Programación de Comunicación en Redes
La programación en redes es el acto de usar código informático para escribir programas o procesos que puedan comunicarse con otros programas o procesos a través de una red.

Existen muchos problemas que surgen al realizar programación en redes que no aparecen al desarrollar aplicaciones de un solo programa. Sin embargo, **JAVA** simplifica el desarrollo de aplicaciones de red gracias a sus bibliotecas fáciles de usar.

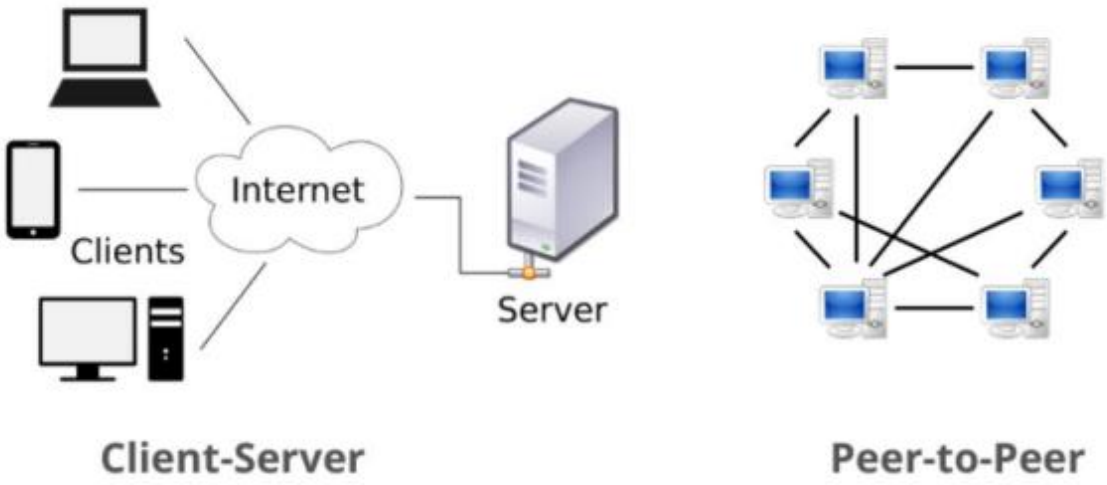
En general, las aplicaciones que tienen componentes ejecutándose en diferentes máquinas se conocen como **aplicaciones distribuidas** y, por lo general, consisten en relaciones de cliente/servidor.
Un **servidor** es una aplicación que proporciona un "servicio" a varios clientes que solicitan dicho servicio.

Existen muchos escenarios de cliente/servidor en la vida real:

- Los cajeros de banco (servidor) ofrecen un servicio a los titulares de cuentas (cliente).
- Los camareros (servidor) ofrecen un servicio a los clientes (cliente).
- Los agentes de viajes (servidor) ofrecen un servicio a las personas que desean ir de vacaciones (cliente).

En algunos casos, los servidores en sí mismos pueden convertirse en clientes en ciertos momentos. Por ejemplo, los agentes de viajes se convierten en clientes cuando llaman a una aerolínea para hacer una reserva o contactan con un hotel para reservar una habitación.

En el escenario general de redes, cualquier entidad puede ser cliente o servidor en cualquier momento. Esto se conoce como **computación entre pares (peer-to-peer)**. En términos de escritura de aplicaciones en Java, es similar a tener muchas aplicaciones comunicándose entre sí.



Existen muchas estrategias para permitir la comunicación entre aplicaciones.
Aquí veremos la estrategia más simple para conectar aplicaciones utilizando **sockets**.

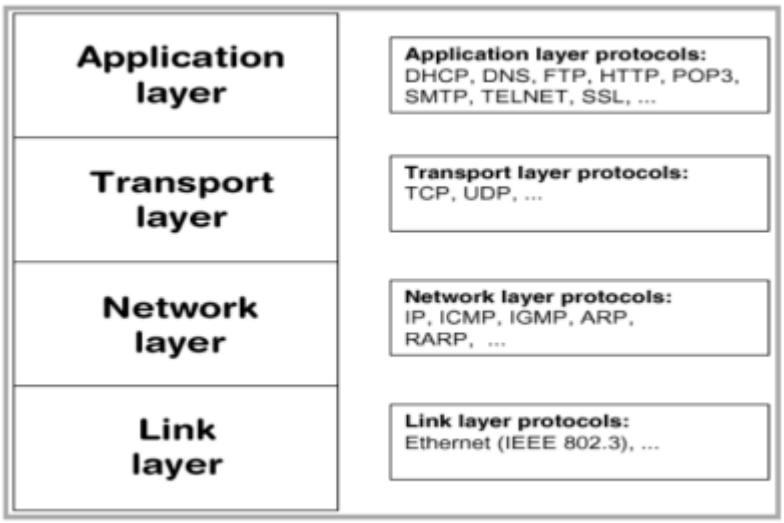
Un **protocolo** es un patrón estándar para intercambiar información, como un conjunto de reglas o pasos para la comunicación.

La **Suite de Protocolos de Internet** es una colección de protocolos —de ahí la palabra *suite*— que determina cómo debe funcionar Internet. El alias **TCP/IP** proviene de dos de los protocolos más importantes que contiene esta suite: el **Protocolo de Control de Transmisión (TCP)** y el **Protocolo de Internet (IP)**. Por simplicidad, a menudo se le denomina **pila de protocolos TCP/IP**.

Los cables y las computadoras que componen la infraestructura de Internet entienden un lenguaje binario muy simple, formado por ceros y unos. Sin embargo, queremos ser capaces de mover datos ricos, como páginas web, correos electrónicos, películas y videollamadas, de manera confiable, sin errores y fácil de establecer. Este es un problema complejo que debe descomponerse en piezas más pequeñas para ser resuelto eficientemente. Por esta razón, la pila de protocolos TCP/IP se organiza en **cuatro capas**.

Cada capa contiene protocolos que describen cómo enrutar, transmitir o recibir datos según un nivel diferente de abstracción. Cuanto más baja sea la capa, más cerca está del hardware y más detalladas son las instrucciones. Cuanto más alta sea la capa, más cerca está del usuario y más abstracta se vuelve la comunicación. Veamos un enfoque de abajo hacia arriba:

- **Capa de enlace.** También conocida como la capa física, contiene protocolos que operan muy cerca del hardware. Los protocolos en esta capa ven la red como un conjunto de máquinas físicamente conectadas que intercambian bits de datos.
- **Capa de red.** También conocida como la capa de Internet, aquí la comunicación comienza a volverse más compleja. Los protocolos en esta capa trabajan en términos de redes de origen y destino y cómo identificarlas.
- **Capa de transporte.** En esta capa, la comunicación se vuelve aún más abstracta. Los protocolos trabajan en términos de procesos que se comunican entre sí a través de canales específicos.
- **Capa de aplicación.** La capa más abstracta, donde los protocolos piensan en términos de servicios para el usuario que intercambian datos de aplicaciones a través de la red.



Inicialmente desarrollado por el Departamento de Defensa de los Estados Unidos y ahora mantenido por el **Internet Engineering Task Force (IETF)**, la **pila de protocolos TCP/IP** define cómo deben ser gestionados, transmitidos, enrutados y recibidos los datos a través de Internet. Cualquier cosa que esté conectada a Internet o que opere con él debe cumplir con las reglas definidas en la pila de protocolos TCP/IP. Dos máquinas que deseen comunicarse a través de Internet deben implementar ambas la pila de protocolos TCP/IP para poder comunicarse correctamente.

