cliente corriendo en el mismo equipo y conectándose al mismo servidor (varias pestañas de un navegador). Las respuestas enviadas por el servidor al cliente contienen la información del socket en el lado del cliente, la cual incluye el puerto asignado individualmente a cada uno de los clientes, no pudiendo haber confusión sobre a qué proceso entregar la respuesta.

Así, todos los mensajes enviados a esa dirección IP y a ese puerto concreto llegarán al proceso receptor.



En la imagen anterior se muestra un proceso cliente (envía un mensaje) y un proceso servidor (recibe un mensaje) comunicándose mediante sockets. Cada socket tiene un puerto asociado, el proceso cliente debe conocer el puerto y la IP del proceso servidor. Los mensajes al servidor le deben llegar al puerto acordado. El proceso cliente podrá enviar el mensaje por el puerto que quiera.

Los procesos pueden usar el mismo conector tanto pare enviar como para recibir mensajes. Además, cada conector se asocia a un protocolo que puede ser UDP o TCP.

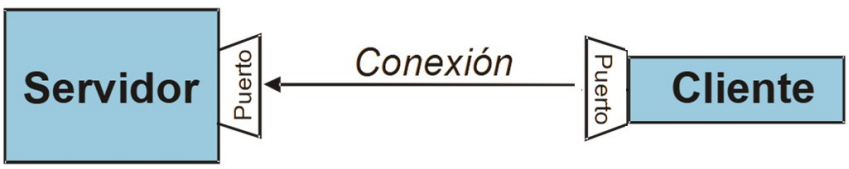
1) TCP: es un protocolo orientado a la conexión que permite que un flujo de bytes originado en una máquina se entregue sin errores en cualquier máquina destino. Es una conexión fiable en la que se garantiza la entrega de los paquetes de datos y el orden en que fueron enviados. Utiliza un esquema de acuse de recibo de los mensajes de tal forma que si el emisor no recibe dicho acuse en un tiempo determinado, vuelve a transmitir el mensaje.

2) UDP: no orientado a conexión, se utiliza para comunicaciones donde se prioriza la velocidad sobre la pérdida de paquetes, o en los casos en los que se desea enviar tan poca información que cabe en un único datagrama. Es una conexión no fiable en la que no se garantiza que la información enviada llegue al destino, ni tampoco se garantiza que lleguen en el orden en el que fueron enviados. No hay acuse de recibo ni reintentos. Se usa, por ejemplo, para la transmisión de audio y vídeo en tiempo real.

5.1. Funcionamiento general de un socket

Un puerto es un punto de destino que identifica hacia qué aplicación o proceso deben dirigirse los datos. En un aplicación cliente-servidor, el programa servidor se ejecuta en una máquina concreta (con una IP) y tiene un socket asociado a un número de puerto específico. El servidor queda a al espera “escuchando” las solicitudes de conexión de los clientes.

La aplicación cliente debe conocer el nombre del servidor y el número de puerto por el que escucha las peticiones. Para solicitar la conexión, el cliente realiza la petición al servidor a través del puerto. Además, el cliente debe identificarse al servidor por lo que durante la conexión se usará un puerto local asignado por el sistema.



Si todo va bien, el servidor acepta la conexión. Una vez aceptada, el sistema operativo crea un nuevo socket con:

- IP servidor + puerto
- IP cliente + puerto efímero del cliente

Ese nuevo Socket es exclusivo para ese cliente.

En definitiva, el servidor no "cambia de puerto", sino que:

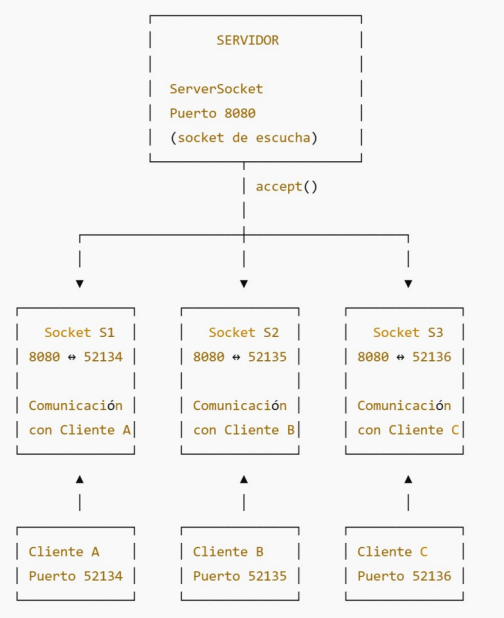
- Usa un socket de escucha (puerto conocido) solo para aceptar conexiones
- Y crea un nuevo socket por cada cliente para comunicarse con él.

