# 2. El API de Sockets

Copyright (c) 2025 Adrián Quiroga Linares Lectura y referencia permitidas; reutilización y plagio prohibidos

## 2.1 Introducción

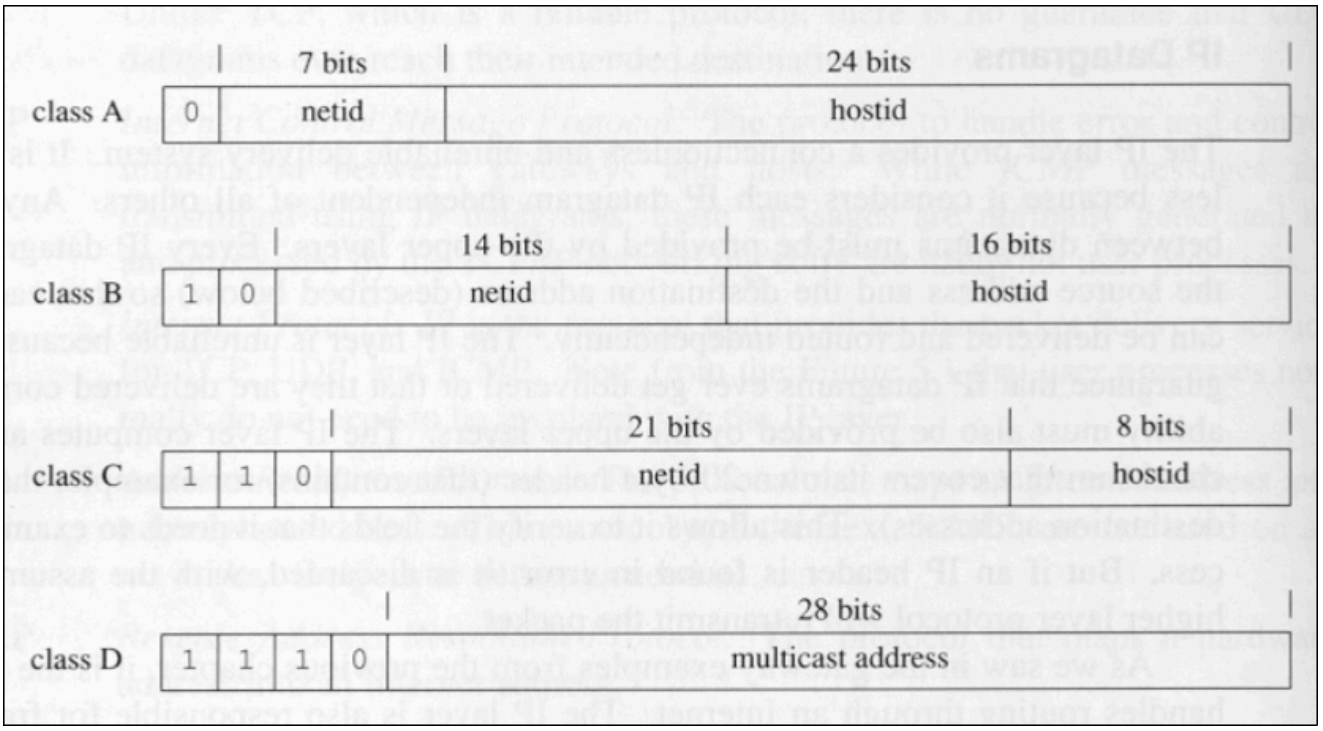
**API de sockets:** interfaz de programación para la comunicación entre procesos (IPC) y ha sido traducido a todos los SO modernos. Es el estándar de facto para la programación IPC y es la base de interfaces IPC más sofisticados tales como Remote Procedure Call (RPC) y Remote Method Invocation (RMI).

## 2.2 Protocolo TPC/IP

El **protocolos TCP/IP** es un protocolos independiente del fabricante, que sirve para permitir que un equipo pueda comunicarse en una red. Está disponible para cualquier ordenador o supercomputador, se utiliza tanto en LANs como WANs y es el más extendido en Internet. Tiene cuatro capas y cada una se construye sobre su predecesora:

- **Capa física:** topología de la red y conexiones globales a la red
- **Capa de datos:** se ocupa del direccionamiento físico, del acceso al medio, de la detección de errores, de la distribución ordenadas de tramas y del control de flujo. Utiliza interfaces de comunicación.
- **Capa de red:** se encarga de identificar el enrutamiento existente entre una o más redes. El objetivo de la capa de red es hacer que los datos lleguen desde el origen al destino (enrutamiento). Aparecen las dirección **IP** y cada ordenador en la red dispone de una dirección única de 32 bits.
- **Capa de transporte:** se encarga de efectuar el transporte de los datos, independizándolos del tipo de red física que esté utilizando. Los datos a través de la red siempre se transmiten en formato Big Endian.

