

MP_0490
Programación de
servicios y procesos



TEMA 2: Programación de comunicaciones en red

Dirección IP y puerto

Para que dos programas puedan comunicarse a través de la red, necesitan de algún sistema para localizarse el uno al otro.

El sistema de localización es a partir de la dirección IP y el número de puerto.

Una dirección IP es un número que identifica de manera única a un equipo dentro de una red.

Por otro lado, una máquina podría tener varios procesos en ejecución utilizando *sockets* y es necesario identificarlos de alguna manera; por ese motivo existe lo que se denomina el número de puerto. Cada número de puerto identifica un *socket* dentro de la misma máquina.

JAVA: La clase *InetSocketAddress*

Existe una clase Java que **representa** una dirección en la red, se trata de la clase *InetSocketAddress* que recibe en el constructor la dirección IP y el número de puerto.

Los objetos *InetSocketAddress* resultan de gran utilidad a la hora de establecer la comunicación mediante **sockets**.

La clase **InetSocketAddress** en Java es una clase de la API de **java.net** que **encapsula** una dirección de Internet y un número de puerto. Se utiliza comúnmente cuando se trabaja con sockets para especificar tanto la dirección IP (o nombre de host) y el puerto de la conexión.

```
InetSocketAddress direccion = new InetSocketAddress("127.191.3.8",2018);
```

Constructores de InetSocketAddress

1. **InetSocketAddress(InetAddress addr, int port):**

- Crea un nuevo socket de dirección IP y un número de puerto a partir de una instancia de **InetAddress**.
- **Parámetros:**
 - **addr:** Un objeto **InetAddress** que representa la dirección IP.
 - **port:** Un número entero que representa el puerto.

2. **InetSocketAddress(String hostname, int port):**

- Crea un nuevo socket de dirección con el nombre de host y el número de puerto dados.
- **Parámetros:**
 - **hostname:** Un string que representa el nombre de host (por ejemplo, "localhost", "example.com") o la dirección IP.
 - **port:** Un número entero que representa el puerto.

3. **InetSocketAddress(int port):**

- Crea un socket de dirección con el puerto especificado y la dirección de bucle invertido (loopback address) como dirección IP.
- **Parámetro:**
 - **port:** Un número entero que representa el puerto.

Métodos principales

1. **getAddress():**

- Devuelve la dirección **InetAddress** asociada con este socket, que puede ser null si fue creado con un nombre de host no resuelto.
- **Valor devuelto:** Un objeto de tipo **InetAddress** o null.

2. **getHostName():**

- Devuelve el nombre de host de la dirección. Si se creó con una dirección IP, este método intenta resolver el nombre de host.
- **Valor devuelto:** Un **String** con el nombre de host.

3. **getPort():**

- Devuelve el número de puerto asociado con este socket.
- **Valor devuelto:** Un entero que representa el puerto.

4. **getHostString():**

- Devuelve el nombre de host asociado o la dirección IP en forma de cadena, sin intentar resolver el nombre si se creó con una dirección IP sin nombre de host.
- **Valor devuelto:** Un **String**.

5. **isUnresolved():**

- Verifica si la dirección de este socket no ha sido resuelta, es decir, si se creó con un nombre de host que aún no ha sido traducido a una dirección IP.
- **Valor devuelto:** Un valor booleano (true si la dirección no está resuelta, false si está resuelta).

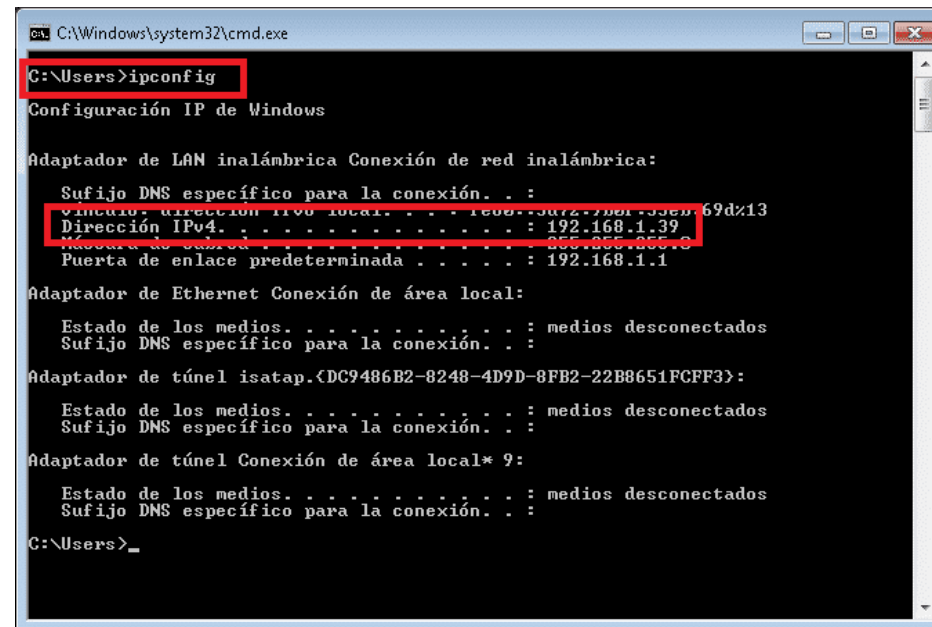
```
import java.net.InetSocketAddress;
public class Main
{
    public static void main(String[] args)
    {
        // Creando una InetSocketAddress con un hostname y puerto
        InetSocketAddress socketAddress1 = new InetSocketAddress("localhost", 8080);
        // Creando una InetSocketAddress con una dirección IP y puerto
        InetSocketAddress socketAddress2 = new InetSocketAddress("192.168.1.1", 8080);
        System.out.println("Hostname: " + socketAddress1.getHostName());
        System.out.println("IP: " + socketAddress2.getAddress());
        System.out.println("Puerto: " + socketAddress1.getPort());
    }
}
```

En este ejemplo, hemos creado dos instancias de `InetSocketAddress`: una con el nombre de host "localhost" y otra con la dirección IP "192.168.1.1", ambas utilizando el puerto 8080.

¿Pero cómo puedo saber cuál es la dirección IP de mi equipo?

