

**MP\_0490.**  
**Programación de servicios y procesos**  
**UF2. Programación de comunicaciones en red**

## **2.2. Desarrollo de aplicaciones cliente / servidor con sockets**

# Índice

---

☰	Objetivos	3
☰	¿Qué es un socket?	4
☰	Dirección IP y puerto	6
☰	Flujos de datos	9
☰	Utilizando sockets stream	11
☰	Análisis del programa servidor	16
☰	Análisis del programa cliente	20
☰	Diagrama de funcionamiento de los sockets stream	23
☰	Utilizando sockets datagram	24
☰	Resumen	28

# Objetivos

---

En esta lección perseguimos los siguientes objetivos:

- 1 Conocer el funcionamiento de los *Sockets*.
  - 2 Distinguir entre *sockets stream* y *socket datagram*.
  - 3 Crear programas java que utilicen el modelo cliente-servidor implementado mediante *Sockets*.
  - 4 Crear programas java que se comuniquen utilizando los *sockets* de tipo *datagram*.
- 

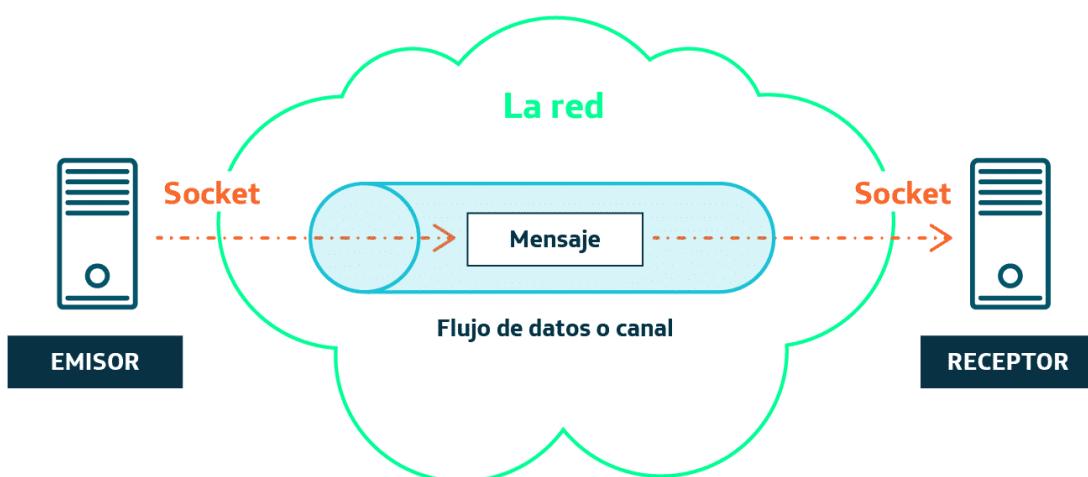
¡Ánimo y adelante!

## ¿Qué es un socket?

Los *sockets* proveen de un mecanismo para la comunicación entre dos procesos que pueden encontrarse en distinta máquina.

Ofrecen una abstracción de la pila de protocolos IP que permite al programador olvidarse de los detalles técnicos de más bajo nivel en las comunicaciones a través de la red.

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



**El uso de sockets es fundamental en los sistemas distribuidos** donde una aplicación está compuesta por varios programas situados en diferentes equipos conectados a través de la red que colaboran y se comunican entre ellos para conseguir un objetivo común.

**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.

# Dirección IP y puerto

---

**Para que un ordenador pueda enviar un mensaje a otro, necesita conocer su dirección.**

## Dirección IP y puerto

Si quieras enviar una carta a un amigo por correo postal necesitarás saber su dirección (calle, número, piso y puerta). De la misma manera, 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.

En conclusión; para que un proceso pueda comunicarse con otro proceso situado en otra máquina de la red necesita saber su dirección completa (número de IP y número de puerto).

**Una dirección IP es una secuencia de 32 bits formada por 4 grupos de 8 bits.** Con 8 bits, es posible representar números con un rango entre 0 y 255. Un ejemplo de dirección IP podría ser: 230.4.27.3.