**Clases de redes IP:**



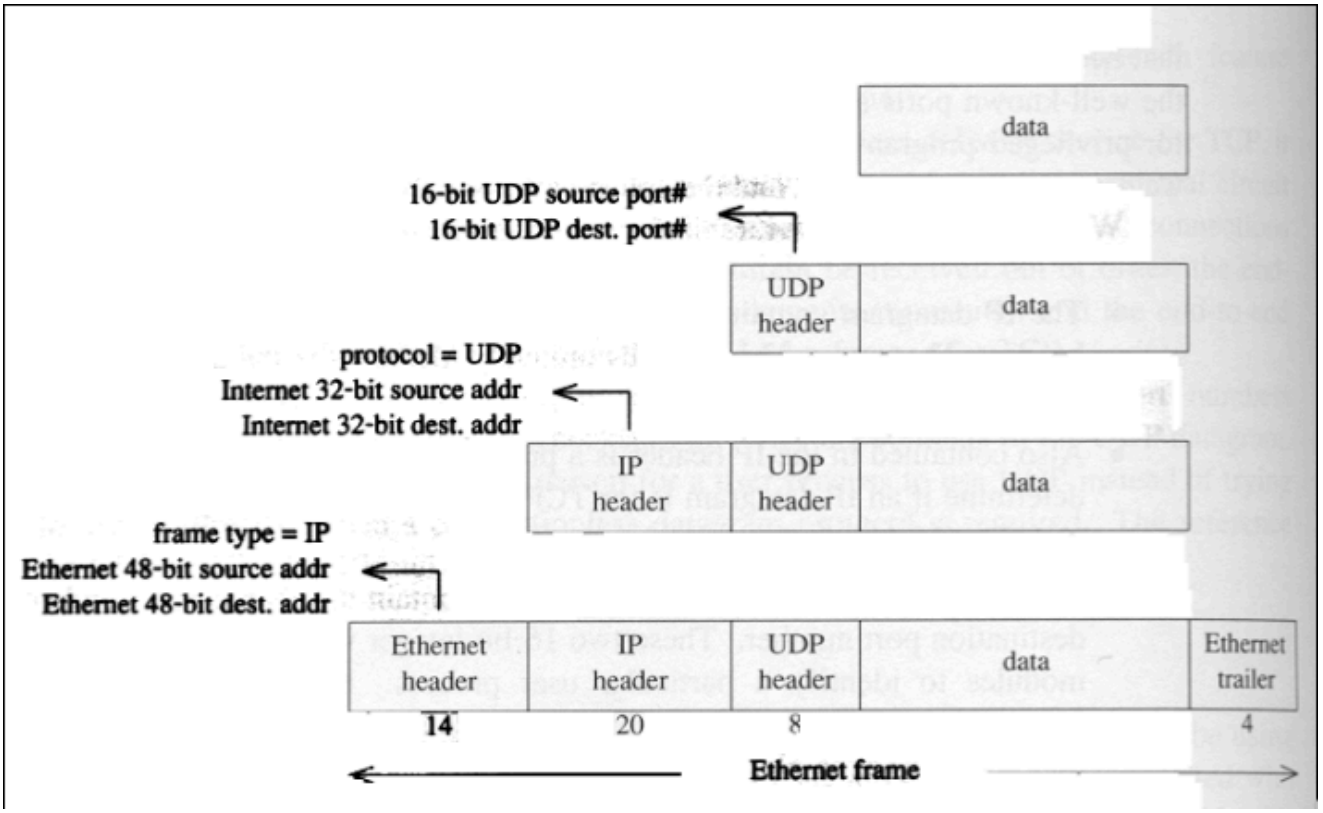

**UDP:** protocolo no orientado a conexión sino al paso de mensajes. No garantiza la correcta recepción de los datos ni el orden de los mismos. Es simple y rápido. UDP utiliza puertos para permitir la comunicación entre aplicaciones. Existen fronteras entre mensajes. Es ideal para apps que requieren alta tasa de transferencia y donde las pérdidas no son importantes. No garantiza un ancho de banda y puede haber retardos.

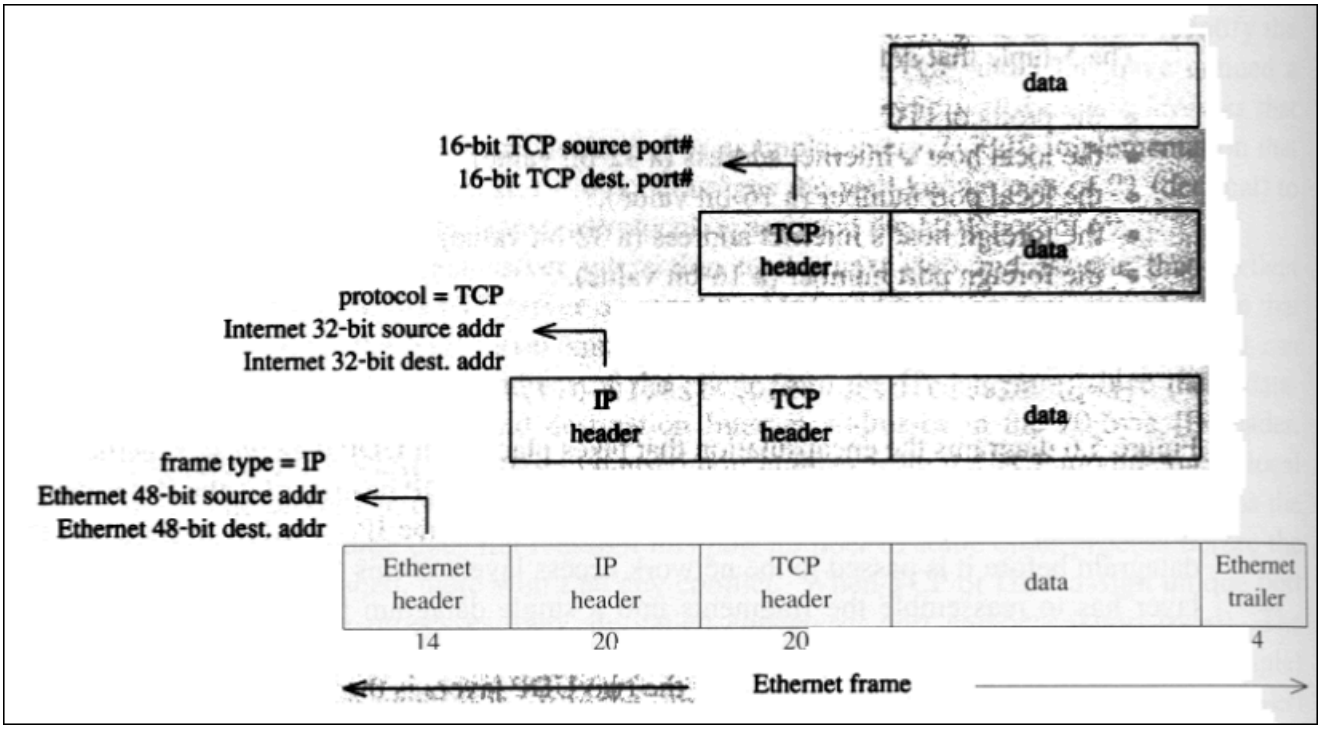
**TCP:** protocolo orientado a conexión que garantiza la correcta recepción de la información y el orden de los paquetes. Se usa para transmisión fiable de información.