Puedes conocer la IP local de tu equipo desde el símbolo del sistema ejecutando el comando ***ipconfig***.

Si tienes varios equipos conectados en una red local, la dirección que aparece en la línea "Dirección IPv4" será la que identifica al ordenador que estás utilizando dentro de la red local.



```
C:\Windows\system32\cmd.exe
C:\Users>ipconfig
Configuración IP de Windows

Adaptador de LAN inalámbrica Conexión de red inalámbrica:
    Sufijo DNS específico para la conexión. . . : 
    Dirección IPv4 local. . . . . : fe80::3a72:7801:53eb:69d%13
    Dirección IPv4. . . . . : 192.168.1.39
    Dirección de enlace. . . . . : fe80::3a72:7801:53eb:69d%13
    Puerta de enlace predeterminada . . . . . : 192.168.1.1

Adaptador de Ethernet Conexión de área local:
    Estado de los medios. . . . . : medios desconectados
    Sufijo DNS específico para la conexión. . . : 

Adaptador de túnel isatap.{DC9486B2-8248-4D9D-8FB2-22B8651FCFF3}:
    Estado de los medios. . . . . : medios desconectados
    Sufijo DNS específico para la conexión. . . : 

Adaptador de túnel Conexión de área local* 9:
    Estado de los medios. . . . . : medios desconectados
    Sufijo DNS específico para la conexión. . . : 

C:\Users>_
```

En Ubuntu y otros sistemas basados en Linux, el comando equivalente a **ipconfig** de Windows es **ifconfig** o **ip**. A continuación te muestro cómo funcionan ambos:

1. **ifconfig**:

- Este es el comando tradicional para mostrar y configurar interfaces de red, aunque en distribuciones más recientes de Linux ha sido reemplazado por el comando **ip**.

Ejecución:

En terminal:

```
ifconfig
```

2. **ip**:

- El comando **ip** es la herramienta más moderna y poderosa para trabajar con configuraciones de red.

Ejecución:

En terminal

```
ip addr show
```

Ambos comandos te mostrarán información sobre las interfaces de red, direcciones IP, máscaras de subred, etc.

InetAddress es una clase en Java que representa una dirección IP y un número de puerto. Es básicamente una combinación de los datos necesarios para definir la **ubicación** de una conexión de red (la dirección y el puerto). No establece una conexión por sí misma; solo guarda la información de la dirección.

1. ifconfig

```
eth0      Link encap:Ethernet  HWaddr 08:00:27:8d:ca:0a
          inet addr:192.168.1.105  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fe8d:ca0a/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:325478 errors:0 dropped:0 overruns:0 frame:0
          TX packets:243236 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:221076182 (221.0 MB)  TX bytes:36163436 (36.1 MB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:42100 errors:0 dropped:0 overruns:0 frame:0
          TX packets:42100 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:29840200 (29.8 MB)  TX bytes:29840200 (29.8 MB)
```

ip addr show

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 08:00:27:8d:ca:0a brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.105/24 brd 192.168.1.255 scope global eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::a00:27ff:fe8d:ca0a/64 scope link
        valid_lft forever preferred_lft forever
```

Existen dos tipos de direcciones IP:

1

La IP privada: se trata de la dirección IP que identifica a tu ordenador dentro de una red local. Como acabamos de comentar anteriormente, puedes saber tu IP privada ejecutando el comando *ipconfig*.

2

La IP pública: es la que identifica tu red desde el exterior, se trata de la IP del router de tu casa. A no ser que dispongas de una IP pública fija, lo más habitual es que cambie cada vez que reinicias el router.

¿Cómo puedo saber mi IP pública?

Existen numerosos sites en internet que te ayudan a averiguar tu IP pública.



The screenshot shows a web browser window with the address bar displaying <https://cual-es-mi-ip-publica.com>. The website has a red header with the title "¿Cual es mi IP Publica?". Below the header, the main content area displays "Mi IP Publica es **82.155.107.228**".

¿Que es una IP Publica?

Una **dirección IP publica** es un número que identifica de manera lógica y jerárquica a una interfaz de un dispositivo (habitualmente un ordenador) dentro de una red, en este caso el numero identifica tu punto de enlace con internet.

Suelen darse dos casos de **IP Publica**

- Si tienes varios ordenadores conectados en red y a su vez a un router la IP Publica la que tiene el router sea de cable o adsl e independiente de los ordenadores que tengas conectados.
- Si por el contrario solo tienes un equipo conectado mediante un modem de cable o adsl, la IP Publica es la que tendrá el ordenador.

¿Como funcionan las direcciones IP?

Una típica dirección IP es en forma decimal más o menos así: 70.35.200.140 - o sea cuatro números, separados con puntos. Los puntos tienen la tarea de comunicar con redes superiores o redes subordinadas.

De la misma manera como en la red mundial de teléfonos un número de teléfono tiene un prefijo para el país, un prefijo para la ciudad, el número del usuario y a veces hasta un número directo de entrada, existe también en internet un prefijo - el número de red, y un número de entrada - el número host.

La primera parte de la dirección IP es el número de red, la segunda parte es el número de host. Donde se encuentra la frontera entre número de red y número de host, es determinada por un esquema de clasificación para tipos de redes.

<https://cual-es-mi-ip-publica.com/>

InetSocketAddress es una clase en Java que representa una dirección IP y un número de puerto. Es básicamente una combinación de los datos necesarios para definir la **ubicación** de una conexión de red (la dirección y el puerto).

```
InetSocketAddress direccion = new InetSocketAddress("127.0.0.1",8080);
```

No establece una conexión por sí misma; solo guarda la información de la dirección.

Socket, por otro lado, es la clase que se usa para **crear una conexión de red real** entre dos dispositivos (cliente y servidor).

```
Socket socket = new Socket("127.0.0.1", 8080);
```

Un Socket utiliza una InetSocketAddress para conectarse a un servidor específico, permitiendo la comunicación bidireccional entre el cliente y el servidor.