**El puerto es un número de 16 bits,** lo que significa que puede alcanzar un rango entre 0 y 65535. Lo más habitual es utilizar 4 cifras decimales. Ejemplo: 8085.

## 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.

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

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

## ¿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 . . . . . :
  Vinculo dirección IPv4 local . . . . . : 192.168.1.39
  Dirección IPv4 . . . . . : 192.168.1.39
  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>
```

Ejecución del comando *ipconfig*.

## 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. Pulsa el botón de la derecha y descubrirás uno de ellos.

[VER IP PUBLICA](#)

## Flujos de datos

---

Toda la información que se trasmite 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).

---

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*.

---

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*.



En los siguientes apartados verás cómo se manejan los flujos de datos en los programas con *Sockets*.

# 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.

**Ejemplo:**

---

**Vamos a implementar una aplicación distribuida basada en dos procesos, un cliente y un servidor. Nuestra aplicación funcionará así:**

- 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 cerrara 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.

---

Para que tengas más claro el funcionamiento de los programas que vamos a implementar, el siguiente [video](#) te muestra la ejecución de ambos en dos ventanas de símbolo del sistema diferentes.

Por simplificación a la hora de realizar las pruebas, estamos ejecutando ambos programas en el mismo equipo, aunque podrían ejecutarse en dos equipos distintos conectados a la red, esa es la idea.

## Paso a paso:

1

Comencemos por la implementación de la clase *Servidor*.

Crea un nuevo proyecto en Eclipse con el nombre *SocketServidor* y dentro la clase *Servidor* con el siguiente código:

```
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("192.168.56.1", 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];
                entrada.read(mensaje);
                texto = new String(mensaje);
                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());
        }
    }
}
```

Recuerda sustituir la dirección IP que aparece por la dirección de tu equipo local. Si vas a correr cliente y servidor en el mismo ordenador, puedes utilizar como dirección IP la palabra *localhost* que hace referencia al equipo donde está corriendo el proceso.

2

Y ahora vamos con la implementación de la clase *SocketCliente*.

Si tienes en tu casa dos ordenadores conectados a través de una red local, puedes crear un nuevo proyecto Eclipse en el segundo ordenador llamado *ProyectoSocketCliente* y dentro la clase *SocketCliente*. Si no dispones de dos ordenadores y piensas ejecutar cliente y servidor desde Eclipse, te recomendamos que crees proyectos diferenciados para cliente y servidor y que además cada proyecto esté situado en un workspace distinto. No puedes correr dos programas a la vez en Eclipse, pero si puedes tener corriendo dos Eclipses, cada uno con un workspace distinto, así podrás ejecutar los dos programas a la vez.

```
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("          APPLICACIÓN CLIENTE");
        System.out.println("-----");

        Scanner lector = new Scanner(System.in);
        try {
            Socket cliente = new Socket();
            InetSocketAddress direccionServidor = new InetSocketAddress("192.168.56.1", 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());
                byte[] mensaje = new byte[100];
                entrada.read(mensaje);
                System.out.println("Servidor responde: " + new String(mensaje));
            }
            entrada.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
        salida.close();
        cliente.close();

        System.out.println("Comunicación cerrada");

    } catch (UnknownHostException e) {
        System.out.println("No se puede establecer comunicación con el servi-
dor");
        System.out.println(e.getMessage());
    } catch (IOException e) {
        System.out.println("Error de E/S");
        System.out.println(e.getMessage());
    }
}
```



No te preocupes si no comprendes completamente la implementación de los procesos cliente y servidor. En los próximos apartados lo analizaremos con detenimiento.

# Análisis del programa servidor

En este apartado vamos a analizar detenidamente el proceso que hace el servidor.

1

Creación del objeto *ServerSocket* y establecimiento de la dirección IP y puerto.