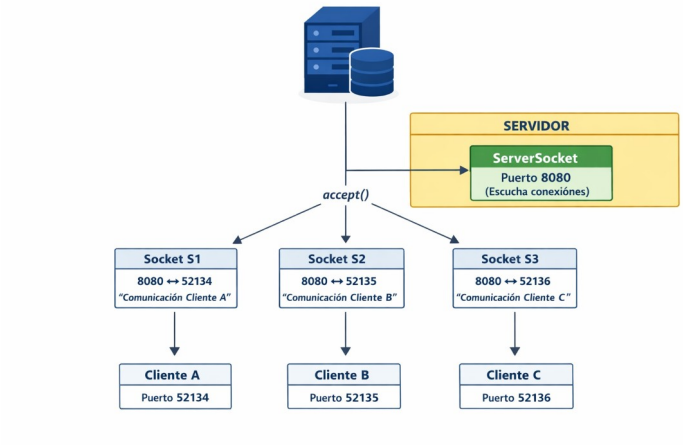
Esto se debe a que por un lado debe seguir atendiendo las peticiones de conexión mediante el socket original y por otro debe atender las necesidades del cliente que se conectó.

De este modo se pueden atender a múltiples clientes simultáneamente y mantener separadas las conexiones.

¿Por qué no usar el mismo socket? Si el servidor usara el mismo socket para todos los clientes:

- No podría distinguir qué datos vienen de qué cliente.
- No podría atender clientes simultáneamente.
- Una conexión bloquearía a las demás.
- Se mezclarían flujos de datos.





5.2. Clases para sockets TCP

Para trabajar con sockets TCP Java proporciona las clases:

- ServerSocket, que permite a un servidor escuchar y recibir las peticiones de los clientes por la red.
- Socket permite a un cliente conectarse a un servidor para enviar y recibir información.

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

5.2.1. Clase Socket

Esta clase implementa un extremo de la conexión TCP. Algunos de sus constructores son:

Constructor	Descripción
Socket()	Crea un Socket sin ningún puerto asociado.
Socket(String host, int port)	Crea un socket y lo conecta al número de puerto y al nombre de host especificados. Puede lanzar UnknownHostException e IOException.
Socket(InetAddress address, int port)	Crea un socket y lo conecta al puerto y dirección establecida a través del objeto InetAddress.
Socket(String host, int port, InetAddress localAddr, int localPort)	Idem al anterior pero estableciendo la dirección y el puerto local desde el que realizar la conexión.
Socket(InetAddress address, int port, InetAddress localAddr, int localPort)	Idem al anterior pero estableciendo la conexión remota a través de un objeto InetAddress.

¿Qué ocurre si trato de usar como localPort un puerto que el Sistema Operativo ya tiene asignado a otro socket? Supongamos que el puerto 8080 ya está ocupado y creamos un conector socket en el cliente:

```
Socket socket = new Socket(
    "example.com",
    80,
    InetAddress.getByName("127.0.0.1"),
    8080
);

Exception in thread "main" java.net.BindException: Address already in use
```

Por qué ocurre (a nivel del SO):

- Cada puerto local solo puede estar asociado a un socket activo.
- El Sistema Operativo mantiene una tabla con (IP local, puerto local, protocolo)
- Si ese puerto ya está ocupado, no puede reutilizarse (Aunque el puerto remoto sea distinto, el conflicto es local).

Por tanto, el Sistema Operativo protege la integridad de las conexiones
¿Qué se recomienda hacer?