```
InetSocketAddress address = new InetSocketAddress("127.0.0.1", 8080);  
Socket socket = new Socket();  
socket.connect(address);
```

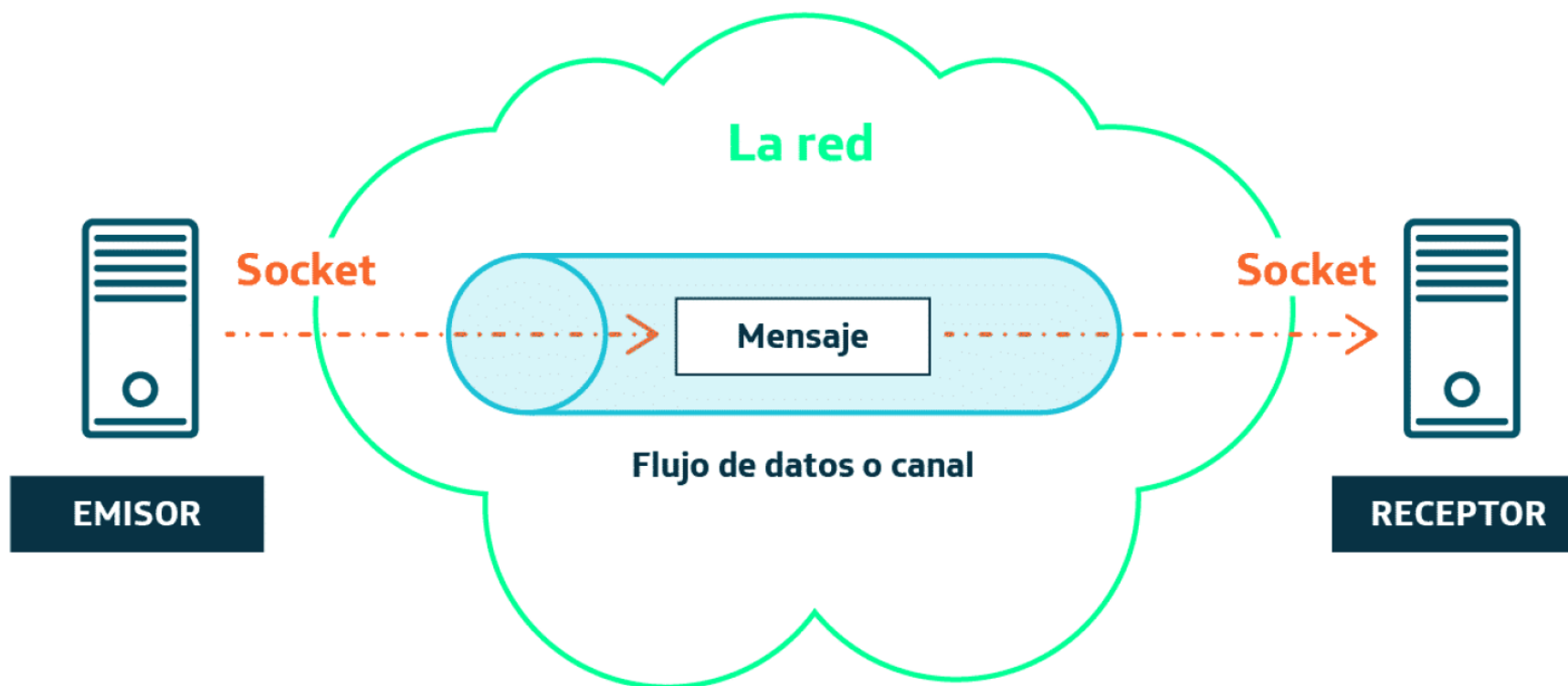
Entonces porqué o cuándo **InetSocketAddress**?

Te permite realizar configuraciones adicionales en el Socket antes de conectarlo, como establecer un tiempo de espera (`socket.setSoTimeout()`), o usar métodos de configuración adicionales.

Configuración de timeout: Puedes usar una versión sobrecargada de `connect` para especificar un tiempo de espera en la conexión:

```
InetSocketAddress address = new InetSocketAddress("127.0.0.1", 8080);  
Socket socket = new Socket();  
socket.connect(address, 5000); // tiempo de espera de 5 segundos
```

La traducción literal de **socket** es "enchufe" y representa un extremo de un canal o flujo de datos que comunicará dos procesos.



Existen dos tipos de **sockets**:

sockets stream

Están orientados a conexión, ya que en la capa de transporte de la pila de protocolos IP utilizan el protocolo TCP , lo que los hace más fiables al gozar de las características de este protocolo; **se garantiza que los mensajes llegan a su destino y en el orden correcto.** Se utilizan en la implementación de aplicaciones cliente / servidor.

sockets datagram

No están orientados a conexión, ya que en la capa de transporte de la pila de protocolos IP utilizan el protocolo UDP, lo que **les resta fiabilidad**: no se garantiza al 100% que los mensajes lleguen a su destino y tampoco se garantiza que lleguen en el orden correcto. En la comunicación mediante *sockets datagram* no existe un rol de cliente y otro rol de servidor.

Toda operación de lectura o escritura requiere del uso de un flujo de datos, también denominado canal o *stream*. Como ya hemos comentado los **sockets** sirven para transmitir información entre dos equipos conectados a la red y evidentemente para realizar esa transmisión también necesitan *streams*.



Flujo de datos o *stream*.

La clase **Socket** en Java es parte del paquete **java.net** y representa un **extremo de una conexión bidireccional** entre dos programas a través de una red (generalmente usando el protocolo TCP). Un socket puede ser utilizado tanto en el lado del cliente como en el lado del servidor para enviar y recibir datos.

Concepto de Socket

Un **socket** es un punto de comunicación entre dos dispositivos a través de una red. En el contexto de la programación, el socket permite que un programa se conecte a otro para intercambiar datos. Cada socket está identificado por dos componentes principales:

- 1.Dirección IP:** La dirección de red del dispositivo.
- 2.Número de puerto:** Un número que identifica la aplicación específica en ese dispositivo con la cual el socket interactuará.

Constructores de la clase Socket

1.Socket():

- Crea un socket sin conectar (debe conectarse después con el método connect()).

2.Socket(String host, int port):

- Crea un socket y se conecta al servidor que se encuentra en la dirección **host** (puede ser una dirección IP o un nombre de dominio) y al número de **puerto** especificado.
- Este constructor es muy común para establecer conexiones con un servidor.

3.Socket(InetAddress address, int port):

- Similar al anterior, pero usa una instancia de la clase **InetAddress** para especificar la dirección del servidor.

4.Socket(String host, int port, InetAddress localAddr, int localPort):

- Conecta el socket a una dirección y puerto específicos, utilizando una dirección IP y puerto locales (por ejemplo, si el equipo tiene varias interfaces de red).

Métodos principales de la clase **Socket**

close():

- Cierra el socket y libera todos los recursos asociados. Una vez cerrado, no se puede volver a utilizar.

isClosed():

- Devuelve un valor booleano indicando si el socket está cerrado o no.

getInetAddress():

- Devuelve la dirección IP del servidor al que el socket está conectado.

getLocalPort():

- Devuelve el número de puerto local al que este socket está enlazado.

connect(SocketAddress endpoint):

- Establece una conexión con un servidor en la dirección especificada. Solo se usa si el socket no se conectó en el momento de la creación.

getInputStream():

- Devuelve un **InputStream** para leer los datos recibidos desde el servidor (o cliente). Este es el flujo que se usa para **recibir** datos del otro extremo de la conexión.