Por ejemplo, el **navegador web** implementa el **Protocolo de Transferencia de Hipertexto (HTTP)**, uno de los muchos protocolos de la pila TCP/IP y la base de la **World Wide Web (WWW)**. El protocolo HTTP determina cómo se debe enviar el texto de una página web desde el servidor web —la computadora remota que almacena la información— hasta un navegador web a través de Internet. El protocolo también describe cómo debe comunicarse el navegador con el servidor web para iniciar el intercambio de datos.

Muchas otras partes del software deben ser compatibles con TCP/IP. Por ejemplo, el sistema operativo que corre en tu dispositivo debe implementar varios protocolos de la suite TCP/IP para proporcionar capacidades de Internet al sistema completo (incluido el navegador web).

En una red TCP/IP, cada dispositivo debe tener una dirección IP.
La **dirección IP** identifica el dispositivo, por ejemplo, una computadora.
IPv4 es la primera versión del protocolo de Internet que se utilizó ampliamente. Significa **Internet Protocol Version 4**. Fue lanzada por primera vez en 1983 y sigue en uso hoy en día.
Las direcciones IP que utilizan este protocolo son grupos de 4 números decimales, cada uno con un valor de 0 a 255, separados por puntos, en lo que se conoce como **notación decimal punteada**. Un ejemplo de una dirección IPv4 sería:
192.168.10.150.

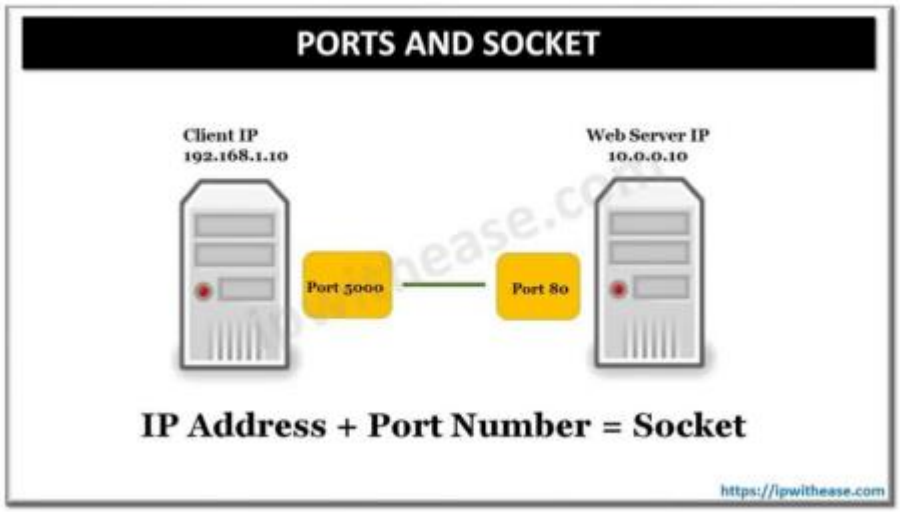
Cada número en una dirección IP almacena información que indica a los paquetes de datos hacia dónde deben ir. Cada grupo de números puede almacenar un byte de información, y hay cuatro grupos en cada dirección IPv4, lo que da un total de 32 bits de almacenamiento de información. Por esa razón, el sistema de direccionamiento IPv4 se conoce como un sistema de **32 bits**.
Este sistema permite hasta **4.3 mil millones** de direcciones únicas, lo cual parece mucho hasta que te das cuenta de cuántas personas y dispositivos están conectados a Internet hoy en día. Los dispositivos en el **Internet de las Cosas (IoT)** también necesitan direcciones IP. Esa explosión de uso, a medida que nuestras vidas se conectan cada vez más a Internet, significa que el sistema de direccionamiento IPv4 se está quedando sin espacio.

IPv6 es una versión más nueva del protocolo de Internet con direcciones más largas que contienen tanto números como letras. Aunque es más nuevo que la versión 4, no es tan nuevo: fue implementado por primera vez en 1999.
Las direcciones **IPv6** tienen **128 bits** de almacenamiento de información. Se escriben en **notación hexadecimal con dos puntos**, lo que significa que cada grupo de números y letras está separado del siguiente por un **dos puntos (:)** en lugar de un punto.
Un ejemplo de una dirección IPv6 sería:
3002:0bd6:0000:0000:0000:ee00:0033:6778
Este sistema de direccionamiento más largo admite **2128** direcciones únicas, o **1028 veces** el

número de direcciones de IPv4. **4.3 mil millones multiplicado por 1028** es suficiente para que no tengamos que preocuparnos por quedarnos sin direcciones en mucho tiempo.

Una dirección IP por sí sola no es suficiente para ejecutar aplicaciones de red, ya que una computadora puede ejecutar múltiples aplicaciones y/o servicios.
Así como la dirección IP identifica la computadora, el **puerto de red** identifica la aplicación o servicio que se está ejecutando en la computadora.
El uso de **puertos** permite que las computadoras/dispositivos ejecuten múltiples servicios/aplicaciones.

El diagrama a continuación muestra una conexión de computadora a computadora e identifica las direcciones IP y los puertos.



Un número de puerto usa 16 bits y, por lo tanto, puede tener un valor de 0 a 65535. Los números de puerto se dividen en los siguientes rangos:

- **Puertos 0-1023** – Puertos bien conocidos. Estos están asignados a servicios de servidor por la **Autoridad de Números Asignados de Internet (IANA)**. Por ejemplo, los servidores web normalmente usan el puerto 80 y los servidores SMTP usan el puerto 25.
- **Puertos 1024-49151** – Puertos registrados. Estos pueden ser registrados para servicios con la IANA y deben ser tratados como semi-reservados. Los programas escritos por los usuarios no deben usar estos puertos.
- **Puertos 49152-65535** – Estos son utilizados por programas cliente y puedes usarlos libremente en programas cliente. Cuando un navegador web se conecta a un servidor web, el navegador se asignará a un puerto en este rango. También se conocen como puertos efímeros.

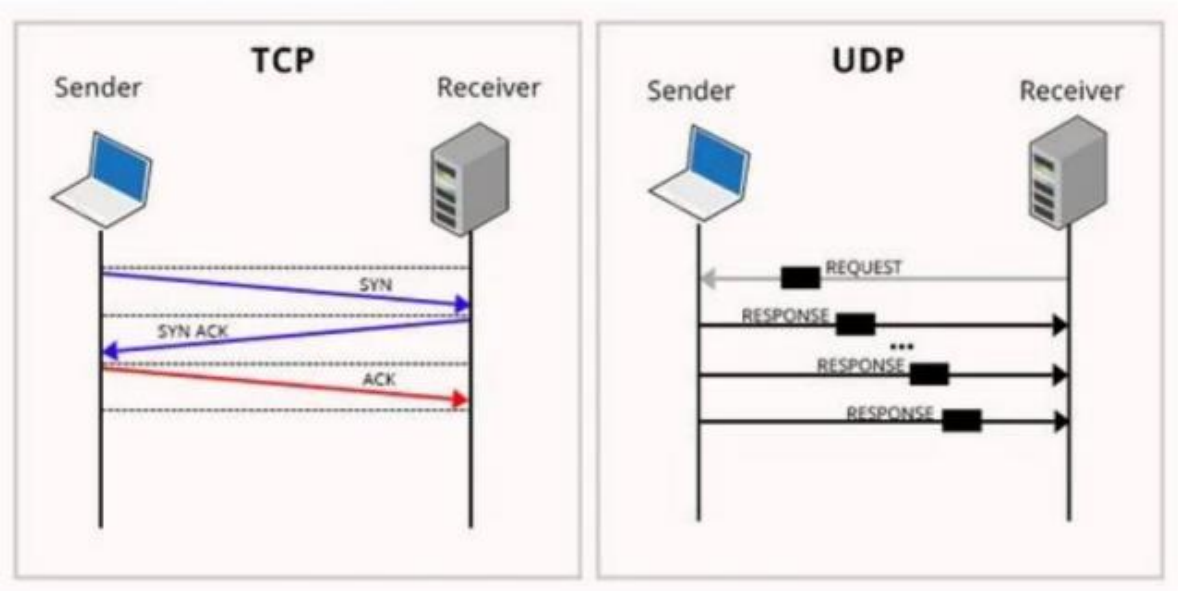
Cuando escribes aplicaciones en **Java** que se comunican a través de una red, estás programando en la **Capa de Aplicación**. **JAVA** permite dos tipos de comunicación a través de dos protocolos principales de la **Capa de Transporte**:

- **TCP**. Un protocolo basado en conexiones que proporciona un flujo de datos confiable entre dos computadoras. Garantiza que los datos enviados desde un extremo de la conexión lleguen realmente al otro extremo y en el mismo orden. TCP proporciona un

canal punto a punto para aplicaciones que requieren comunicaciones confiables. Tiene un tiempo de sobrecarga para establecer una conexión de extremo a extremo.

- **UDP**. Este es un protocolo que envía paquetes independientes de datos, llamados datagramas, de una computadora a otra. **UDP** no da garantías sobre la llegada de los datos. **UDP** no está basado en conexión como **TCP**. Proporciona comunicación que no está garantizada entre los dos extremos. Enviar paquetes es como enviar una carta a través del servicio postal. El orden de entrega no es importante ni está garantizado. Cada mensaje es independiente de cualquier otro. **UDP** es más rápido ya que no tiene sobrecarga por establecer una conexión de extremo a extremo. Muchos **firewalls** y **enrutadores** han sido configurados para no permitir paquetes UDP.

¿Por qué alguien querría usar el protocolo UDP si la información puede perderse? Bueno, ¿por qué usamos el correo electrónico o la oficina de correos? Nunca se nos garantiza que nuestro correo llegará a la persona a la que lo enviamos, sin embargo, seguimos confiando en esos servicios de entrega. A veces, puede ser más rápido que intentar contactar a una persona por teléfono para transmitir los datos (es decir, como un protocolo TCP).



Clase java.net.NetworkInterface en Java

Esta clase representa la interfaz de red, tanto de software como de hardware, su nombre, la lista de direcciones IP asignadas a ella y toda la información relacionada. Se puede usar en casos en los que queramos usar una interfaz específica para transmitir nuestro paquete en un sistema con múltiples NICs.

¿Qué es una interfaz de red?

Se puede pensar en una interfaz de red como un punto en el que tu computadora se conecta a la red. No es necesariamente un componente de hardware, sino que también puede ser implementado en software. Por ejemplo, una interfaz de loopback, que se usa para fines de prueba.

Método	Descripción
`public String getName()`	Devuelve el nombre de esta interfaz de red.
`public Enumeration getInetAddresses()`	Devuelve una enumeración de todas las direcciones Inet asignadas a esta interfaz de red, si el administrador de seguridad lo permite.
`public List getInterfaceAddresses()`	Devuelve una lista de todas las direcciones de interfaz en esta interfaz.
`public Enumeration getSubInterfaces()`	Devuelve una enumeración de todas las subinterfaces o interfaces virtuales de esta interfaz de red. Por ejemplo, eth0:2 es una subinterfaz de eth0.
`public NetworkInterface getParent()`	En el caso de una subinterfaz, este método devuelve la interfaz principal. Si no es una subinterfaz, este método devolverá `null`.
`public int getIndex()`	Devuelve el índice asignado a esta interfaz de red por el sistema. Los índices se pueden usar en lugar de nombres largos para hacer referencia a cualquier interfaz del dispositivo.
`public String getDisplayName()`	Este método devuelve el nombre de la interfaz de red en un formato de cadena legible.
`public static NetworkInterface getByName(String name) throws SocketException`	Encuentra y devuelve la interfaz de red con el nombre especificado, o `null` si no existe.
`public static NetworkInterface getByIndex(int index) throws SocketException`	Realiza una función similar a la anterior, pero utilizando el índice como parámetro de búsqueda en lugar del nombre.
`public static NetworkInterface getByInetAddress(InetAddress addr) throws SocketException`	Este método devuelve la interfaz de red a la que está vinculada la dirección Inet especificada. Si una dirección Inet está vinculada a varias interfaces, cualquiera de ellas puede ser devuelta.
`public static Enumeration getNetworkInterfaces() throws SocketException`	Devuelve todas las interfaces de red en el sistema.
`public boolean isUp()`	Devuelve un valor booleano que indica si esta interfaz de red está activa y en funcionamiento.
`public boolean isLoopback()`	Devuelve un valor booleano que indica si esta interfaz es una interfaz de loopback.
`public boolean isPointToPoint()`	Devuelve un valor booleano que indica si esta interfaz es una interfaz punto a punto.
`public boolean supportsMulticast()`	Devuelve un valor booleano que indica si esta interfaz soporta multicast.

Método	Descripción
`public byte[] getHardwareAddress()`	Devuelve un array de bytes que contiene la dirección de hardware (MAC) de esta interfaz. El llamador debe tener permisos adecuados antes de invocar este método.
`public int getMTU()`	Devuelve la unidad máxima de transmisión (MTU) de esta interfaz.
`public boolean isVirtual()`	Devuelve un valor booleano que indica si esta interfaz es virtual.
`public boolean equals(Object obj)`	Este método se usa para comparar dos interfaces de red en cuanto a su igualdad. Dos interfaces de red son iguales si tienen el mismo nombre y las direcciones vinculadas a ellas.
`public int hashCode()`	Devuelve el valor hash para este objeto.
`public String toString()`	Devuelve una descripción textual de este objeto.

Clase java.net.InterfaceAddress en Java

Esta clase representa una dirección de interfaz de red. Cada dispositivo que tiene una dirección IP tiene una dirección IP en la interfaz de red. De hecho, el comando ping no hace ping a un dispositivo, sino a la dirección de interfaz del dispositivo. Java proporciona ciertos métodos para tratar con direcciones de interfaz, que se pueden usar en lugares donde es necesario conocer la topología de la red, para la detección de fallos, etc.

Esta clase representa una dirección IP, una máscara de red y una dirección de difusión (cuando la dirección es IPv4). Solo representa una dirección IP y una longitud de prefijo de red en el caso de direcciones IPv6.

Método	Descripción
`public InetAddress getAddress()`	Devuelve una `InetAddress` para esta dirección.
`public InetAddress getBroadcast()`	Devuelve la `InetAddress` para la dirección de difusión de esta dirección de interfaz. Dado que solo las direcciones IPv4 tienen direcciones de difusión, se devolverá `null` si se usa una dirección IPv6.
`public short getNetworkPrefixLength()`	Devuelve la longitud del prefijo para esta dirección de interfaz, es decir, la máscara de subred para esta dirección.
`public boolean equals(Object obj)`	Se usa para comparar el objeto especificado con esta dirección de interfaz. Devuelve `true` solo si el objeto dado no es `null` y representa la misma dirección de interfaz que este objeto.
`public int hashCode()`	Devuelve el valor hash para esta dirección de interfaz.
`public String toString()`	Devuelve una representación en cadena de esta dirección

Método	Descripción
	de interfaz. La cadena tiene el formato: Dirección de interfaz/prefijo de longitud.