```
ServerSocket servidor = new ServerSocket();
InetSocketAddress direccion = new InetSocketAddress("192.168.56.1", 2000);
servidor.bind(direccion);
```

La clase *ServerSocket*, situada en el paquete *java.net* representa un socket o enchufe servidor cuya misión será aceptar la solicitud de un cliente y responder a sus mensajes entrantes.

El servidor quedará a la espera de recibir solicitudes a través de una dirección IP y puerto que hemos establecido a través de un objeto *InetSocketAddress*. El método *bind()* se encarga de vincular el servidor con la dirección IP y puerto establecido en el objeto *InetSocketAddress*.

2

Aceptar la comunicación entrante.

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

El método *accept()* de la clase *ServerSocket* detiene la ejecución del proceso hasta que un cliente solicite una conexión. En el momento en que un cliente solicita comunicación con el servidor, el servidor recogerá el *socket* cliente en el objeto que hemos llamado *enchufeAlCliente*, aceptará la conexión y continuará el proceso mostrando de ese modo en pantalla el mensaje "Comunicación entrante".

3

Recoger los flujos de entrada y salida de datos.

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

El *socket* cliente entrante, cuya referencia hemos llamado *enchufeAlCliente*, es el encargado de devolver los flujos de datos de entrada y salida mediante la llamada a los métodos *getInputStream()* y *getOutputStream()*.

Observa que hemos creado referencias a las clases abstractas *InputStream* y *OutputStream* que representan la abstracción de todos los flujos de datos en formato binario; sin embargo, *getInputStream()* retornará un objeto de la clase *SocketInputStream* y *getOutputStream()* un objeto de la clase *SocketOutputStream*, ambas son implementaciones de las clases abstractas *InputStream* y *OutputStream* para comunicación específicamente mediante sockets.

La siguiente imagen muestra la jerarquía de las clases *SocketOutputStream* y *SocketInputStream* y los paquetes de donde provienen.

org.apache.commons.net.io

### Class OutputStream

```
java.lang.Object
  java.io.OutputStream
    java.io.FilterOutputStream
      org.apache.commons.net.io.SocketOutputStream
```

org.apache.commons.net.io

### Class InputStream

```
java.lang.Object
  java.io.InputStream
    java.io.FilterInputStream
      org.apache.commons.net.io.SocketInputStream
```

4

Comunicación entre cliente y servidor hasta que el cliente envía el mensaje "FIN"

```
String texto = "";
while (!texto.trim().equals("FIN")) {
    byte[] mensaje = new byte[100];
    entrada.read(mensaje);
    texto = new String(mensaje);
    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());
    }
}
```

El servidor irá leyendo los mensajes enviados por el cliente mientras el mensaje leído no contenga el texto "FIN", condición que establecemos en la estructura *while*.

Vamos a analizar por partes el contenido de la estructura *while*.

```
byte[] mensaje = new byte[100];
entrada.read(mensaje);
texto = new String(mensaje);
```

El método *read()* lee los bytes almacenados en el buffer de entrada y los guarda en el array de bytes que se pasa como argumento. El array de bytes *mensaje* finalmente contendrá los códigos numéricos de los caracteres qué componen en mensaje enviado por el cliente. Finalmente el array de bytes es convertido a texto en la expresión *new String(mensaje)*.

```
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());  
}
```

La variable *texto* proviene de la conversión de un array de 100 bytes a *String*. Aunque el mensaje recibido por el cliente ocupe sólo 10 bytes, *texto* tendrá una longitud de 100 caracteres, los que no se utilicen serán espacios en blanco. Por este motivo, se aplica el método *trim()* para recortar los espacios en blanco de la cadena.

## 5

Cierre de los flujos de datos y los *sockets*.

```
entrada.close();  
salida.close();  
enchufeAlCliente.close();  
servidor.close();
```

Una vez que el cliente ha cerrado la comunicación, el proceso servidor también se cierra los flujos de datos y los *sockets*.

# Análisis del programa cliente

Ahora vamos a analizar el programa que hace el rol de cliente solicitando conexión con el servidor.

1

Creación del flujo de datos asociado al teclado (*System.in*).