A la hora de **encapsular un frame** para convertirlo en un Ethernet frame:

- **Puerto:** entero de 16 bits que se utiliza para identificar sin ambigüedad procesos que intervienen en un diálogo
- **Asociación:** una comunicación en Internet que consta de un protocolo, una dirección IP de la máquina local y otra de la remota de 32 bits y un puerto local y otro remoto de 16 bits
- **Fragmentación:** dividir un mensaje en paquetes porque la mayoría de capas de red tienen un tamaño máximo de paquete que pueden manejar. Los paquetes de un mensaje pueden recibirse en desorden, pero los paquetes tienen un número de orden para reconstruirlos después y detectar fallos de transmisión en el ensamblado.
- **Ensamblado:** reconstruir de nuevo el mensaje ordenando sus paquetes por el número y detección de fallos de transmisión (y solicitarlos de nuevo en caso de que sea necesario).
- **Out of Band (OOB):** mecanismo utilizado para no necesitar una memoria intermedia en la transmisión/recepción de información a través de una interfaz de comunicación. Los datos del canal OOB se envían antes que el resto.

- **IPC de UNIX BSD 4.x:** a partir de BSD 4.2 se implementa una IPC a través de llamadas al sistema. Antes de comunicarse, cada proceso debe crear un **socket**, especificando **protocolo** y **dominio** y se devolverá un **descriptor** del socket susceptible de ser utilizado en llamadas posteriores. Debe tener un id que los demás procesos utilicen como dirección destino. El socket es el encargado del envío y recepción de mensajes que se ponen en una cola hasta que transmiten/extraen.




## 2.3 Comunicación

- **Comunicación por datagramas:**
  - Primitiva de envío de mensajes: `sendto` (especificando mensaje, descriptor de socket propio y destino)
  - Primitiva de recepción de mensajes: `recvfrom` (epecificando socket propio, buffer de almacenamiento y remitente)
- **Comunicación cliente-servidor:**
  - El cliente y servidor crean un socket (`socket()`), asociándole una dirección (`bind()`). El servidor la publica de algún modo y el cliente la obtiene. El cliente envía un `sento()` y el servidor lo recoge con un `recvfrom()`. Para responder sus papeles se invierten
- **Comunicación encauzada (stream):**
  - Crear e identificar el par del **sockets**
  - En este caso el servidor escucha con `listen()`
  - El cliente solicita la conexión: `connect()`, y el servidor aceptará la conexión `accept()`
  - Cuando hay una conexión (*connect-acept*), el SO crea un nuevo socket y los conecta al del cliente de modo que el servidor sigue escuchando en el socket original.
  - Una vez conectados, la comunicación se hace a través de `read()`, `write()`, `send()`, y `recv()`
  - Finalmente, se cierra la conexión `close()`

## 2.4 Servidor y Cliente TCP

| | |
|---|---|
| **Servidor TCP en JAVA** | `ServerSocket s = new ServerSocket(port);`<br>`Socket socket = s.accept();`<br>`DataInputStream input = new DataInputStream(socket.getInputStream());`<br>`DataOutputStream output = new DataOutputStream(socket.getOutputStream());`<br>`System.out.println(input.readByte());`<br>`output.writeInt(8);`<br>`socket.close();`<br>`s.close();` |
| **Clientes TCP en JAVA** | `Socket socket = new Socket("usc.es", port);`<br>`DataInputStream input = new DataInputStream(socket.getInputStream());`<br>`DataOutputStream output = new DataOutputStream(socket.getOutputStream());`<br>`output.writeByte(5);`<br>`System.out.println(input.readInt());`<br>`socket.close();` |

1. **El servidor** abre el ServerSocket y espera.
2. **El cliente** crea su Socket y llama al servidor.
3. El ServerSocket **acepta** la llamada y crea su propio Socket para hablar con ese cliente.
4. Ahora **ambos tienen su propio Socket** y pueden enviarse mensajes.

**Sockets no orientados a conexión:** socket con lo que es posible que múltiples procesos simultáneamente envíen datagramas al mismo socket creado por el proceso receptor.

