# Arquitectura de Sistemas

## Paso de mensajes

---

Gustavo Romero López

Updated: 16 de mayo de 2025

Departamento de Ingeniería de Computadores, Automática y Robótica

# Índice

# Lecturas recomendadas

| | |
|---|---|
| J. Bacon | Operating Systems (12, 13, 14) |
| A. Silberschatz | Fundamentos de Sistemas Operativos (3.6, 16) |
| W. Stallings | Sistemas Operativos (5, 13) |
| A. Tanuenbaum | Sistemas Operativos Modernos (2, 8.2.4, 10) |

# Motivación

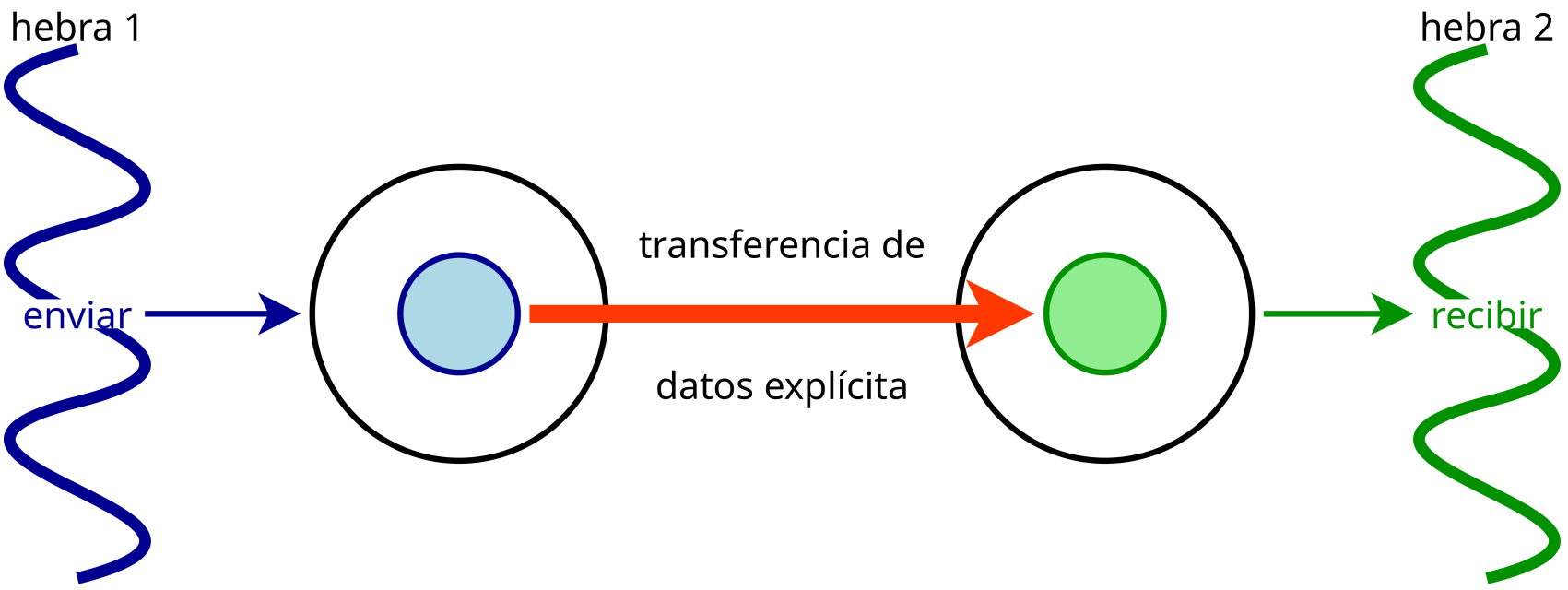
- ¿De verdad necesitamos otro mecanismo de sincronización como el **paso de mensajes**?