```
Scanner lector = new Scanner(System.in);
```

Creamos un objeto lector para permitir al usuario introducir por teclado los mensajes que serán enviados al servidor.

2

Crear el socket cliente y establecer comunicación con el servidor.

```
Socket cliente = new Socket();
InetSocketAddress direccionServidor = new InetSocketAddress("192.168.56.1", 2000);
cliente.connect(direccionServidor);
```

La clase *Socket* situada en el paquete *java.net* representa un socket o enchufe cliente cuya misión será solicitar conexión con el servidor con el fin de consumir los servicios que este provee. El método *connect()* intentará conectar con el servidor especificado por el objeto *InetSocketAddress* del argumento.

3

Recoger los flujos de entrada y salida de datos.

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

Igual que en caso del servidor, el cliente necesita sus correspondientes flujos de entrada y salida de datos para establecer la comunicación.

4

Comunicándose con el servidor.

```
String texto = "";
while (!texto.equals("FIN")) {
    System.out.println("Escribe mensaje (FIN para terminar): ");
    texto = lector.nextLine();
    salida.write(texto.getBytes());
    byte[] mensaje = new byte[100];
    entrada.read(mensaje);
    System.out.println("Servidor responde: " + new String(mensaje));
}
```

El usuario tendrá que introducir por teclado los mensajes que serán enviados al servidor. La entrada por teclado se efectúa por medio del método *lector.nextLine()*. El método *nextLine()* de la clase *Scanner* espera la entrada por teclado de una línea de texto y la almacena como un *String* en la variable *texto*. Estamos enviando el texto al servidor con la siguiente sentencia:

```
salida.write(texto.getBytes());
```

El método *write()* envía un array de bytes, no un *String*, por ese motivo ejecutamos el método *getBytes()* sobre el *texto* para obtener un array con los códigos numéricos de los caracteres del objeto *String texto*.

Cuando el servidor recibe el mensaje, responderá, y dicha respuesta es recogida en la variable mensaje mediante este código:

```
byte[] mensaje = new byte[100];  
entrada.read(mensaje);
```

5

Cierre de los flujos de datos y el socket cliente.