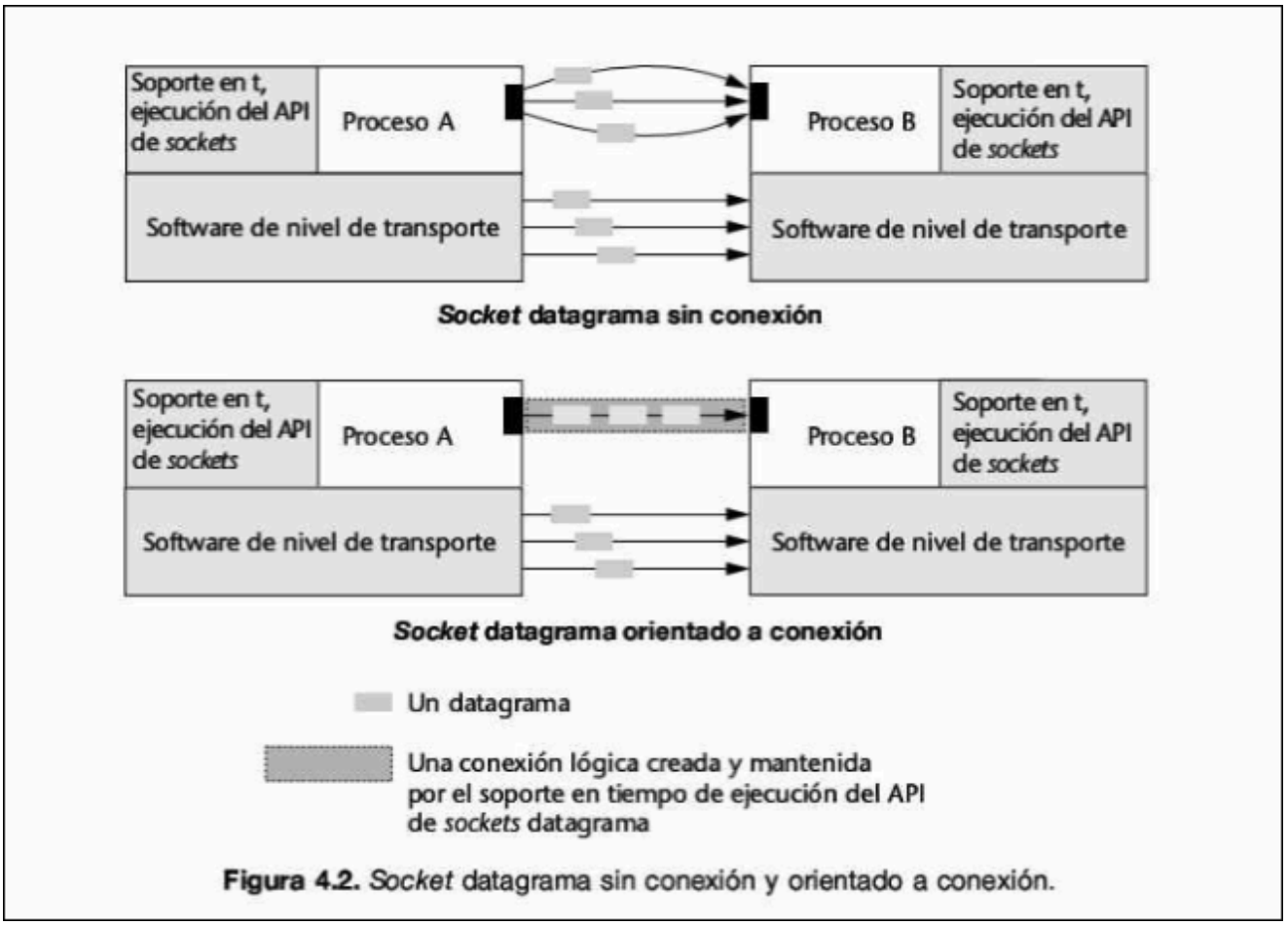



**Figura 4.2.** *Socket* datagrama sin conexión y orientado a conexión.

# 2.5 Sockets Datagramas en Java (UDP)

En Java tenemos dos clases proporcionadas por el API de sockets para datagramas:

- La clase `DatagramSocket` para especificar el **socket**
- La clase `DatagramPacket` para representar al **datagrama** intercambiado

Un proceso que desee enviar o recibir datos usando esta API debe crear una instancia de `DatagramSocket`. Cada socket está ligado a un puerto UDP de la máquina local donde reside el proceso.

Para enviar un datagrama a otro proceso, un proceso hay que crear un objeto que represente el datagrama (`DatagramPacket`) que contenga los datos y la dirección de destino . Se emite una llamada a `send()` del `DatagramSocket` cuyo argumento es el `DatagramPacket`. En el proceso receptor se debe instanciar un objeto de tipo `DatagramSocket` y ligarlos a un puerto local. Para recibir datagramas enviados al socket, el proceso crea un objeto de tipo `DatagramPacket` que referencia a un *array de bytes* y llamar al método `receive` del objeto `DatagramSocket`, especificando como argumento una referencia al objeto `DatagramPacket`.

**Timeout:** se usa para **evitar bloqueos** indefinidos.

**Datos:**

- Tanto bajo TCP como UDP existe un rango de **puertos reservados** [1,1023]. Unicamente los procesos con permisos de root podrán generar un socket en uno de esos puertos.
- El **puerto** siempre se fija en la parte del **servidor**. El cliente genera su puerto de forma automática
- Por defecto, un socket siempre es **bloqueante**, de forma que cuando realizamos una operación de lectura sobre el mismo, se suspende la ejecución del proceso si no hay datos en el buffer.
- Suele ser preferible generar un proceso aparte para leer datos de un socket que ponerlo en modo no bloqueante y gestionar su lectura por interrupciones.

| | |
|---|---|
| **Programa receptor** | `DatagramSocket ds = new DatagramSocket(2345);`<br>`InetAddress receiverHost = InetAddress.getByName("localHost");`<br>`DatagramPacket dp = new DatagramPacket(buffer, MAXLEN);`<br>`ds.receive(dp);` |
| **Programa emisor** | `InetAddress receiverHost = InetAddress.getByName("localhost");`<br>`DatagramSocket theSocket = new DatagramSocket();`<br>`String message = "Hola mundo";`<br>`byte[] data = message.getBytes();`<br>`DatagramPacket paq = new DatagramPacket(data, data.length, receiverHost, 2345);`<br>`theSocket.send(theOutput);` |

- El **emisor** crea un paquete con el mensaje y lo envía al receptor (puerto 2345).
- El **receptor** espera recibir el paquete en ese puerto y lo almacena al llegar.