Clase java.net.InetAddress en Java

La clase `java.net.InetAddress` proporciona métodos para obtener la dirección IP de cualquier nombre de host. Una dirección IP está representada por un número sin signo de 32 bits o 128 bits. InetAddress puede manejar direcciones IPv4 e IPv6.

Existen 2 tipos de direcciones:

- **Unicast**: Un identificador para una sola interfaz.
- **Multicast**: Un identificador para un conjunto de interfaces.

La clase InetAddress se usa para encapsular tanto la dirección IP numérica como el nombre de dominio de esa dirección. La clase InetAddress no tiene constructores visibles. La clase InetAddress no tiene la capacidad de crear objetos directamente, por lo que se utilizan métodos de fábrica para este propósito. Los métodos de fábrica son métodos estáticos en una clase que devuelven un objeto de esa clase.

Método	Descripción
`public static InetAddress getLocalHost() throws UnknownHostException`	Este método devuelve la instancia de `InetAddress` que contiene el nombre de host y la dirección local.
`public static InetAddress getByName(String host) throws UnknownHostException`	Este método devuelve la instancia de `InetAddress` que contiene la IP y el nombre de host del host representado por el argumento `host`.
`public static InetAddress[] getAllByName(String hostName) throws UnknownHostException`	Este método devuelve el array de instancias de la clase `InetAddress` que contiene las direcciones IP.
`public static InetAddress getByAddress(byte IPAddress[]) throws UnknownHostException`	Este método devuelve un objeto `InetAddress` creado a partir de la dirección IP cruda.
`public static InetAddress getByAddress(String hostName, byte IPAddress[]) throws UnknownHostException`	Este método crea y devuelve un `InetAddress` basado en el nombre de host proporcionado y la dirección IP.

Ejemplo:

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

public class EjemploInetAddress {
    public static void main(String[] args) throws UnknownHostException {
        // Obtener e imprimir la dirección InetAddress del Host Local
```

```
        InetAddress direccion1 = InetAddress.getLocalHost();
        System.out.println("Dirección InetAddress del Host Local: " + direccion1);

        // Obtener e imprimir la dirección InetAddress del Host Nombrado
        InetAddress direccion2 = InetAddress.getByName("www.iessanclemente.net");
        System.out.println("Dirección InetAddress del Host Nombrado: " + direccion2);

        // Obtener e imprimir TODAS las direcciones InetAddress del Host Nombrado
        InetAddress[] direccion3 = InetAddress.getAllByName("www.google.com");
        for (int i = 0; i < direccion3.length; i++) {
            System.out.println("TODAS las direcciones InetAddress del Host Nombrado: " +
direccion3[i]);
        }

        // Obtener e imprimir la dirección InetAddress del Host con una dirección IP específica
        byte[] direccionIP = { 1, 24, 0, 1 };
        InetAddress direccion4 = InetAddress.getByAddress(direccionIP);
        System.out.println("Dirección InetAddress del Host con una dirección IP específica: " +
direccion4);

        // Obtener e imprimir la dirección InetAddress del Host con una dirección IP específica
y un nombre de host
        byte[] direccionIP2 = { (byte)193, (byte)144, 43, (byte)236 };
        InetAddress direccion5 = InetAddress.getByAddress("mestre.iessanclemente.net",
direccionIP2);
        System.out.println("Dirección InetAddress del Host con una dirección IP específica y un
nombre de host: " + direccion5);
    }
}
```

Métodos de la clase `InetAddress`:

La clase `InetAddress` tiene muchos métodos de instancia que pueden ser llamados utilizando el objeto. Aquí tienes una lista con la descripción de algunos métodos:

Método	Descripción
`equals(Object obj)`	Compara este objeto con el objeto especificado.
`getAddress()`	Devuelve la dirección IP en crudo de este objeto `InetAddress`, en bytes.
`getCanonicalHostName()`	Devuelve el nombre de dominio completamente calificado para esta dirección IP.
`getHostAddress()`	Obtiene la dirección IP en formato de cadena.
`getHostName()`	Devuelve el nombre del host para esta dirección IP.
`hashCode()`	Obtiene un código hash para esta dirección

Método	Descripción
	IP.
`isAnyLocalAddress()`	Verifica si la dirección `InetAddress` es una dirección no predecible.
`isLinkLocalAddress()`	Verifica si la dirección no está vinculada a una dirección local unicast.
`isLoopbackAddress()`	Verifica si la dirección `InetAddress` representa una dirección de loopback.
`isMCGlobal()`	Verifica si esta dirección tiene un alcance multicast global.
`isMCLinkLocal()`	Verifica si esta dirección tiene un alcance multicast local de enlace.
`isMCNodeLocal()`	Verifica si la dirección multicast tiene un alcance de nodo.
`isMCOrgLocal()`	Verifica si la dirección multicast tiene un alcance de organización.
`isMCSiteLocal()`	Verifica si la dirección multicast tiene un alcance de sitio.
`isMulticastAddress()`	Verifica si la dirección es multicast.
`isReachable(int timeout)`	Verifica si esa dirección es alcanzable.
`isReachable(NetworkInterface netif, int ttl, int timeout)`	Verifica si esa dirección es alcanzable, considerando la interfaz de red, el tiempo de vida y el tiempo de espera.
`isSiteLocalAddress()`	Verifica si la dirección `InetAddress` es una dirección local de sitio.
`toString()`	Convierte y devuelve la dirección IP en formato de cadena.

Clase `java.net.URL` en Java

La clase `java.net.URL` proporciona una representación del Localizador Uniforme de Recursos (URL) de Internet. Un recurso puede ser cualquier cosa, desde un archivo de texto simple hasta otros elementos como imágenes, archivos de directorio, etc.

Una URL tiene las siguientes partes:

- **Protocolo**: El protocolo puede ser HTTP o HTTPS.
- **Nombre del host o IP**: El nombre del host representa la dirección de la máquina donde se encuentra el recurso.
- **Número de puerto**: Este es un atributo opcional. Si no se especifica, se devuelve -1. Si no se especifica, se utiliza el puerto predeterminado por el protocolo indicado en el primer campo.
- **Nombre del recurso**: Es el nombre del recurso ubicado en el servidor dado. Dependiendo de la configuración del servidor, el nombre del archivo puede tener un valor predeterminado.
- **Ejemplo de URL: https://www.iessanclemente.net/o-centro/historia/**
- Los constructores de la clase `URL` fueron:

Método	Descripción
`URL(String address) throws MalformedURLException`	Crea un objeto URL a partir de la cadena especificada.
`URL(String protocol, String host, String file)`	Crea un objeto URL a partir del protocolo, host y nombre de archivo especificados.
`URL(String protocol, String host, int port, String file)`	Crea un objeto URL a partir del protocolo, host, puerto y nombre de archivo.
`URL(URL context, String spec)`	Crea un objeto URL analizando la especificación dada en el contexto dado.
`URL(String protocol, String host, int port, String file, URLStreamHandler handler)`	Crea un objeto URL a partir del protocolo, host, número de puerto, archivo y manejador especificados.
`URL(URL context, String spec, URLStreamHandler handler)`	Crea una URL analizando la especificación dada con el manejador especificado dentro de un contexto dado.

Los constructores de `java.net.URL` están obsoletos. Se recomienda a los desarrolladores que utilicen `java.net.URI` para analizar o construir una URL.

Métodos importantes utilizados en la clase `URL`:

Método	Descripción
`getAuthority()`	Devuelve la parte de autoridad de la URL o `null` si está vacía.
`getDefaultPort()`	Devuelve el puerto predeterminado utilizado.
`getFile()`	Devuelve el nombre del archivo.
`getHost()`	Devuelve el nombre del host de la URL en formato IPv6.
`getPath()`	Devuelve la ruta de la URL, o `null` si está vacía.
`getPort()`	Devuelve el puerto asociado con el protocolo especificado por la URL.
`getProtocol()`	Devuelve el protocolo utilizado por la URL.
`getQuery()`	Devuelve la parte de consulta de la URL. Una consulta es la parte después del `?` en la URL. Siempre que se use lógica para mostrar el resultado, habrá un campo de consulta en la URL. Es similar a hacer una consulta en una base de datos.
`getRef()`	Devuelve la referencia del objeto URL. Normalmente, la referencia es la parte marcada por un `#` en la URL.
`toString()`	Como en cualquier clase, `toString()` devuelve la representación en cadena del objeto URL dado.

Veamos un ejemplo de código Java utilizando la clase `URL`:

```
import java.net.MalformedURLException;
```

```
import java.net.URL;
import java.net.URI;
import java.net.URISyntaxException;
public class URLDemo {
 public static void main(String[] args) throws URISyntaxException,
MalformedURLException {
 URL url=new
URI("https://en.wikipedia.org/wiki/Internet#Terminology").toURL();
 System.out.println("Protocol: "+url.getProtocol());
 System.out.println("Host Name: "+url.getHost());
 System.out.println("Port Number: "+url.getPort());
 System.out.println("Default Port Number: "+url.getDefaultPort());
 System.out.println("Query String: "+url.getQuery());
 System.out.println("Path: "+url.getPath());
 System.out.println("File: "+url.getFile());
 }
}
```

Clase `java.net.URLConnection` en Java

`URLConnection` es una clase abstracta cuyas subclases forman el enlace entre la aplicación del usuario y cualquier recurso en la web. Podemos usarla para leer/escribir desde/a cualquier recurso referenciado por un objeto `URL`. Principalmente, existen dos subclases que extienden la clase `URLConnection`:

- **HttpURLConnection**: Si nos estamos conectando a una URL que usa "http" como su protocolo, se utiliza la clase `HttpURLConnection`.
- **JarURLConnection**: Si, en cambio, estamos intentando establecer una conexión con un archivo jar en la web, entonces se usa `JarURLConnection`.

Una vez establecida la conexión y teniendo un objeto `URLConnection`, podemos usarlo para leer o escribir o para obtener más información sobre cuándo fue la última vez que la página/archivo fue modificado, la longitud del contenido, etc. Sin embargo, obtener solo la información del estado no es el verdadero motivo de una aplicación del mundo real. Lo que buscamos es recuperar la información, procesarla y enviar los resultados de vuelta al servidor, o simplemente mostrar la información requerida recuperada del servidor.

Algunos de los métodos de la clase `URLConnection` son los siguientes:

Método	Descripción
`getContent()`	Recupera el contenido de la `URLConnection`.
`getContentEncoding()`	Devuelve el valor del campo de encabezado `content-encoding`.

Método	Descripción
`getContentLength()`	Devuelve la longitud del campo de encabezado `content-length`.
`getDate()`	Devuelve el valor de la fecha en el campo de encabezado.
`getHeaderFields()`	Devuelve el mapa que contiene los valores de los diversos campos de encabezado en el encabezado HTTP.
`getHeaderField(int i)`	Devuelve el valor del índice i del encabezado.
`getHeaderField(String field)`	Devuelve el valor del campo llamado "field" en el encabezado.
`getInputStream()`	Devuelve el flujo de entrada para esta conexión abierta.
`getOutputStream()`	Devuelve el flujo de salida para esta conexión del tipo `OutputStream`.
`openConnection()`	Abre la conexión a la URL especificada.
`setAllowUserInteraction()`	Configurar esto como `true` significa que un usuario puede interactuar con la página. El valor predeterminado es `true`.
`setDefaultUseCaches()`	Establece el valor predeterminado del campo `useCache` como el valor dado.
`setDoInput()`	Establece si al usuario se le permite tomar entradas o no.
`setDoOutput()`	Establece si al usuario se le permite escribir en la página. El valor predeterminado es `false`, ya que la mayoría de las URLs no permiten escribir.

Típicamente, un programa cliente que se comunica con un servidor a través de una URL sigue esta secuencia de pasos:

1. Crear un objeto `URL`.
2. Obtener un objeto `URLConnection` de la URL.
3. Configurar la `URLConnection`.
4. Leer los campos de encabezado.
5. Obtener un flujo de entrada y leer datos.
6. Obtener un flujo de salida y escribir datos.
7. Cerrar la conexión.

Los pasos 3 a 6 son opcionales, y los pasos 5 y 6 son intercambiables.

El siguiente ejemplo usa la clase `URLConnection`:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.URI;
import java.net.URISyntaxException;
import java.net.URL;
import java.net.URLConnection;

public class EjemploURL {
    public static void main(String[] args) throws IOException,
URISyntaxException {
        // Creando un objeto de la clase URL
        // Se pasa la URL personalizada como argumento
        URL u = new URI("https://www.simplilearn.com/java-
networking-article").toURL();

        // Creando un objeto de la clase URLConnection para
        // comunicar la aplicación con la URL
        URLConnection urlconnect = u.openConnection();

        // Creando un objeto de la clase InputStream
        // para leer los flujos de la aplicación
        InputStream stream = urlconnect.getInputStream();
        BufferedReader in = new BufferedReader(new
InputStreamReader(stream));

        // Mientras se lee la URL
        String line;
        while ((line = in.readLine()) != null) {
            // Continuar imprimiendo el flujo
            System.out.println(line);
        }
    }
}
```

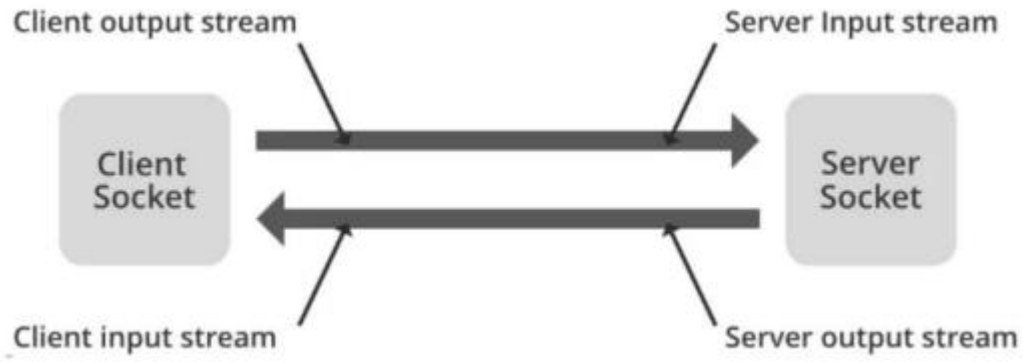
Comunicación Cliente/Servidor mediante Sockets

Muchas aplicaciones de red están basadas en el modelo cliente/servidor. Según este modelo, una tarea se ve como un servicio que puede ser solicitado por los clientes y gestionado por los servidores.