```
entrada.close();  
salida.close();  
cliente.close();  
System.out.println("Comunicación cerrada");
```

Cuando el usuario introduce la palabra FIN como mensaje para el servidor, el control del programa saldrá del bucle *while* para cerrar la conexión y los flujos de datos.

# Diagrama de funcionamiento de los sockets stream

En este apartado te mostraremos una representación gráfica de nuestra aplicación cliente-servidor.

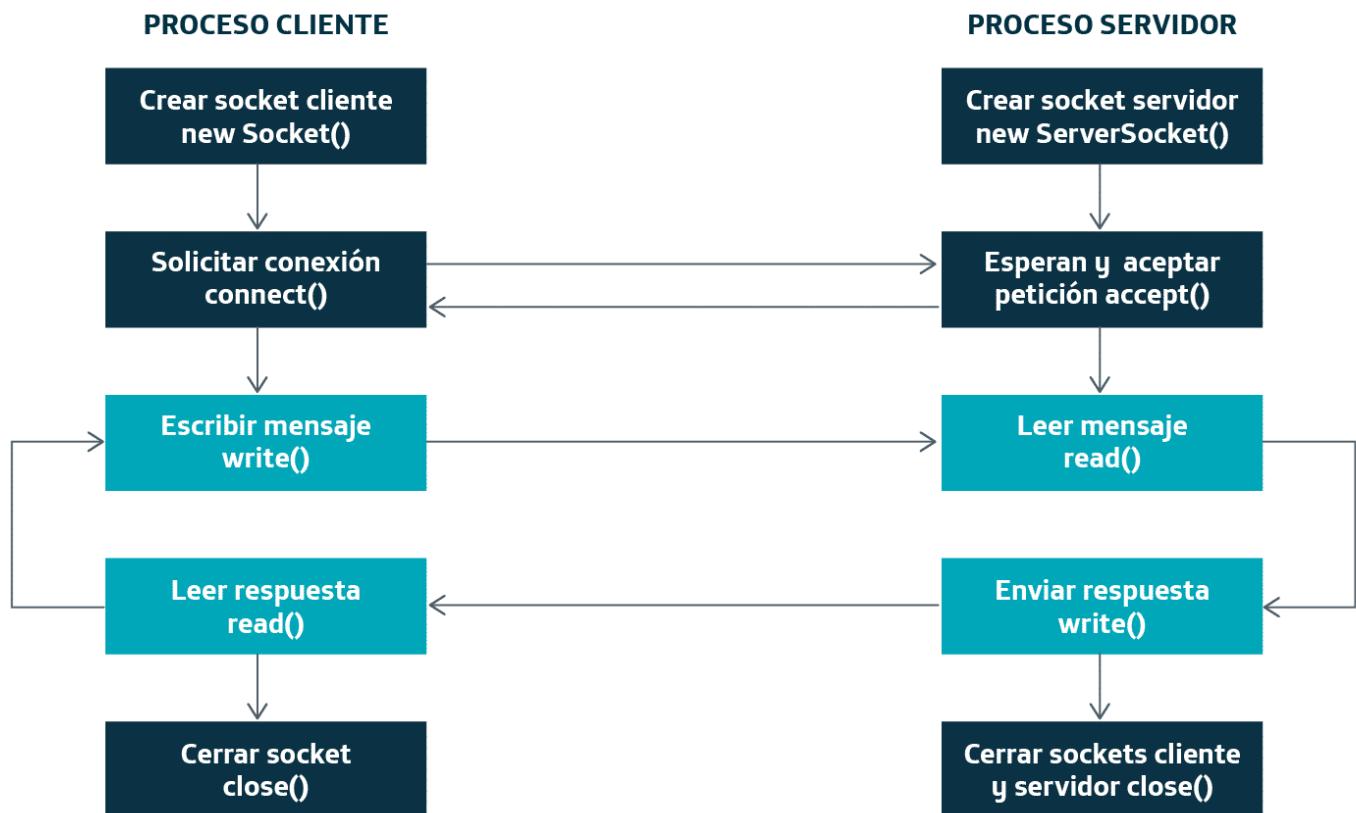


Diagrama de funcionamiento de los sockets stream.

# Utilizando sockets datagram

---

Aunque el principal objetivo de esta lección está en la utilización de los sockets *stream* para diseñar aplicaciones distribuidas cliente-servidor, vamos a ver también las características y un pequeño ejemplo del uso de los sockets *datagram*.

- 1 No están orientados a conexión, es decir, no se abre una comunicación y por lo tanto tampoco se cierra.
- 2 El proceso se parece bastante al envío de una carta postal, escribo una dirección y envío un mensaje a dicha dirección, una vez hecho me despreocupó de lo que ocurra en el destino.
- 3 Utilizan el protocolo UDP en la capa de transporte dentro de la pila de protocolos IP.
- 4 No se puede garantizar al 100% que los mensajes lleguen al destino y tampoco que lleguen en el orden correcto.

---

**El siguiente *ejemplo* muestra un programa basado en un socket *datagram* que envía sucesivos mensajes a un equipo.**

```
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.SocketException;
import java.net.UnknownHostException;
import java.util.Scanner;

public class Principal {

    public static void main(String[] args) {
        Scanner lector = new Scanner(System.in);
        try {
            DatagramSocket ds = new DatagramSocket();
            InetAddress destino = InetAddress.getByName("192.168.56.1");
            String mensaje = "";
            while (!mensaje.equals("FIN")) {
                System.out.println("Escribe un mensaje: ");
                mensaje = lector.nextLine();
                int lon = mensaje.length();
                DatagramPacket carta = new DatagramPacket(mensaje.-getBytes(), lon, destino, 5000);
                ds.send(carta);
                System.out.println("Enviado");
            }
            ds.close();
            System.out.println("Socket Datagram cerrado");
        } catch (SocketException | UnknownHostException e) {
            System.out.println(e.getMessage());
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Este programa se dedica exclusivamente a enviar mensajes, aunque un sólo programa podría enviar y recibir. Enviará los mensajes que escriba el usuario por teclado a la dirección IP = 192.168.56.1 y al puerto 5000. El programa terminará cuando el usuario introduzca el texto FIN.