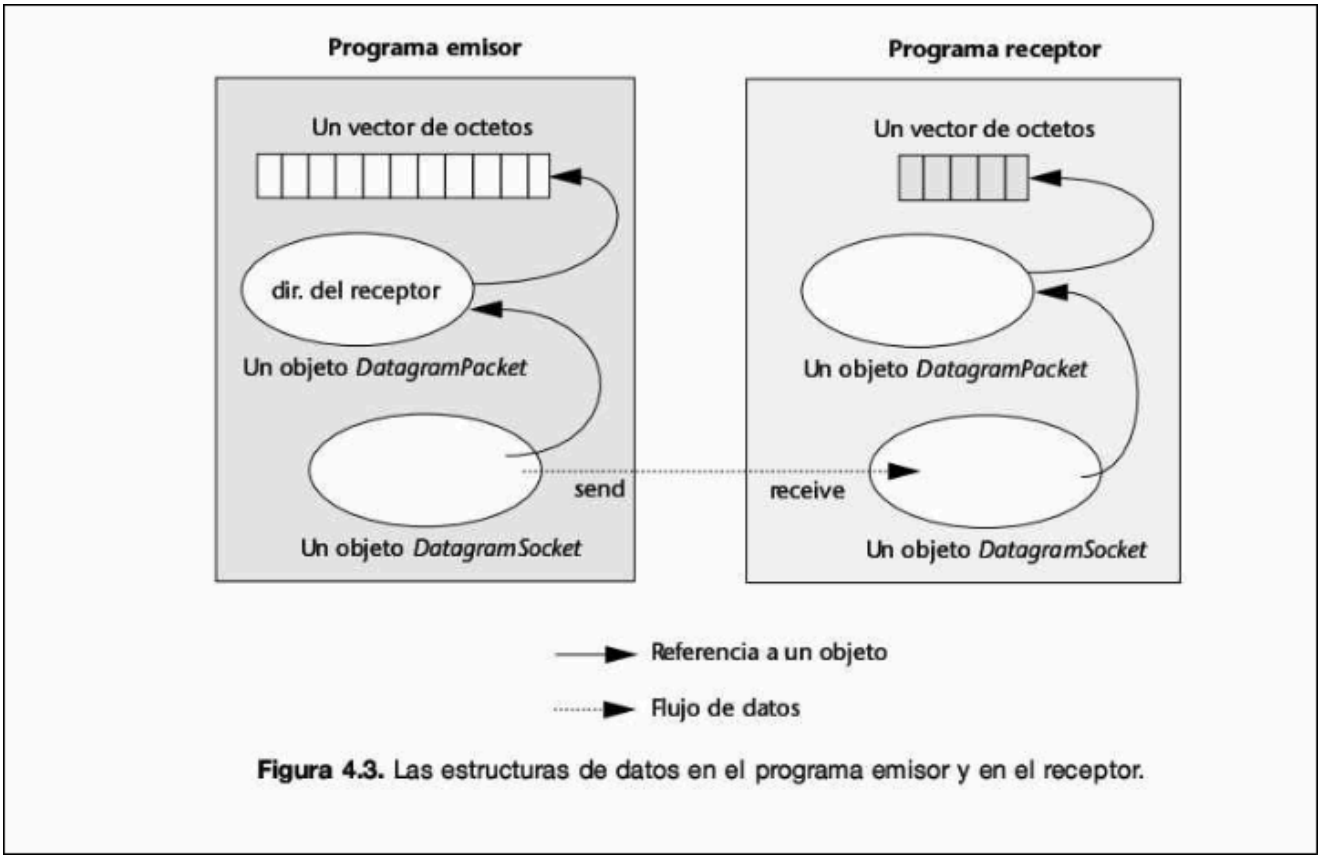



**Figura 4.3.** Las estructuras de datos en el programa emisor y en el receptor.

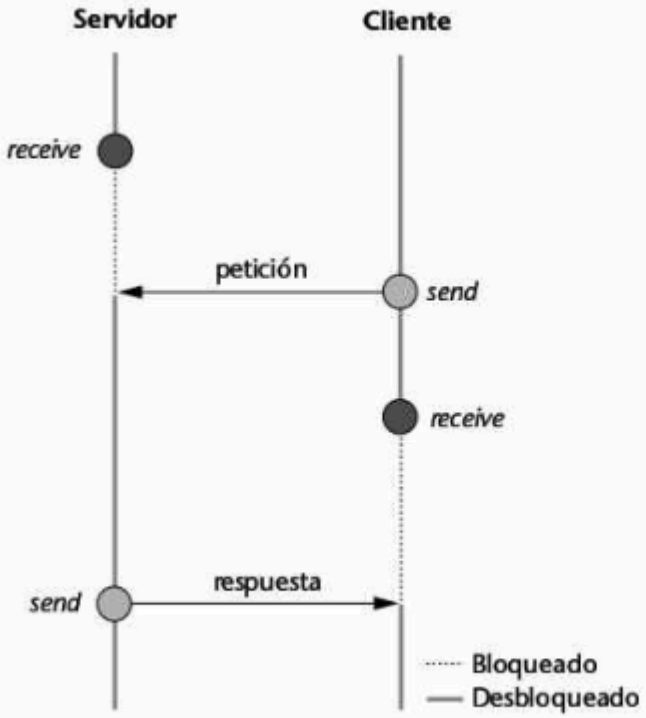



**Figura 4.7.** Sincronización de eventos con *sockets* sin conexión.

## 2.6 API de Sockets en Modo Stream

**API de sockets:** en modo stream proporciona un modelo para la transmisión de datos basado en el **modo encauzado de E/S** proporcionado por los SO Unix. Da únicamente soporte a comunicación orientada o **conexión**.