Un **socket** es un canal de comunicación simple a través del cual dos programas se comunican a través de una red. Un socket admite comunicación bidireccional entre un cliente y un servidor, utilizando un protocolo bien establecido. El protocolo simplemente prescribe reglas y comportamientos que tanto el servidor como el cliente deben seguir para establecer una comunicación bidireccional.

De acuerdo con este protocolo, un programa servidor crea un socket en un puerto determinado y espera hasta que un cliente solicite una conexión. Un **puerto** es una dirección particular o punto de entrada en el equipo host, que generalmente tiene cientos de puertos potenciales. Por lo general, se representa como un valor entero simple. Por ejemplo, el puerto estándar para un servidor HTTP (Web) es el 80. Una vez que se establece la conexión, el servidor crea flujos de entrada y salida hacia el socket y comienza a enviar y recibir mensajes del cliente. Tanto el cliente como el servidor pueden cerrar la conexión, pero generalmente lo hace el cliente.

Desde el lado del cliente, el protocolo sigue el siguiente proceso: El cliente crea un socket e intenta hacer una conexión con el servidor. El cliente debe conocer la URL del servidor y el puerto en el que existe el servicio. Una vez que se establece la conexión, el cliente crea flujos de entrada y salida hacia el socket y comienza a intercambiar mensajes con el servidor. El cliente puede cerrar la conexión cuando el servicio se haya completado.



Ahora veamos cómo se codificaría una aplicación cliente/servidor en Java. El primer paso que toma el servidor es crear un **ServerSocket**. El primer argumento del método **ServerSocket()** es el puerto en el que residirá el servicio. El segundo argumento especifica la cantidad de clientes que pueden estar en espera (en cola) en el servidor antes de que un cliente sea rechazado. Si más de un cliente solicita el servicio al mismo tiempo, Java establecerá y gestionará una lista de espera, rechazando clientes cuando la lista esté llena.

El siguiente paso es esperar una solicitud del cliente. El método **accept()** bloqueará la ejecución hasta que se establezca una conexión. El sistema Java es responsable de despertar al servidor cuando se recibe una solicitud del cliente.

Una vez que se establece la conexión, el servidor puede comenzar a comunicarse con el cliente. Tanto el cliente como el servidor pueden "hablar" entre sí. La conversación bidireccional se gestiona conectando tanto un flujo de entrada como uno de salida al socket. Una vez que se ha finalizado la conversación entre el cliente y el servidor (una vez que el servidor ha entregado el servicio solicitado), el servidor puede cerrar la conexión llamando al método **close()**. Así, los cuatro pasos involucrados en el lado del servidor son:

- Crear un **ServerSocket** y establecer un número de puerto.
- Escuchar y aceptar una conexión de un cliente.
- Conversar con el cliente.
- Cerrar el socket.

Lo que distingue al servidor del cliente es que el servidor establece el puerto y acepta la conexión.

```
java
CopiarEditar
Socket socket; // Referencia al socket
ServerSocket puerto; // El puerto donde el servidor escuchará

try {
    puerto = new ServerSocket(10001, 5); // Crear un puerto
    socket = puerto.accept(); // Esperar que el cliente se
conecte
    // Comunicar con el cliente
    socket.close();
} catch (IOException e) {
    e.printStackTrace();
}
```

El protocolo del cliente es igual de fácil de implementar. De hecho, en el lado del cliente, solo hay tres pasos involucrados. El primer paso es solicitar una conexión con el servidor. Esto se hace en el constructor **Socket()** proporcionando la URL y el número de puerto del servidor. Una vez que se establece la conexión, el cliente puede llevar a cabo la comunicación bidireccional con el servidor. Finalmente, cuando el cliente termina, puede simplemente cerrar() la conexión. Así que, desde el lado del cliente, el protocolo involucra solo tres pasos:

- Abrir una conexión de socket con el servidor, dado su dirección.
- Conversar con el servidor.
- Cerrar la conexión.

Lo que distingue al cliente del servidor es que el cliente inicia la conexión bidireccional solicitando el servicio.

```
java
CopiarEditar
Socket conexion; // Referencia al socket

try { // Solicitar una conexión
    conexion = new Socket("java.cs.trincoll.edu", 10001);
    // Mantener una comunicación bidireccional
    conexion.close(); // Cerrar el socket
} catch (IOException e) {
    e.printStackTrace();
}
```

Ahora veamos la verdadera comunicación bidireccional que ocurre. Dado que esta parte del proceso será exactamente la misma tanto para el cliente como para el servidor, podemos desarrollar un único conjunto de métodos, **writeToSocket()** y **readFromSocket()**, que pueden ser llamados por cualquiera de los dos.

El método **writeToSocket()** toma dos parámetros: el **Socket** y un **String**, que será enviado al proceso en el otro extremo del socket:

```
java
CopiarEditar
public void writeToSocket(Socket sock, String str) throws
IOException {
    oStream = sock.getOutputStream();
    for (int k = 0; k < str.length(); k++) {
        oStream.write(str.charAt(k));
    }
} // writeToSocket()
```

Si **writeToSocket()** es llamado por el servidor, entonces el string será enviado al cliente. Si es llamado por el cliente, el string será enviado al servidor.

Dada la referencia al flujo de salida del socket, simplemente escribimos cada carácter del string usando el método **OutputStream.write()**. Este método escribe un solo byte. Por lo tanto, el flujo de entrada en el otro lado del socket debe leer bytes y convertirlos de nuevo a caracteres.

El método **readFromSocket()** toma un parámetro **Socket** y devuelve un **String**:

```
java
CopiarEditar
protected String readFromSocket(Socket sock) throws
IOException {
    iStream = sock.getInputStream();
    String str = "";
    char c;
    while ((c = (char) iStream.read()) != '\n') {
        str = str + c + "";
    }
    return str;
}
```

Utiliza el método **Socket.getInputStream()** para obtener una referencia al flujo de entrada del socket, que ya ha sido creado. Así que aquí también, es importante que no cierres el flujo en este método. Los flujos de entrada y salida de un socket se cerrarán automáticamente cuando se cierre la conexión del socket.

El método **InputStream.read()** lee un solo byte a la vez del flujo de entrada hasta que se recibe un carácter de fin de línea. Observa el uso del operador de conversión (**char**) en la instrucción **read()**. Debido a que se están leyendo bytes, deben convertirse a **char** antes de que puedan ser comparados con el carácter de fin de línea o concatenados al **String**. Cuando el bucle de lectura encuentra un carácter de fin de línea, termina y devuelve el **String** que se ingresó.

Servidor Secuencial Trivial

Este es probablemente el servidor más simple posible. Escucha en el puerto 57777. Cuando un cliente se conecta, el servidor envía la fecha y hora actual al cliente. El socket de conexión se crea en un bloque **try-with-resources**, por lo que se cierra automáticamente al final del bloque. Solo después de servir la fecha y hora y cerrar la conexión, el servidor volverá a esperar al siguiente cliente.

```
import java.io.IOException;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.util.Date;

public class DateServer {
    public static void main(String[] args) throws IOException
{
        try (var listener = new ServerSocket(57777)) {
            System.out.println("El servidor de fecha está
funcionando...");
            while (true) {
                try (var socket = listener.accept()) {
                    var out = new
PrintWriter(socket.getOutputStream(), true);
                    out.println(new Date().toString());
                }
            }
        }
    }
}
```

Este código no maneja bien múltiples clientes; cada cliente debe esperar hasta que el cliente anterior haya sido completamente atendido antes de ser aceptado. La llamada a **ServerSocket.accept()** es una **LLAMADA BLOQUEANTE**.