```
InputStream input = clientSocket.getInputStream();
```

getOutputStream():

- Devuelve un **OutputStream** para enviar datos al servidor (o cliente). Este es el flujo que se usa para **enviar** datos al otro extremo de la conexión.

```
OutputStream output = clientSocket.getOutputStream();
```

.SERVIDOR SOCKET

```
import java.net.Socket;
public class EjemploSocketBasico
{
    public static void main(String[] args)
    {
        try
        {
            // Crear un socket y conectarse a la dirección 127.0.0.1 (localhost) en el puerto 8080
            Socket socket = new Socket("127.0.0.1", 8080);
            System.out.println("Socket creado y conectado a 127.0.0.1 en el puerto 8080");
            // Cerrar el socket
            socket.close();
            System.out.println("Socket cerrado.");
        }
        catch (Exception e)
        {
            System.out.println("Error al crear o conectar el socket: " + e.getMessage());
        }
    }
}
```

.CLIENTE SOCKET

```
import java.net.Socket;
public class EjemploClienteSocketBasico
{
    public static void main(String[] args)
    {
        try
        {
            // Crear un socket y conectarse al servidor en 127.0.0.1:8080
            Socket socket = new Socket("127.0.0.1", 8080);
            System.out.println("Conectado al servidor en 127.0.0.1:8080");

            // Cerrar el socket
            socket.close();
            System.out.println("Conexión cerrada.");
        } catch (Exception e)
        {
            System.out.println("No se pudo conectar al servidor: " + e.getMessage());
        }
    }
}
```

NO FUNCIONA TODAVIA, FALTA ALGO!!

La clase **ServerSocket** en Java pertenece al paquete **java.net** y es utilizada para implementar el lado servidor de una conexión TCP/IP. Es decir, un **ServerSocket** escucha conexiones de clientes en un puerto específico y permite aceptar estas conexiones para luego intercambiar datos a través de un **Socket**.

ES DECIR, SI TENEMOS UN SERVIDOR, NECESITAMOS CREAR UN SERVERSOCKET
PARA ESCHUCAR LOS INTENTOS DE CONEXIÓN DE UN CLIENTE

Y UNA VEZ UN CLIENTE CONECTA, CREAMOS UN SOCKET ENTRE EL CLIENTE Y EL
SERVIDOR

EL SOCKET DEBE CREARSE A AMBOS LADOS (TANTO EN LA CLASE CLIENTE COMO EN
LA CLASE SERVIDOR)

Constructores de **ServerSocket**

1. **ServerSocket()** (Constructor por defecto):

- Crea un servidor de sockets no enlazado. Es decir, crea un objeto **ServerSocket** que no está vinculado a un puerto específico. Debes llamar posteriormente a **bind()** para enlazarlo a un puerto.

2. **ServerSocket(int port)**:

- Crea un servidor de sockets que escucha en el puerto especificado.
- **Parámetros:**
 - port: El número de puerto donde el servidor escuchará las conexiones entrantes. Si el valor es 0, se asignará un puerto disponible automáticamente.

3. **ServerSocket(int port, int backlog)**:

- Crea un servidor de sockets que escucha en el puerto especificado y con una cola de espera específica.
- **Parámetros:**
 - port: El número de puerto en el cual se va a enlazar el servidor.
 - backlog: El tamaño máximo de la cola de conexiones entrantes. Si la cola está llena, nuevas conexiones serán rechazadas hasta que haya espacio disponible.

4. **ServerSocket(int port, int backlog, InetAddress bindAddr)**:

- Crea un servidor de sockets que escucha en el puerto especificado, con una cola de conexiones específica, y enlazado a una dirección IP específica.
- **Parámetros:**
 - port: El número de puerto en el cual se va a enlazar el servidor.
 - backlog: El tamaño máximo de la cola de conexiones entrantes.
 - bindAddr: La dirección IP a la que se va a enlazar el socket. Esto es útil si el servidor tiene múltiples interfaces de red.

Métodos Principales de ServerSocket

1.accept():

- Este es uno de los métodos más importantes. Bloquea hasta que se establece una conexión entrante. Cuando un cliente intenta conectarse, el método devuelve un nuevo **Socket** para comunicarse con el cliente.
- Valor devuelto:** Un objeto **Socket** que representa la conexión con el cliente.

2.close():

- Cierra el servidor de sockets. Esto detiene el servidor y libera todos los recursos asociados con él.
- Throws: IOException** si ocurre un error al intentar cerrar el socket.

3.bind(SocketAddress endpoint):

- Enlaza el servidor a una dirección de socket específica, que puede incluir un número de puerto y una dirección IP.
- Parámetro:**
 - endpoint: Un objeto **SocketAddress** que especifica la dirección y puerto a los que se vincula el socket.

4.bind(SocketAddress endpoint, int backlog):

- Similar al anterior, pero también permite especificar el tamaño de la cola de conexiones pendientes (backlog).

- **getInetAddress():**

- Devuelve la dirección de la interfaz a la que el ServerSocket está enlazado (la dirección IP local en la que está escuchando).

- **getLocalPort():**

- Devuelve el número de puerto en el que el socket está escuchando conexiones.

- **setSoTimeout(int timeout):**

- Establece un tiempo límite en milisegundos para el método **accept()**. Si se supera este tiempo sin recibir una conexión, se lanza una excepción **SocketTimeoutException**.

- **Parámetro:**

- timeout: Tiempo en milisegundos de espera. Un valor de 0 indica que no hay límite de tiempo.

- **getSoTimeout():**

- Devuelve el tiempo de espera establecido para las conexiones entrantes a través del método **setSoTimeout()**.

- **setReuseAddress(boolean on):**

- Configura si el ServerSocket puede enlazarse a un puerto que está en el estado de **TIME_WAIT**. Esto es útil si el servidor se reinicia rápidamente y el puerto aún está marcado como "en uso".

- **Parámetro:**

- on: Un valor booleano que indica si se puede reutilizar la dirección (true) o no (false).

.SERVIDOR SOCKET