✔ No especificar localPort, dejar que el Sistema Operativo asigne uno efímero:

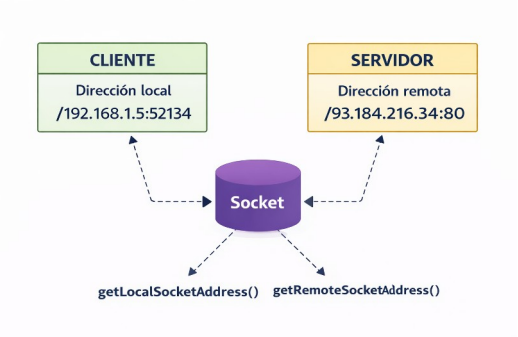
```
Socket socket = new Socket("example.com", 80);
```

✔ O usar localPort = 0 (puerto automático). El Sistema Operativo elegirá un puerto libre.

```
Socket socket = new Socket("example.com",80,InetAddress.getByName("127.0.0.1"),0);
```

Otros métodos a destacar son:

Método	Retorno	Descripción
close()	void	Cierra el socket
getInetAddress()	InetAddress	IP remota. Devuelve un objeto InetAddress con todo el direccionamiento asociado al socket al que el cliente está conectado. Si no lo está devuelve null.
getLocalAddress()	InetAddress	Idem que el anterior con todo el direccionamiento asociado al socket desde el que el cliente está conectado.
getPort()	int	Devuelve el puerto remoto al que está enlazado el socket o 0 si no está conectado a ningún puerto.
getLocalPort()	int	Devuelve el puerto local al que está enlazado el socket o -1 si no está enlazado a ningún puerto.
getRemoteSocketAddress()	SocketAddress	Dirección remota completa (/93.184.216.34:80)
getLocalSocketAddress()	SocketAddress	Dirección local completa (/192.168.1.5:52134)
getInputStream()	InputStream	Stream de entrada. Devuelve un InputStream que permite leer bytes desde el socket utilizando los mecanismos de streams. El socket debe estar conectado. Puede lanzar IOException.
getOutputStream()	OutputStream	Stream de salida. Devuelve un OutputStream que permite escribir bytes sobre el socket utilizando los mecanismos de streams. El socket debe estar conectado. Puede lanzar IOException.



En resumen, los pasos que realiza el cliente para realizar una comunicación son:

1. **Conectarse con el servidor**.

   El cliente utiliza un constructor *Socket* para conectarse a un determinado servidor y a un puerto específico. Una vez realizada la conexión se crea el socket por donde se realizará la comunicación.

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

2. **Envío y recepción de datos**.

   Para poder recibir/enviar datos es necesario crear un flujo (stream) de entrada y otro de salida.

3. Una vez finalizada la comunicación se **cierra el socket**

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

5.2.2. Clase ServerSocket

Algunos de los constructores de la esta clase son:

Constructor	Descripción
ServerSocket()	Crea un socket sin ningún puerto asociado.
ServerSocket(int port)	Crea un socket que se enlaza al puerto especificado.
ServerSocket(int port, int max)	Crea un socket que se enlaza al puerto especificado. El parámetro max especifica cuántas conexiones entrantes pendientes de accept() puede almacenar el sistema operativo antes de rechazarlas. Solo controla cuántos clientes pueden esperar antes de que sean rechazados temporalmente. Ejemplo: 3 → Hasta 3 clientes esperando pueden estar en cola. Si el parámetro max es 0 o negativo, el Sistema Operativo asigna un valor por defecto.
ServerSocket(int port, int max, InetAddress direc)	Idem al constructor anterior pero establece la dirección IP local a la que se "ata" el servidor. Solo escuchará conexiones que lleguen a esta IP. Si se usa null, escucha en todas las interfaces (0.0.0.0).

En definitiva, los tres primeros constructores crean objetos ServerSocket que escuchan en todas las interfaces de red. En cambio, el último constructor crea objetos ServerSocket que solo escucha en la interfaz de red establecida a través del objeto InetAddress pasado como parámetro.

En el siguiente fragmento de código se muestra un ejemplo del uso del constructor estableciendo una cola máxima de peticiones en espera.

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

            // Crea un servidor en el puerto 5000 con max 10
            ServerSocket server = new ServerSocket(5000, 10);
            System.out.println("Servidor escuchando en puerto: " + server.getLocalPort());


            while (true) {
                Socket client = server.accept(); // acepta conexiones
                System.out.println("Cliente conectado: " + client.getRemoteSocketAddress());
                client.close();
            }

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

En el siguiente fragmento de código se muestra un ejemplo del uso del constructor estableciendo una cola máxima de peticiones en espera y una interfaz a la que se ata el servidor para solo escuchar peticiones de dicha interfaz de red (127.0.0.1).