La comunicación de los sockets siempre es con bytes; por lo tanto, los sockets vienen con flujos de entrada y salida. Pero al envolver el flujo de salida del socket con un **PrintWriter**, podemos especificar cadenas de texto para escribir, que Java convertirá (decodificará) automáticamente a bytes.

La comunicación a través de sockets siempre está "bufferizada". Esto significa que no se enviará ni recibirá nada hasta que los buffers se llenen, o hasta que explícitamente vacíes el buffer. El segundo argumento del **PrintWriter**, en este caso **true**, le indica a Java que vacíe el buffer automáticamente después de cada **println**.

Definimos todos los sockets dentro de un bloque **try-with-resources** para que se cierren automáticamente al final de su bloque. No se requiere una llamada explícita a **close()**.

Después de enviar la fecha y hora al cliente, el bloque **try** termina y el socket de comunicación se cierra, por lo que en este caso, el servidor inicia el cierre de la conexión.

Ahora veamos cómo escribir nuestro propio cliente en Java:

```
import java.io.IOException;
import java.net.Socket;
import java.util.Scanner;

public class DateClient {
    public static void main(String[] args) throws IOException
{
        if (args.length != 1) {
            System.err.println("Pase la IP del servidor como
único argumento de línea de comandos");
            return;
        }

        var socket = new Socket(args[0], 57777);
        var in = new Scanner(socket.getInputStream());
        System.out.println("Respuesta del servidor: " +
in.nextLine());
    }
}
```

En el lado del cliente, el constructor **Socket** toma la dirección IP y el puerto del servidor. Si la solicitud de conexión es aceptada, obtenemos un objeto **Socket** para comunicar.

Nuestra aplicación es tan simple que el cliente nunca escribe al servidor, solo lee. Como estamos comunicándonos con texto, lo más sencillo es envolver el flujo de entrada del socket en un **Scanner**. Estos son poderosos y convenientes. En nuestro caso, leemos una línea de texto desde el servidor con **Scanner.nextLine()**.

Un servidor simple con hilos

Este siguiente servidor recibe líneas de texto de un cliente y devuelve las líneas en mayúsculas. Maneja eficientemente múltiples clientes a la vez. Cuando un cliente se conecta, el servidor genera un hilo dedicado solo a ese cliente para leer, convertir a mayúsculas y responder. El servidor puede escuchar y servir a otros clientes al mismo tiempo, por lo que tenemos concurrencia real.

```
import java.io.IOException;
import java.net.ServerSocket;

public class CapitalizeMultiServer {
    private ServerSocket serverSocket;

    public static void main(String[] args) {
        CapitalizeMultiServer server = new
CapitalizeMultiServer();
        server.begin(55555);
```

```
    }

    public void begin(int port) {
        try {
            serverSocket = new ServerSocket(port);
            System.out.println("Servidor escuchando");
            while (true) {
                new Thread(new
CapitalizeClientHandler(serverSocket.accept())).start();
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            stop();
        }
    }

    public void stop() {
        try {
            serverSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Nota que llamamos a **accept** dentro de un bucle **while**. Cada vez que el bucle se ejecuta, bloquea la llamada a **accept** hasta que un nuevo cliente se conecta. Luego, se crea el hilo **CapitalizeClientHandler** para este cliente.

Lo que sucede dentro del hilo es lo mismo que en el **DateServer**, donde solo manejábamos un cliente. El **CapitalizeMultiServer** delega este trabajo a **CapitalizeClientHandler** para poder seguir escuchando más clientes en el bucle **while**.

En este caso, podemos crear múltiples clientes, cada uno enviando y recibiendo múltiples mensajes del servidor.

El manejador del cliente para Capitalize

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;

public class CapitalizeClientHandler implements Runnable {
    private Socket clientSocket;
    private PrintWriter out;
```

```
    private BufferedReader in;

    public CapitalizeClientHandler(Socket socket) {
        this.clientSocket = socket;
    }

    public void run() {
        try {
            out = new
PrintWriter(clientSocket.getOutputStream(), true);
            in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
            String inputLine;
            while ((inputLine = in.readLine()) != null) {
                if (".".equals(inputLine)) {
                    out.println("bye");
                    break;
                }
                out.println(inputLine.toUpperCase());
            }
            in.close();
            out.close();
            clientSocket.close();
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

El cliente de Capitalize podría ser el siguiente:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.Scanner;

public class CapitalizeClient {
    private Socket clientSocket;
    private PrintWriter out;
    private BufferedReader in;

    public static void main(String[] args) throws IOException
{
        if (args.length != 1) {
            System.err.println("Pase la IP del servidor como
único argumento de línea de comandos");
            return;
        }
```

```
        CapitalizeClient capitalizeClient = new
CapitalizeClient();
        capitalizeClient.startConnection(args[0], 55555);
        capitalizeClient.communicate();
        capitalizeClient.stopConnection();
    }

    public void communicate() throws IOException {
        System.out.println("Ingrese líneas de texto. Finalice
con .");
        Scanner scanner = new Scanner(System.in);
        String inputLine;
        String cap;
        while (scanner.hasNextLine()) {
            inputLine = scanner.nextLine();
            cap = this.sendMessage(inputLine);
            System.out.println(cap);
            if (cap.equals("bye")) {
                break;
            }
        }
    }

    public void startConnection(String ip, int port) {
        try {
            clientSocket = new Socket(ip, port);
            out = new
PrintWriter(clientSocket.getOutputStream(), true);
            in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
        } catch (IOException e) {
            //LOG.debug("Error al inicializar la conexión",
e);
        }
    }

    public String sendMessage(String msg) {
        try {
            out.println(msg);
            return in.readLine();
        } catch (Exception e) {
            return null;
        }
    }

    public void stopConnection() {
        try {
            in.close();
            out.close();
            clientSocket.close();
```

```
        } catch (IOException e) {
            //LOG.debug("Error al cerrar la conexión", e);
        }
    }
}
```

Este código proporciona ejemplos de servidores y clientes utilizando comunicación a través de sockets, con manejo de múltiples clientes mediante hilos y un servidor que responde con la fecha o convierte el texto de los clientes a mayúsculas.

Flujos DataInputStream y DataOutputStream en Java

Usando `DataOutputStream` y `DataInputStream`, es posible escribir tipos de datos primitivos (byte, short, int, long, float, double, boolean, char) así como cadenas (Strings) en un archivo binario y leerlos posteriormente.

`DataInputStream` y `DataOutputStream` son clases muy poderosas para leer y escribir tipos de datos primitivos desde y hacia un flujo de datos. Se utilizan en una amplia variedad de aplicaciones, incluidas redes, aplicaciones de bases de datos y procesamiento de archivos.

Métodos de la clase `DataInputStream`:

- public byte readByte()
- public short readShort()
- public int readInt()
- public long readLong()
- public float readFloat()
- public double readDouble()
- public char readChar()
- public boolean readBoolean()
- public String readUTF()
- public String readLine()

Métodos de la clase `DataOutputStream`:

- public void writeByte(byte b)
- public void writeShort(short s)
- public void writeInt(int i)
- public void writeLong(long l)
- public void writeFloat(float f)
- public void writeDouble(double d)
- public void writeChar(char ch)
- public void writeBoolean(boolean b)
- public void writeUTF(String s)
- public void writeBytes(String s)

Flujos `ObjectInputStream` y `ObjectOutputStream` en Java

La serialización es el proceso de convertir un objeto en un formato que pueda ser fácilmente almacenado, transmitido o reconstruido más tarde. La serialización es particularmente útil cuando necesitas guardar el estado de un objeto en un disco, enviarlo a través de una red o pasarlo entre diferentes partes de un programa.