```
import java.net.ServerSocket;
import java.net.Socket;
public class EjemploServidorSuperbasico {
    public static void main(String[] args) {
        try {
            // Crear un ServerSocket que escuche en el puerto 8080
            ServerSocket serverSocket = new ServerSocket(8080);
            System.out.println("Servidor escuchando en el puerto 8080...");
            // Esperar a que un cliente se conecte
            Socket ClientSocket = serverSocket.accept();
            System.out.println("Cliente conectado.");
            // Cerrar la conexión con el cliente y el servidor
            ClientSocket.close();
            System.out.println("Conexión con el cliente cerrada.");
            serverSocket.close();
            System.out.println("Servidor cerrado.");
        } catch (Exception e) {
            System.out.println("Error en el servidor: " + e.getMessage());
        }
    }
}
```

```
import java.net.ServerSocket;
import java.net.Socket;
public class ServidorSuperBasico {
    public static void main(String[] args) {
        try {
            // Crear un ServerSocket que escuche en el puerto 8080
            ServerSocket serverSocket = new ServerSocket(8080);
            System.out.println("Servidor escuchando en el puerto 8080...");
            // Esperar a que un cliente se conecte
            Socket clientSocket = serverSocket.accept();
            System.out.println("Cliente conectado.");
            // Cerrar la conexión con el cliente y el servidor
            ClientSocket.close();
            System.out.println("Conexión con el cliente cerrada.");
            serverSocket.close();
            System.out.println("Servidor cerrado.");
        } catch (Exception e) {
            System.out.println("Error en el servidor: " + e.getMessage());
        }
    }
}
```

Flujo de trabajo típico de un socket

1. Conexión:

- El cliente crea un socket utilizando un nombre de host (o IP) y un puerto.
- El servidor acepta la conexión utilizando un ServerSocket.

2. Intercambio de datos:

- Una vez que la conexión está establecida, el cliente y el servidor pueden utilizar los flujos de entrada y salida para intercambiar datos (enviar y recibir).

3. Cierre de la conexión:

- Tanto el cliente como el servidor cierran los sockets cuando terminan de comunicarse.

Flujos de datos

Toda la información que se transmite a través de un ordenador fluye desde una entrada hacia una salida.

Para transmitir información, Java utiliza unos objetos especiales denominados *streams* (flujos o corrientes).

Los *stream* permiten transmitir secuencias ordenadas de datos desde un origen a un destino. Para transmitir información de un ordenador a otro, necesitamos un *socket* a cada extremo y cada uno de estos *sockets* suministrará un flujo de entrada o salida para realizar la transferencia.

Java dispone de dos grupos de flujos de datos

- ☐ Flujos de entrada o lectura *input streams*. Para recibir los datos desde otro *socket* emisor.
- ☐ Flujos de salida o escritura *output streams*. Para emitir datos hacia otro *socket* receptor.

Todo proceso de lectura o escritura de datos consta de tres pasos:

- 1 Abrir el **flujo** de datos de lectura o de escritura.
- 2 Leer o escribir datos a través del flujo abierto.
- 3 Cerrar el flujo de datos.

Todas las clases que representan flujos de datos están ubicadas en el paquete *java.io*.

Dentro del paquete *java.io* disponemos de varias clases para representar flujos de datos que están organizadas en dos grandes grupos:



Flujos de datos en formato Unicode de 16 bits: derivados de las clases abstractas *Reader* y *Writer*.



Flujos de bytes (información binaria): derivados de las clases abstractas *InputStream* y *OutputStream*.

UNICODE

Representa más de **143,000 caracteres** de prácticamente todos los lenguajes escritos, símbolos y emojis.

- **Compatibilidad Multilingüe:**

Permite usar múltiples idiomas en un solo sistema sin cambiar la codificación.

- **Escalabilidad:**

Utiliza puntos de código y admite varias formas de codificación (UTF-8, UTF-16, UTF-32).

- **Compatibilidad con ASCII:**

UTF-8 es retrocompatible con ASCII, facilitando la transición de sistemas más antiguos.

- **Consistencia entre plataformas:**

Asegura que los caracteres se representen de la misma manera en cualquier dispositivo o sistema operativo.

- **Soporte de Emojis y Símbolos Especiales:**

Incluye emojis y símbolos modernos, no soportados por codificaciones antiguas.

- **Manejo de Texto Multilingüe y Direccionalidad:**

Soporta escritura compleja y direcciones como de **derecha a izquierda** (árabe, hebreo).

Utilizando sockets stream

Los sockets stream se utilizan en las aplicaciones distribuidas que utilizan un modelo de comunicación de tipo cliente-servidor.

Los *socket stream* tienen las siguientes características:

- Están orientados a conexión, es decir, se establece un canal de comunicación entre dos procesos que se mantendrá abierto durante un cierto tiempo.
- Son fiables, ya que se garantiza que los mensajes llegarán a su destino y en el orden establecido.
- Establecen un modelo de comunicación donde un proceso hace de servidor y otro proceso hace de cliente.

Los métodos **getInputStream()** y **getOutputStream()** son parte de la clase **Socket** en Java, y su objetivo principal es permitir que el programa obtenga flujos de entrada y salida para la comunicación a través del socket.