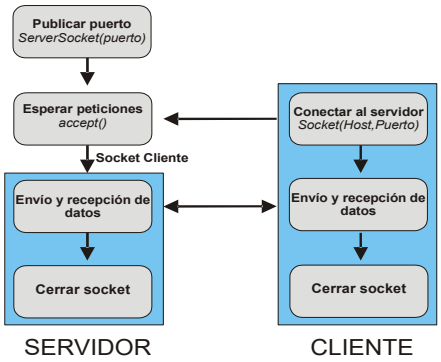
```
public class ServerBindAddrExample
{
    public static void main(String[] args) {
        try {
            InetAddress localAddr = InetAddress.getByName("127.0.0.1"); // solo localhost
            int port = 5003;
            int max = 5;

            ServerSocket server = new ServerSocket(port, max, localAddr);
            System.out.println("Servidor escuchando en " + server.getInetAddress() +
                               " puerto " + server.getLocalPort() +
                               " con tamaño máximo de espera en cola " + max);

            while (true) {
                Socket client = server.accept();
                System.out.println("Cliente conectado: " + client.getRemoteSocketAddress());
                client.close();
            }

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

Ver: MultiConstructorServer

Algunos otros métodos de interés son:

Método	Retorno	Descripción
accept()	Socket	Espera una conexión entrante y la acepta. Una vez que se ha establecido la conexión con el cliente, devuelve un objeto de tipo Socket, a través del cual se establecerá la comunicación con el cliente. Tras esto, el serverSocket sigue disponible para realizar nuevos accept(). Puede lanzar IOException.
bind(InetSocketAddress point)	void	Asocia el socket a una dirección y puerto.
bind(InetSocketAddress point, int max)	void	Asocia el socket a una dirección y puerto con cola de espera.
close()	void	Cierra el socket servidor.
isClosed()	boolean	Indica si el socket está cerrado.
isBound()	boolean	Indica si el socket está asociado a una dirección.
getInetAddress()	InetAddress	Devuelve objeto de direccionamiento asociado al socket.
getLocalPort()	int	Devuelve el puerto local al que está enlazado el ServerSocket.
getLocalSocketAddress()	SocketAddress	Devuelve la dirección local completa.
getRemoteSocketAddress()	SocketAddress	Devuelve la dirección remota completa.
setSoTimeout(int timeout)	void	Establece timeout para accept().
getSoTimeout()	int	Devuelve el timeout configurado.
toString()	String	Representación en texto del socket.

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



1. **Publicar puerto**. Se utiliza la función ServerSocket( ) indicando el puerto por donde se van a recibir las conexiones:

```
ServerSocket skServidor = new ServerSocket(puerto);
ServerSocket skServidor = new ServerSocket(puerto, max);
```

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

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

3. **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.

4. Una vez finalizada la comunicación se **cierra el socket del cliente y del Servidor**.

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

El siguiente ejemplo crea un socket de servidor y lo enlaza al puerto 6000, visualiza el puerto por el que se esperan las conexiones y espera a que se conecten dos clientes:

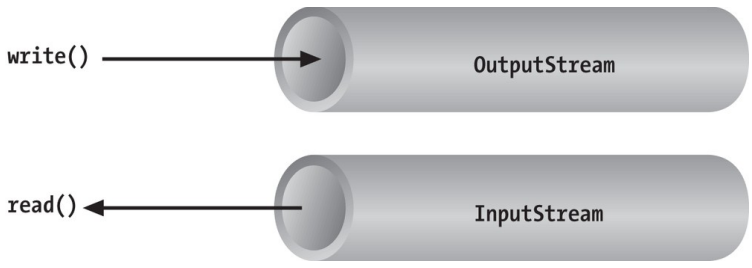
```
int puerto=6000;
ServerSocket servidor=new ServerSocket(puerto);
System.out.println("Escuchando en " + servidor.getLocalPort();

Socket cliente1=servidor.accept(); //esperando a un cliente
//realizar acciones con el cliente1

Socket cliente2=servidor.accept(); //esperando a un cliente
//realizar acciones con el cliente2

cliente1.close();
cliente2.close();
servidor.close(); //Se cierra el socket servidor
```

El siguiente ejemplo crea un socket cliente y lo conecta al host local al puerto 6000 (debe haber un serverSocket escuchando en ese puerto). Después se visualiza el puerto local al que está conectado el socket, el puerto, host y dirección IP de la máquina remota a la que se conecta).