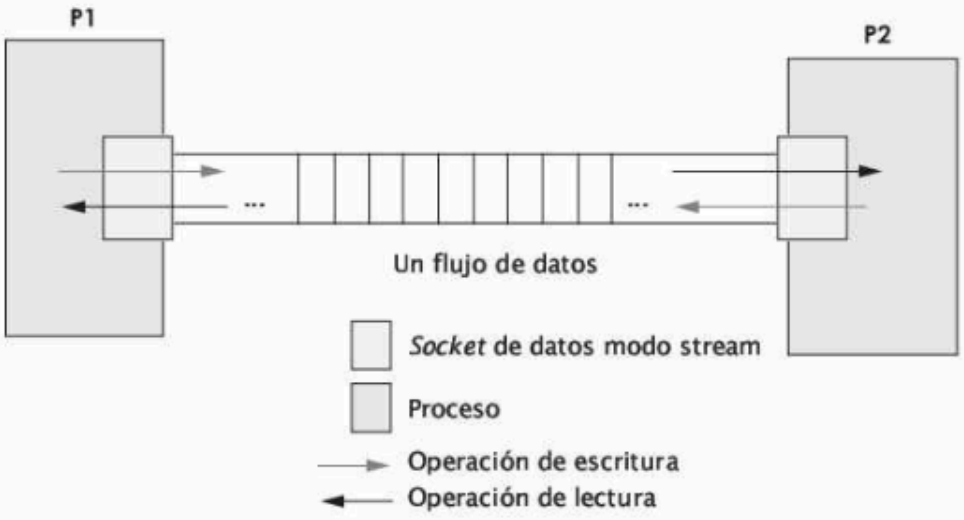



**Figura 4.15.** Uso de un *socket* en modo *stream* para transferencia de datos.

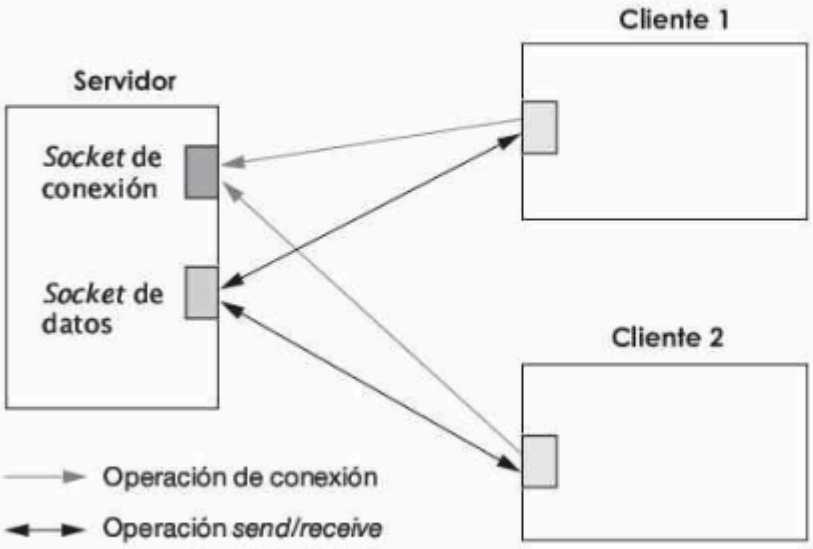



**Figura 4.17.** El API de *socket* en modo *stream*.

Un socket en modo stream se crea para el intercambio de datos entre dos procesos específicos. Los datos se escriben en un extremo del socket y se leen en el otro extremo. Un socket stream no puede utilizarse para comunicarnos con más de un proceso. *Receive* y *accept* son bloqueantes; *send* es no bloqueante.
En Java, el API de sockets en modo stream se proporciona mediante dos clases:

- **ServerSocket:** para aceptar conexiones; llamaremos a un objeto de esta clase un socket de conexión
- **Socket:** para el intercambio de datos; llamaremos a un objeto de esta clase un socket de datos

| | |
|---|---|
| **Aceptador de conexión** | `ServerSocket connectionSocket = new ServerSocket(port);`<br>`Socket dataSocket = connectionSocket.accept();`<br>`OutputStream outStream = dataSocket.getOutputStream();`<br>`PrintWriter socketOutput = new PrintWriter(new OutputStreamWriter(outStream));`<br>`socketOutput.println(message);`<br>`dataSocket.close();`<br>`connectionSocket.close();` |
| **Receptor de conexión** | `InetAddress acceptorHost = InetAddress.getByName(args[0]);`<br>`int acceptorPort = Integer.parseInt(port);`<br>`Socket mySocket = new Socket(acceptorHost, acceptorPort);`<br>`InputStream inStream = mySocket.getInputStream();`<br>`BufferedReader socketInput = new BufferedReader(new InputStreamReader(inStream));`<br>`String message = socketInput.readLine();`<br>`mySocket.close();` |

## 2.7 API de Sockets Seguros

**Sockets seguros:** estos sockets realizan un **cifrado** de los **datos** transmitidos.

El paquete Java TM Secure Socket Extension (**JJSE**) permite comunicaciones seguras en internet, implementa una versión Java de los protocolos **SSL** y **TLS** e incluye funcionalidades de cifrado, autentificación e integridad.