```
InputStream inputStream = socket.getInputStream();  
OutputStream outputStream = clientSocket.getOutputStream();
```

Las clases `InputStream` y `OutputStream` en Java son clases abstractas en el paquete `java.io` y forman la base de la mayoría de las operaciones de **entrada y salida de datos en Java**. Estas clases se usan para manejar datos como bytes y permiten leer (con `InputStream`) y escribir (con `OutputStream`) datos en archivos, redes, memoria y otros tipos de flujos de datos.

```
import java.net.ServerSocket;
import java.net.Socket;
import java.io.OutputStream;

public class ServidorBasicoConOutputStream {
    public static void main(String[] args) {
        try {
            // Crear un ServerSocket que escuche en el puerto 8080
            ServerSocket serverSocket = new ServerSocket(8080);
            System.out.println("Servidor escuchando en el puerto 8080...");

            // Esperar a que un cliente se conecte
            Socket clientSocket = serverSocket.accept();
            System.out.println("Cliente conectado.");

            // Enviar un mensaje al cliente usando OutputStream
            OutputStream outputStream = clientSocket.getOutputStream();
            String mensaje = "¡Hola, cliente! Este es un mensaje desde el servidor.";
            outputStream.write(mensaje.getBytes()); // Convertir el mensaje a bytes y enviarlo
            outputStream.flush(); // Asegurar que los datos se envíen inmediatamente

            // Cerrar la conexión con el cliente y el servidor
            clientSocket.close();
            System.out.println("Conexión con el cliente cerrada.");

            serverSocket.close();
            System.out.println("Servidor cerrado.");
        } catch (Exception e) {
            System.out.println("Error en el servidor: " + e.getMessage());
        }
    }
}
```

SERVIDOR SOCKET

```
import java.net.Socket;
import java.io.InputStream;

public class ClienteBasicoConInputStream
{
    public static void main(String[] args)
    {
        try
        {
            // Crear un socket y conectarse al servidor en 127.0.0.1:8080
            Socket socket = new Socket("127.0.0.1", 8080);
            System.out.println("Conectado al servidor en 127.0.0.1:8080");

            // Leer el mensaje completo del servidor
            InputStream inputStream = socket.getInputStream();

            byte[] datos = inputStream.readAllBytes(); // Leer todos los bytes disponibles
            String mensaje = new String(datos); // Convertir los bytes a String
            System.out.println("Mensaje del servidor: " + mensaje);

            // Cerrar el socket
            socket.close();
            System.out.println("Conexión cerrada.");
        } catch (Exception e) {
            System.out.println("No se pudo conectar al servidor: " + e.getMessage());
        }
    }
}
```

.CLIENTE SOCKET

UN EJEMPLO MAS DE INTERCAMBIO DE MENSAJES ENTRE EL SERVIDOR Y EL CLIENTE

.SERVIDOR SOCKET

```
import java.net.ServerSocket;
import java.net.Socket;
import java.io.InputStream;
import java.io.OutputStream;

public class ServidorIntercambio {
    public static void main(String[] args) {
        try {
            // Crear un ServerSocket que escuche en el puerto 8080
            ServerSocket serverSocket = new ServerSocket(8080);
            System.out.println("Servidor escuchando en el puerto 8080...");

            // Esperar a que un cliente se conecte
            Socket clientSocket = serverSocket.accept();
            System.out.println("Cliente conectado.");

            // Enviar un mensaje al cliente
            OutputStream outputStream = clientSocket.getOutputStream();
            String mensajeServidor = "¡Hola, cliente! Este es un mensaje desde el servidor.";
            outputStream.write(mensajeServidor.getBytes()); // Enviar el mensaje en bytes
            outputStream.flush(); // Asegurar que los datos se envíen inmediatamente
            System.out.println("Mensaje enviado al cliente.");

            // Recibir el mensaje de respuesta del cliente usando readAllBytes
            InputStream inputStream = clientSocket.getInputStream();
            byte[] datosCliente = inputStream.readAllBytes(); // Leer todos los bytes disponibles
            String mensajeCliente = new String(datosCliente); // Convertir los bytes a String
            System.out.println("Mensaje del cliente: " + mensajeCliente);

            // Cerrar la conexión con el cliente y el servidor
            clientSocket.close();
            System.out.println("Conexión con el cliente cerrada.");
            serverSocket.close();
            System.out.println("Servidor cerrado.");
        } catch (Exception e) {
            System.out.println("Error en el servidor: " + e.getMessage());
        }
    }
}
```

```
import java.net.Socket;
import java.io.InputStream;
import java.io.OutputStream;

public class ClienteIntercambio {
    public static void main(String[] args) {
        try {
            // Crear un socket y conectarse al servidor en 127.0.0.1:8080
            Socket socket = new Socket("127.0.0.1", 8080);
            System.out.println("Conectado al servidor en 127.0.0.1:8080");

            // Leer el mensaje del servidor usando readAllBytes
            InputStream inputStream = socket.getInputStream();
            byte[] datosServidor = inputStream.readAllBytes(); // Leer todos los bytes disponibles
            String mensajeServidor = new String(datosServidor); // Convertir los bytes a String
            System.out.println("Mensaje del servidor: " + mensajeServidor);

            // Enviar un mensaje de respuesta al servidor
            OutputStream outputStream = socket.getOutputStream();
            String mensajeCliente = "¡Hola, servidor! Este es un mensaje desde el cliente.";
            outputStream.write(mensajeCliente.getBytes()); // Enviar el mensaje en bytes
            outputStream.flush();
            System.out.println("Mensaje enviado al servidor.");

            // Cerrar el socket
            socket.close();
            System.out.println("Conexión cerrada.");
        } catch (Exception e) {
            System.out.println("No se pudo conectar al servidor: " + e.getMessage());
        }
    }
}
```

.CLIENTE SOCKET

Por qué falló: `readAllBytes()` esperaba que el flujo se cerrara para saber que la comunicación había terminado, pero el flujo seguía abierto, lo que causó un bloqueo.

Solución: Leer en bloques (con un buffer) permite al programa procesar los datos disponibles sin esperar que el flujo se cierre, funcionando mejor para conexiones de red en tiempo real.

- **Usamos un buffer** (`byte[] buffer = new byte[1024];`) y leemos datos en partes con `InputStream.read(buffer);` en lugar de `readAllBytes()`. Esto permite leer solo los datos que ya han llegado sin esperar que el flujo se cierre.

- **Procesamos solo los datos leídos** usando el tamaño de bytes leídos (`bytesRead`). Convertimos solo esos datos a `String` para evitar leer bytes vacíos del buffer.