```
String host="localhost";
int puerto=6000;

//Abrir socket
Socket cliente=new Socket(host, puerto);

InetAddress i=cliente.getInetAddress();
System.out.println("Puerto local: " + cliente.getLocalPort();
System.out.println("Puerto remoto: " + cliente.getPort();
System.out.println("Nombre host/ip: " + cliente.getInetAddress();
System.out.println("Host remoto: " + i.getHostName().toString();
System.out.println("Ip host remoto: " + i.getHostAddress().toString();

cliente.close(); //Cierra el socket
```

5.2.3. Streams de entrada y salida

Hay dos formas mayoritarias de enviar y recibir la información a través de los streams que proporciona un socket.

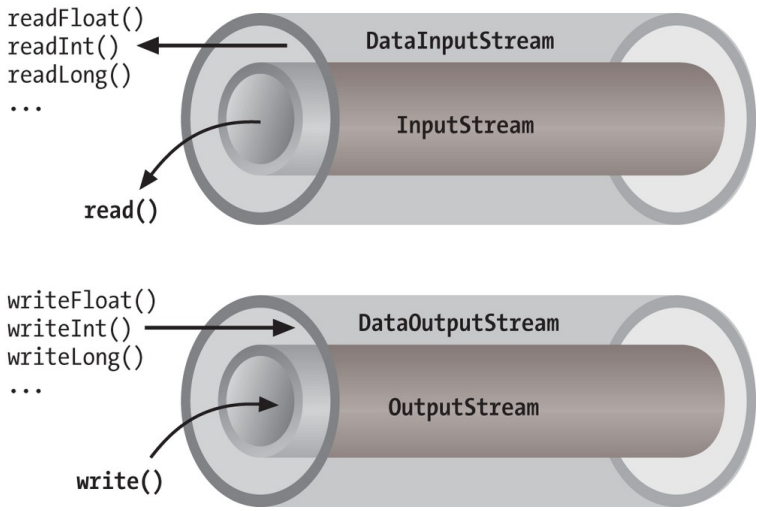


En cualquier caso, a través de los streams enviamos bytes, que es la forma más básica de generar información, bien sea a través de la red o entre procesos.

Como es complicado gestionar a nivel de bytes toda la información que queremos enviar o recibir, usamos Wrappers para enviar tipos de datos de un nivel de abstracción mayor.

En los temas anteriores, cuando hemos tenido que intercambiar información entre procesos, hemos estado usando BufferedReader y PrintWriter. Estas clases trabajan a nivel de Strings, y son muy útiles cuando lo que queremos intercambiar a través de los streams son cadenas de texto.

Sin embargo, puede haber ocasiones en las que nos interese trabajar con tipos de datos privmitivos. En ese caso, DataInputStream y DataOutputStream proporcionan métodos para leer y escribir Strings y todos los tipos de datos primitivos de Java, incluyendo números y valores booleanos.



Así, podemos trabajar con DataInputStream y DataOutputStream a partir de los streams que nos proporcionan los sockets.

En el programa servidor se puede usar DataInputStream para recuperar los mensajes que el cliente escriba en el socket, previamente hay que usar el método getInputStream() para obtener el flujo de entrada del socket del cliente:

```
InputStream entrada=null;
try
{
	entrada=sCliente.getInputStream();
}
catch (IOException e)
{
	System.out.println(e.getMessage());
}

DataInputStream dis=new DataInputStream(entrada);

dis.readDouble();
```

En el programa cliente se puede realizar la misma operación para recibir los mensajes procedentes del servidor.

Por otro lado, en el programa servidor se puede usar DataOutputStream para escribir los mensajes que queremos que el cliente recibe. Previamente hay que usar el método getOutputStream() para obtener el flujo de salida del socket del cliente:

```
OutputStream salida=null;
try{
	salida=sCliente.getOutputStream();
}
catch (IOException e)
{
	System.out.println(e.getMessage());
}

DataOutputStream dos=new DataOutputStream(salida);

dos.writeDouble();
```

En el programa cliente se puede realizar la misma operación para enviar mensajes al servidor.

5.2.4. Cierre de sockets

El orden de cierre de los sockets es relevante. Primero se han de cerrar los streams relacionados con un socket antes que el propio socket:

```
try
{
	entrada.close();
	dis.close();
	salida.close();
	dos.close();
	servidor.close();
}
catch(IOException e)
{
	System.out.println(e.getMessage());
}
```