## Vamos a analizar poco a poco el ejemplo

```
DatagramSocket ds = new DatagramSocket();
```

Creamos el socket datagram como un objeto de la clase *DatagramSocket*.

```
InetAddress destino = InetAddress.getByName("192.168.56.1");
```

La clase *InetAddress* representa una dirección IP.

```
DatagramPacket carta = new DatagramPacket(mensaje.getBytes(), lon, destino,  
5000);  
ds.send(carta);
```

Podríamos decir que esto equivale a enviar una carta. La clase *DatagramPacket* representa un paquete a enviar. Construimos el objeto *carta* con los siguientes argumentos: el mensaje a enviar, la longitud del mensaje, el destino (objeto *InetAddress*), el puerto). Con el método *send()* de la clase *DatagramSocket* enviamos el paquete (objeto *DatagramPacket*) especificado en el argumento.

---

**Ahora vamos a crear otro programa basado en un *socket datagram* que se encargará exclusivamente de recibir mensajes.**

```
import java.io.IOException;  
import java.net.DatagramPacket;  
import java.net.DatagramSocket;  
import java.net.InetSocketAddress;  
import java.net.SocketException;  
import java.net.UnknownHostException;  
  
public class Principal {  
  
    public static void main(String[] args) {  
        try {  
            InetSocketAddress direccion = new InetSocketAddress("192.168.56.1", 5000);  
            DatagramSocket ds = new DatagramSocket(direccion);  
            System.out.println("Preparado para recibir");
```

```
        String texto = "";
        while (!texto.equals("FIN")) {
            byte[] mensaje = new byte[100];
            DatagramPacket carta = new DatagramPacket(mensaje, 100);
            ds.receive(carta);
            texto = new String(mensaje).trim();
            System.out.println("RECIBIDO: " + texto);
        }
        ds.close();
        System.out.println("Socket Datagram cerrado");
    } catch (SocketException | UnknownHostException e) {
        System.out.println(e.getMessage());
    } catch (IOException e) {
        System.out.println(e.getMessage());
    }
}
```

Este programa se dedica exclusivamente a recibir mensajes, algo así como un buzón de correo.. Recibirá mensajes desde la dirección IP = 192.168.56.1 y al puerto 5000. El programa terminará cuando se reciba el mensaje FIN.

## Vamos a analizar el programa anterior

```
InetSocketAddress direccion = new InetSocketAddress("192.168.56.1", 5000);
DatagramSocket ds = new DatagramSocket(direccion);
```

Creamos un objeto *DatagramSocket* capaz de recibir mensajes de la dirección IP = 192.168.56.1 y el puerto 5000.

```
byte[] mensaje = new byte[100];
DatagramPacket carta = new DatagramPacket(mensaje, 100);
ds.receive(carta);
```

Creamos un paquete vacío con una longitud de 100 bytes y lo llenamos en la ejecución del método *receive()* con un paquete recibido a partir del primer programa que hemos creado. El método *receive()* hará una pausa que terminará con la recepción de algún paquete o mensaje.

## Resumen

---

Has terminado la lección, repasemos los puntos más importantes que hemos tratado.

- Los *sockets* proveen de un mecanismo para la comunicación entre dos procesos que pueden encontrarse en distinta máquina.
- Los *sockets stream* se utilizan en las aplicaciones distribuidas de tipo cliente-servidor. En la capa de transporte de la pila de protocolos IP utilizan el protocolo TCP, lo que los hace fiables. En java se implementan a partir de las clases *Socket* y *ServerSocket*.
- Los *sockets datagram* se utilizan en aplicaciones donde se envían mensajes sin que existan roles específicos de cliente-servidor. En la pila de protocolos IP utilizan el protocolo UDP, lo que los hace menos fiables. En java se implementan mediante el uso de la clase *DatagramSocket*.



**PROEDUCA**