```
byte[] datos = inputStream.readAllBytes();  
String mensaje = new String(datos);
```

```
Byte[] buffer = new byte[1024];  
int bytesRead = inputStream.read(buffer);  
String mensaje = new String(buffer, 0, bytesRead);
```

La clase **PrintWriter** en Java es una clase que proporciona una manera conveniente de escribir texto formateado a un flujo de salida (como archivos, sockets o la consola). A diferencia de otras clases como **OutputStream**, que trabajan a nivel de bytes, **PrintWriter** trabaja a nivel de caracteres, lo que lo hace más adecuado para escribir datos de texto.

- **Trabaja con caracteres:**

- **PrintWriter** es parte de las clases que trabajan con texto (a nivel de caracteres), lo que lo diferencia de clases que trabajan a nivel de bytes como **OutputStream**.

- Puedes usar **PrintWriter** para escribir **cadenas de texto, números y caracteres** fácilmente.

- **Automatización del vaciado del buffer (auto-flushing):**

- Una característica opcional de **PrintWriter** es el "**auto-flush**", que puede configurarse para que el contenido del buffer se envíe automáticamente al destino (por ejemplo, a un archivo o socket) cuando se usa uno de los métodos `println()`, `printf()`, o `format()`. Si el "auto-flush" no está habilitado, el buffer no se vaciará automáticamente, lo que puede retrasar la escritura de los datos.

TEMA 2.2: Desarrollo de aplicaciones cliente / servidor con sockets

```
import java.io.PrintWriter;
import java.net.Socket;
public class Cliente
{
    public static void main(String[] args)
    {
        try
        {
            // Conectar al servidor en localhost y puerto 8080
            Socket socket = new Socket("localhost", 8080);
            // Crear un PrintWriter para enviar datos al servidor
            PrintWriter writer = new PrintWriter(socket.getOutputStream(), true);
            // autoFlush habilitado
            // Enviar un mensaje al servidor writer.println("Hola, servidor!");
            // Cerrar el socket y PrintWriter
            writer.close();
            socket.close();
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

La clase **BufferedReader** en Java es parte del paquete **java.io** y se utiliza para **leer texto de un flujo de entrada** de manera eficiente, ya sea desde archivos, sockets o cualquier otro flujo de entrada. Lo hace utilizando un **buffer interno** que almacena temporalmente los datos antes de que sean procesados, lo que reduce el número de llamadas al sistema para leer datos y, por lo tanto, mejora el rendimiento, especialmente cuando se trabaja con operaciones de E/S (Entrada/Salida) que son relativamente lentas, como la lectura de archivos grandes o datos de red.

Buffering (almacenamiento en memoria temporal):

- La clase **BufferedReader** utiliza un **buffer** interno para leer grandes bloques de datos de una vez, en lugar de hacer muchas pequeñas operaciones de lectura directamente desde el origen (como un archivo o una conexión de red). Esto reduce el número de operaciones de E/S y mejora el rendimiento.
- **Lectura por líneas:**
 - Uno de los métodos más útiles de esta clase es **readLine()**, que permite leer una línea completa de texto de una vez. Esto es útil cuando se trabaja con archivos de texto o protocolos de red que envían datos en formato de líneas.

Constructores principales de **BufferedReader**

1. **BufferedReader(Reader in):**

- Crea un **BufferedReader** que envuelve cualquier tipo de **Reader** (como un **FileReader** o un **InputStreamReader**).

```
BufferedReader reader = new BufferedReader(new FileReader("archivo.txt"));
```

2. **BufferedReader(Reader in, int size):**

- Crea un **BufferedReader** con un tamaño de buffer específico. Esto te permite ajustar el tamaño del buffer (en bytes) que se usará para almacenar los datos temporalmente.

```
BufferedReader reader = new BufferedReader(new FileReader("archivo.txt"), 8192); /
```

Métodos principales de **BufferedReader**

1.**read()**:

- Lee un solo carácter y devuelve su valor en forma de entero. Si llega al final del flujo de datos, devuelve -1.
`int caracter = reader.read(); // Lee un carácter`

2.**read(char[] cbuf, int off, int len)**:

- Lee un bloque de caracteres y los almacena en el arreglo **cbuf**, comenzando en la posición **off** y leyendo hasta **len** caracteres.

`char[] buffer = new char[100]; reader.read(buffer, 0, 100); // Lee hasta 100 caracteres`

3.**readLine()**:

- Este método es uno de los más útiles de **BufferedReader**. Lee una **línea completa de texto** y la devuelve como un String. Si se llega al final del archivo o flujo, devuelve null.

`String linea = reader.readLine(); // Lee una línea completa de texto`

4.**ready()**:

- Devuelve un valor booleano que indica si el **BufferedReader** está listo para ser leído (es decir, si hay datos disponibles para ser leídos sin que se bloquee el flujo de ejecución).

`if (reader.ready()) { // Está listo para leer }`

5.**close()**:

- Cierra el **BufferedReader** y libera los recursos asociados con él. Siempre es una buena práctica cerrar los flujos de entrada o salida después de haber terminado de usarlos.