# 2.8 Multidifusión

**Multidifusión:** la multidifusión IP se construye sobre el protocolos IP.

La multidifusión IP permite que el emisor transmita un único paquete IP a un conjunto de computadores que forman un grupo de multidifusión. El emisor no está al tanto de las identidades de los receptores y del tamaño del grupo. Los grupos de multidifusión se especifican utilizando direcciones Internet de la clase D.

La pertenencia a los grupos de multidifusión es dinámica, permitiendo que los computadores se apuntes o borren a un número arbitrario de grupos en cualquier instante. Es posible enviar mensajes a un grupo de multidifusión sin pertenecer al mismo, pero debes ser miembro para recibir paquetes.

## Multidifusión en IPv4

- **Routers multidifusión:** los paquetes IP pueden multidifundirse tanto en una red local como en todo Internet. Si la multidifusión va dirigida a internet, debe hacer uso de las capacidades de multidifusión de los routers, los cuales reenvían los datagramas únicamente a otros routers de redes que pertenezcan al mismo grupo. Para limitar la distancia de propagación de un datagrama multidifusión, el emisor puede especificar el número de routers que puede cruzar (TTL)
- **Reserva de direcciones multidifusión:** las direcciones multidifusión se pueden reservar de forma temporal o permanente. Existen grupos permanentes, incluso sin ningún miembro. Sus direcciones son asignadas por la autoridad de Internet en el rango de `224.0.0.1` a `224.0.0.255`. El resto de las direcciones multidifusión están disponibles para su uso por parte de grupos temporales. Cuando se crea un grupo temporal se necesita una dirección multidifusión libre para evitar conflictos. El protocolo de multidifusión IP no resuelve el problema de reservar la dirección. Cuando la comunicación es a nivel local, si se pone TTL pequeño, es difícil que entremos en conflicto.

  Si se necesita multidifusión a nivel de Internet es necesario reservar previamente una dirección. El programa de directorio de sesiones sirve para arrancar o unirse a una sesión multidifusión. Proporciona una herramienta que permite a los usuarios detectar sesiones multidifusión existentes y anunicar su propia sesión, especificando el tiempo y la duración de la reserva. Para España quien gestiona dicha agenda el RedIris

| Multidifusión en JAVA | |
|---|---|
| **Enviar** | `MulticastSocket s = MulticastSocket(6789);`<br>`InetAddress group = InetAddress.getByName(args[1]);`<br>`s.joinGroup(group);`<br>`byte [] m = args[0].getBytes();`<br>`DatagramPacket messageOut = new DatagramPacket(m, m.length, group, 6789);`<br>`s.send(messageOut);` |
| **Recibir** | `byte[] buffer = new byte[1000];`<br>`for(int i = 0; i < n; i++) {`<br>`  DatagramPacket messageIn = new DatagramPacket(buffer, buffer.length);`<br>`  s.receive(messageIn);`<br>`}`<br>`s.leaveGroup(group);` |

# 2.9 Hilos en Java

Un **hilo** es una secuencia de control dentro de un proceso que ejecuta instrucciones de manera independiente.

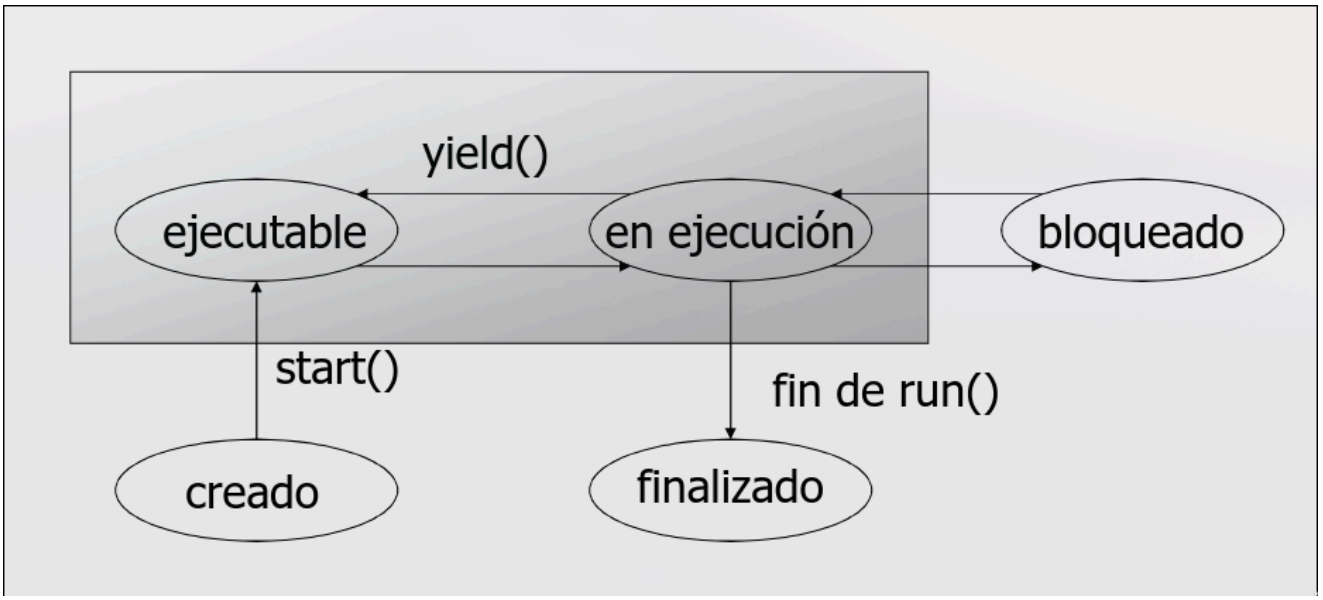
- Los procesos son *entidades pesadas*: requieren llamadas al sistema y cambios de contexto costosos.
- Los hilos son *entidades ligeras*: comparten el espacio de memoria del proceso y su cambio de contexto es menos costoso.