Java emplea su propio protocolo de flujo binario para serialización. La serialización se logra mediante una combinación de las clases `ObjectOutputStream` y `ObjectInputStream`.

Para habilitar la serialización, una clase debe implementar la interfaz `Serializable`.

En el siguiente ejemplo, podemos ver cómo el programa servidor crea un objeto `Person`, le asigna valores y lo envía al programa cliente. El programa cliente realiza cambios en el objeto y lo devuelve modificado al servidor.

Código del servidor:

```
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.ServerSocket;
import java.net.Socket;

public class PersonServer {
    public static void main(String[] args) throws IOException,
ClassNotFoundException {
        int numeroPuerto = 60000; // Puerto
        ServerSocket server = new ServerSocket(numeroPuerto);
        System.out.println("Esperando al cliente...");
        Socket client = server.accept();

        // El flujo de salida está preparado para objetos
        ObjectOutputStream outObject = new
ObjectOutputStream(client.getOutputStream());

        // El objeto se prepara y se envía
        Person per = new Person("Teresa", 20);
        outObject.writeObject(per);

        // Enviar objeto
        System.out.println("Enviado: " + per.getName() + "*" +
per.getAge());

        // Obtener flujo para leer el objeto
        ObjectInputStream inObject = new
ObjectInputStream(client.getInputStream());
        Person person = (Person) inObject.readObject();
        System.out.println("Recibido: " + person.getName() +
"*" + person.getAge());

        // Cerrar flujos y sockets
```

```
        outObject.close();
        inObject.close();
        client.close();
        server.close();
    }
}
```

Código del cliente:

```
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.InetSocketAddress;
import java.net.ServerSocket;
import java.net.Socket;

public class PersonClient {
    public static void main(String[] args) throws IOException,
ClassNotFoundException {
        String host = "localhost";
        int port = 60000; // Puerto remoto

        System.out.println("Cliente iniciado...");
        Socket client = new Socket(host, port);

        // Flujo de entrada para objetos
        ObjectInputStream objIn = new
ObjectInputStream(client.getInputStream());

        // Recibir un objeto
        Person person = (Person) objIn.readObject();

        // Mostrar los datos del objeto
        System.out.println("Recibido: " + person.getName() +
"*" + person.getAge());

        // Modificar el objeto
        person.setName("Teresa Rivas");
        person.setAge(22);

        // Flujo de salida para el objeto
        ObjectOutputStream outPerson = new
ObjectOutputStream(client.getOutputStream());

        // Enviar el objeto
        outPerson.writeObject(person);
        System.out.println("Enviado: " + person.getName() +
"*" + person.getAge());

        // Cerrar flujos y sockets
        objIn.close();
```

```
        outPerson.close();
        client.close();
    }
}
```

La clase `Person`:

```
import java.io.Serializable;

public class Person implements Serializable {
    String name;
    int age;

    public Person(String name, int age) {
        super();
        this.name = name;
        this.age = age;
    }

    public Person() {
        super();
    }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public int getAge() { return age; }
    public void setAge(int age) { this.age = age; }
}
```

Sockets UDP - User Datagram Protocol

UDP es un protocolo alternativo al protocolo TCP, más comúnmente utilizado. Es un protocolo sin conexión, donde se envían paquetes directamente sin necesidad de establecer una conexión adecuada.

Los paquetes UDP tienen encabezados más pequeños en comparación con los encabezados TCP. Además, la comunicación de datos es más rápida, ya que no se intercambia ningún acuse de recibo para garantizar la entrega confiable de los paquetes.

UDP se utiliza para transmisiones de datos sensibles al tiempo, como búsquedas DNS, juegos en línea y transmisión de video. Este protocolo de comunicación mejora las velocidades de transferencia al eliminar la necesidad de una conexión bidireccional formal antes de que comience la transmisión de datos.

En una conexión de red habilitada para UDP, la transmisión de datos comienza sin que la parte receptora se comunique un acuerdo para conectarse. Por lo tanto, las conexiones de red establecidas mediante UDP tienen baja latencia. Además, esto hace que sea imperativo que

las aplicaciones sean tolerantes a la pérdida de datos. UDP se usa a menudo en situaciones donde la velocidad y eficiencia son más importantes que la confiabilidad.

Código del cliente UDP:

```
import java.io.IOException;
import java.net.*;
import java.util.logging.Level;
import java.util.logging.Logger;

public class ClientUDP {
    public static void main(String[] args) {
        // Puerto del servidor
        final int SERVER_PORT = 50000;
        // Buffer para almacenar los mensajes
        byte[] buffer = new byte[1024];

        try {
            // Obtenemos la ubicación del servidor
            InetAddress serverAddress =
InetAddress.getByName("localhost");

            // Creamos el socket UDP
            DatagramSocket socketUDP = new DatagramSocket();
            String message = "¡Hola Mundo desde el cliente!";

            // Convertimos el mensaje a bytes
            buffer = message.getBytes();

            // Creamos el datagrama
            DatagramPacket question = new
DatagramPacket(buffer, buffer.length, serverAddress,
SERVER_PORT);

            // Enviamos el datagrama
            System.out.println("Enviando el datagrama");
            socketUDP.send(question);

            // Preparamos el paquete para la respuesta
            DatagramPacket response = new
DatagramPacket(buffer, buffer.length);

            // Recibimos la respuesta
            socketUDP.receive(response);
            System.out.println("Recibiendo la respuesta");

            // Tomamos los datos y los mostramos
            message = new String(response.getData(), 0,
response.getLength(), "UTF-8");
            System.out.println(message);
```

```
            // Cerramos el socket
            socketUDP.close();
        } catch (SocketException | UnknownHostException |
IOException ex) {

Logger.getLogger(ClientUDP.class.getName()).log(Level.SEVERE,
null, ex);
        }
    }
}
```

Código del servidor UDP:

```
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.SocketException;
import java.util.logging.Level;
import java.util.logging.Logger;

public class ServerUDP {
    public static void main(String[] args) {
        final int PORT = 50000;
        byte[] buffer = new byte[1024];

        try {
            System.out.println("Servidor UDP iniciado");

            // Creación del socket
            DatagramSocket socketUDP = new
DatagramSocket(PORT);

            // Siempre responderá a las solicitudes
            while (true) {
                // Preparamos la respuesta
                DatagramPacket request = new
DatagramPacket(buffer, buffer.length);

                // Recibimos el datagrama
                socketUDP.receive(request);
                System.out.println("Información del cliente
recibida");

                // Convertimos los datos recibidos y mostramos
el mensaje
                String message = new String(request.getData(),
0, request.getLength(), "UTF-8");
                System.out.println(message);
```

```
                // Obtenemos el puerto y la dirección del
cliente
                int clientPort = request.getPort();
                InetAddress address = request.getAddress();

                message = "¡Hola Mundo desde el servidor!";
                buffer = message.getBytes();

                // Creamos el datagrama
                DatagramPacket answer = new
DatagramPacket(buffer, buffer.length, address, clientPort);

                // Enviamos la información
                System.out.println("Enviando información al
cliente");
                socketUDP.send(answer);
            }
        } catch (SocketException | IOException ex) {

Logger.getLogger(ServerUDP.class.getName()).log(Level.SEVERE,
null, ex);
        }
    }
}
```