`reader.close();`

TEMA 2.2: Desarrollo de aplicaciones cliente / servidor con sockets

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.Socket;
public class Cliente {
    public static void main(String[] args) {
        try {
            // Conectar al servidor en localhost y puerto 8080
            Socket socket = new Socket("localhost", 8080);

            // Crear un BufferedReader para leer los datos del servidor
            BufferedReader reader = new BufferedReader(new InputStreamReader(socket.getInputStream()));

            // Leer la respuesta del servidor
            String respuesta = reader.readLine();
            System.out.println("Respuesta del servidor: " + respuesta);

            // Cerrar el BufferedReader y el socket
            reader.close();
            socket.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

IMPLEMENTACION SENCILLA CLIENTE/SERVIDOR CON LOS CAMBIOS

.SERVIDOR SOCKET

```
import java.io.*;
import java.net.*;

public class SimpleServer {
    public static void main(String[] args) {
        try {
            // Crea el ServerSocket en el puerto 8080
            ServerSocket serverSocket = new ServerSocket(8080);
            System.out.println("Servidor iniciado, esperando conexión...");
            // Acepta la conexión de un cliente
            Socket clientSocket = serverSocket.accept();
            System.out.println("Cliente conectado desde: " + clientSocket.getInetAddress());
            // Flujo para leer datos del cliente
            BufferedReader input = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
            PrintWriter output = new PrintWriter(clientSocket.getOutputStream(), true);
            // Comunicación sencilla con el cliente
            String clientMessage = input.readLine();
            System.out.println("Mensaje del cliente: " + clientMessage);
            output.println("Hola cliente, el servidor ha recibido tu mensaje.");
            // Cierra el socket del cliente y el servidor
            clientSocket.close();
            serverSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
import java.io.*;
import java.net.*;
public class SimpleClient {
    public static void main(String[] args) {
        try {
            // Conecta al servidor en localhost y puerto 8080
            Socket socket = new Socket("localhost", 8080);
            System.out.println("Conectado al servidor.");
            // Crear un PrintWriter para enviar datos al servidor
            PrintWriter output = new PrintWriter(socket.getOutputStream(), true);
            // Crear un BufferedReader para recibir la respuesta del servidor
            BufferedReader input = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            // Enviar un mensaje al servidor
            output.println("Hola, servidor!");
            // Leer la respuesta del servidor
            String serverResponse = input.readLine();
            System.out.println("Respuesta del servidor: " + serverResponse);
            // Cerrar los recursos
            input.close();
            output.close();
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

.CLIENTE SOCKET

Ejemplo:

Vamos a implementar una aplicación distribuida basada en dos procesos, un cliente y un servidor.

- 1 El servidor quedará escuchando a través de una dirección IP y un puerto a la espera de que un cliente solicite el inicio de una comunicación.
- 2 El cliente solicitará los servicios del servidor iniciando así la comunicación.
- 3 Una vez que el servidor haya aceptado la solicitud del cliente se establecerá una comunicación entre ellos a través de los flujos de datos de entrada y salida.
- 4 El cliente podrá enviar al servidor varios mensajes terminando en el momento en que envíe el texto "FIN". Cuando el cliente envía el mensaje "FIN", el programa cerrará la conexión. Los mensajes los irá introduciendo el usuario por teclado.
- 5 Por cada uno de los mensajes enviados, el servidor responderá al cliente informándole sobre el número de caracteres que componen el mensaje enviado. Cuando el programa servidor recibe el mensaje "FIN", responderá al cliente con el mensaje "Hasta pronto, gracias por establecer conexión" y cerrará la conexión del servidor.

<https://www.youtube.com/watch?v=dJePadx6Nks>


```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.net.ServerSocket;
import java.net.Socket;

public class Servidor {
    public static void main(String[] args) {
        System.out.println("APLICACIÓN DE SERVIDOR");
        System.out.println("-----");

        try {
            ServerSocket servidor = new ServerSocket();
            InetSocketAddress direccion = new InetSocketAddress("192.168.10.128", 2000);
            servidor.bind(direccion);
            System.out.println("Servidor creado y escuchando .... ");
            System.out.println("Dirección IP: " + direccion.getAddress());

            Socket enchufeAlCliente = servidor.accept();
            System.out.println("Comunicación entrante");

            InputStream entrada = enchufeAlCliente.getInputStream();
            OutputStream salida = enchufeAlCliente.getOutputStream();
            String texto = "";
```

```
while (!texto.trim().equals("FIN")) {
    byte[] mensaje = new byte[100];
    int bytesLeidos = entrada.read(mensaje); // Leer solo los bytes recibidos
    texto = new String(mensaje, 0, bytesLeidos); // Convertir solo los bytes leídos a String

    if (texto.trim().equals("FIN")) {
        salida.write("Hasta pronto, gracias por establecer conexión".getBytes());
    } else {
        System.out.println("Cliente dice: " + texto);
        salida.write(("Tu mensaje tiene " + texto.trim().length() + " caracteres").getBytes());
    }
}

entrada.close();
salida.close();
enchufeAlCliente.close();
servidor.close();
System.out.println("Comunicación cerrada");
} catch (IOException e) {
    System.out.println(e.getMessage());
}
}
```

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.Scanner;

public class SocketCliente {
    public static void main(String[] args) {
        System.out.println("APLICACIÓN CLIENTE");
        System.out.println("-----");
        Scanner lector = new Scanner(System.in);

        try {
            Socket cliente = new Socket();
            InetSocketAddress direccionServidor = new InetSocketAddress("192.168.10.128",
2000);
            System.out.println("Esperando a que el servidor acepte la conexión");
            cliente.connect(direccionServidor);
            System.out.println("Comunicación establecida con el servidor");

            InputStream entrada = cliente.getInputStream();
            OutputStream salida = cliente.getOutputStream();
            String texto = "";
```

```
        while (!texto.equals("FIN")) {
            System.out.println("Escribe mensaje (FIN para terminar): ");
            texto = lector.nextLine();
            salida.write(texto.getBytes());

            // Crear un buffer para el mensaje del servidor y leer la respuesta
            byte[] mensaje = new byte[100];
            int bytesLeidos = entrada.read(mensaje); // Leer el número exacto de bytes

            // Convertir solo los bytes leídos a String
            String respuesta = new String(mensaje, 0, bytesLeidos);
            System.out.println("Servidor responde: " + respuesta);
        }

        entrada.close();
        salida.close();
        cliente.close();
        System.out.println("Comunicación cerrada");
    } catch (UnknownHostException e) {
        System.out.println("No se puede establecer comunicación con el servidor");
        System.out.println(e.getMessage());
    } catch (IOException e) {
        System.out.println("Error de E/S");
        System.out.println(e.getMessage());
    }
}
```

MUCHAS GRACIAS POR VUESTRA ATENCIÓN!