👉 En Java, la **JVM crea un hilo principal** para ejecutar el método `main()`. A partir de ahí podemos crear más hilos según lo necesitemos.

## Estados de un hilo en Java

Un hilo en Java pasa por diferentes **estados**:

1. **Creado** → cuando instanciamos el objeto `Thread`. Se activa con `start()`.
2. **Ejecutable** → listo para ejecutarse cuando la JVM lo decida.
3. **En ejecución** → corriendo su método `run()`. Puede ceder el turno con `yield()`.
4. **Bloqueado** → esperando por: E/S, `sleep()`, `join()`, `wait()`.
5. **Finalizado** → terminó la ejecución de `run()`, y la JVM libera sus recursos.


yield()
ejecutable
en ejecución
bloqueado
start()
fin de run()
creado
finalizado

# Cómo crear hilos en Java

## Opción A: Extender la clase `Thread`

JAVA

```java
public class MiHilo extends Thread {
    String mensaje;
    int veces;

    public MiHilo(String nombre, String msg, int v) {
        super(nombre); // se asigna el nombre al hilo
        mensaje = msg;
        veces = v;
    }

    public void run() {
        for (int i = 1; i <= veces; i++) {
            System.out.println("Thread " + getName() + " imprime " + mensaje
+ " i=" + i);
        }
    }
}

public class PruebaMiHilo {
    public static void main(String[] args) {
        Thread h1 = new MiHilo("Hilo 1", "Hola", 5);
        Thread h2 = new MiHilo("Hilo 2", "Mundo", 5);

        h1.start();
        h2.start();
    }
}
```

### Opción B: Implementar la interfaz `Runnable`

```java
public class MiHilo implements Runnable {
    public void run() {
        System.out.println("Hola desde un hilo Runnable");
    }
}

public class Prueba {
    public static void main(String[] args) {
        Thread t = new Thread(new MiHilo());
        t.start();
    }
}
```

## Acceso a la sección crítica

Cuando varios hilos acceden a recursos compartidos, pueden aparecer problemas de **inconsistencia**.
Para solucionarlo, Java ofrece **cerrojos (locks)** implícitos a través de la palabra clave `synchronized`.

### Bloque sincronizado

```java
synchronized(objetoCerrojo) {
    // sección crítica
}
```

### Método sincronizado

```java
public synchronized void metodoCritico() {
    // sección crítica
}
```

Esto garantiza que **solo un hilo a la vez** ejecute ese bloque o método sobre el mismo objeto.

## Monitores y Productor-Consumidor

Un **monitor** en Java se implementa con clases sincronizadas (`synchronized`).
Ejemplo clásico: **buffer circular** para el problema productor-consumidor.

## Buffer circular

JAVA

```java
public class BufferCircular {
    int elementos = 0;
    int pin = 0, pout = 0;
    Object[] buffer;

    public BufferCircular(int tam) {
        buffer = new Object[tam];
    }

    public synchronized void insertar(Object item) throws
InterruptedException {
        while (elementos == buffer.length) wait(); // buffer lleno
        buffer[pin] = item;
        pin = (pin + 1) % buffer.length;
        elementos++;
        notifyAll();
    }

    public synchronized Object extraer() throws InterruptedException {
        while (elementos == 0) wait(); // buffer vacío
        Object item = buffer[pout];
        pout = (pout + 1) % buffer.length;
        elementos--;
        notifyAll();
        return item;
    }
}
```

## Productor y Consumidor

JAVA

```java
public class Productor extends Thread {
    BufferCircular buffer;
    public Productor(BufferCircular b) { buffer = b; }

    public void run() {
        try {
            for (int i = 0; i < 10; i++) {
                String item = "Item" + i;
                System.out.println("Produce " + item);
                buffer.insertar(item);
            }
        } catch (InterruptedException e) { }
    }
}

public class Consumidor extends Thread {
    BufferCircular buffer;
    public Consumidor(BufferCircular b) { buffer = b; }

    public void run() {
        try {
            for (int i = 0; i < 10; i++) {
                String item = (String) buffer.extraer();
                System.out.println("Consume " + item);
            }
        } catch (InterruptedException e) { }
    }
}
```

## Métodos importantes de `Thread`

Algunos métodos clave:

- `start()` → inicia el hilo.
- `run()` → código del hilo (no invocar directamente).
- `sleep(ms)` → suspende el hilo.
- `join()` → espera a que un hilo termine.
- `yield()` → cede el turno a otro hilo de misma prioridad.
- `wait()` / `notify()` / `notifyAll()` → coordinación entre hilos.
- `currentThread()` → devuelve el hilo en ejecución.

- `setName()` / `getName()` → cambiar o leer nombre del hilo.

## Resumen

- Los **hilos** permiten ejecutar varias tareas concurrentemente dentro de un proceso.
- En Java se crean **extendiéndo** `Thread` o **implementando** `Runnable`.
- Tienen **ciclo de vida** con estados bien definidos.
- El acceso a recursos compartidos debe controlarse con **synchronized**.
- La comunicación entre hilos se hace con **wait() / notify()**.
- Ejemplo clásico: **Productor-Consumidor con buffer circular**.