- **Si**, por supuesto, porque...
  - Los mecanismos ya vistos requieren **memoria compartida** $\Longrightarrow$ estas soluciones no funcionarán en **sistemas distribuidos**.
  - Las hebras de diferentes aplicaciones necesitan **protección** $\Longrightarrow$ incluso en sistemas con memoria compartida no siempre querremos abrir nuestro espacio de direcciones a terceros en los que podríamos **no confiar**.
  - Para conseguir que el **núcleo** del sistema operativo sea **muy pequeño** algunos sistemas sólo ofrecen paso de mensajes como único mecanismo de comunicación entre procesos.

# Paso de mensajes

- La comunicación entre procesos (“*Inter-Process Communication*” o IPC) se usa entre hebras dentro de...
  - un sistema distribuido.
  - un mismo ordenador.
  - un mismo espacio de direcciones.
- El motivo es poder realizar...
  - Intercambio de información.
  - Otra forma de sincronización y paso de señales.
- Requiere de al menos dos primitivas:
  - `enviar(receptor, mensaje)`
  - `recibir(emisor, mensaje)`
- ¿Cómo implementarlo?
  - ¿Llamadas al sistema? ¿Sockets?...
  - Pista: la hebras en comunicación pueden bloquearse.

# Principios básicos del paso de mensajes




- Las hebras de **un proceso** cooperan mediante **variables globales**, ej: búfer del problema productor/consumidor.
- Las hebras de **diferentes procesos** interaccionan mediante **paso de mensajes** si no pueden o no quieren establecer regiones de **memoria compartida**.

# Problemas del paso de mensajes

La **inconsistencia** de datos puede ser un problema porque...

- Los mensajes pueden llegar **desordenados** $\Longrightarrow$ incluso aunque cada mensaje sea correcto, su concatenación es incorrecta.
- Los mensajes pueden estar **incompletos** porque el receptor no tenga suficiente espacio libre en el búfer de recepción.
- Los mensajes pueden **perderse**.
- Los mensajes pueden estar **desfasados**, ej: transcurrido cierto tiempo un mensaje puede dejar de reflejar el estado del emisor.
- Cada mensaje podría suponer un problema de **seguridad**: goteo de información, desbordamiento, ataques DoS $\Longrightarrow$ deberíamos poder controlar si los mensajes deben ser enviados o no.

# Parámetros de diseño

- conexión de las partes en comunicación
- sincronización
- direccionamiento

- relación hebra/mensaje
- propiedad del mensaje

- Organización de la transferencia de datos:
  - ordenación de mensajes
  - formato de mensajes (tamaño)
  - almacenamiento temporal
  - planificación interna

# Conexión: orientados a conexión

`abrir_conexión(dirección)` Comprueba si el receptor existe y si quiere crear una conexión con el emisor.

`enviar(mensaje)` Envía mensaje.

`recibir(mensaje)` Recibe mensaje.

`cerrar_conexión()` Vacía el almacén temporal de mensajes y elimina la conexión.

# Conexión: no orientados a conexión

`enviar(destino, mensaje)` envía `mensaje` a `destino`.

`recibir(origen, mensaje)` recibe `mensaje` desde `origen`.

Interacción típica de sistemas cliente/servidor:

- La dirección destino suele ser un servidor.
- La dirección origen suele ser un cliente.

# Sincronización en el emisor

envío no sincronizado (no bloqueante): si no hay receptor esperando un mensaje, olvidar el mensaje y continuar.

envío asíncrono (no bloqueante): si no hay receptor esperando un mensaje, depositar el mensaje en un almacén temporal si queda suficiente espacio libre y continuar.

envío síncrono (bloqueante): si no hay receptor esperando un mensaje, depositar el mensaje y esperar a que aparezca uno.

en cualquier caso: si hay un receptor esperando un mensaje, transferir el mensaje y continuar.

# Sincronización en el receptor

recepción (no bloqueante): recupera un mensaje vacío si no hay mensaje... o comprobar antes si ha llegado alguno.

recepción (bloqueante): espera si no hay ningún mensaje hasta que uno esté disponible.

- En ambos casos, si hay un mensaje almacenado, transferir el mensaje al espacio de direcciones del receptor y continuar.

# Combinaciones emisor/receptor

| | recepción no bloqueante | recepción bloqueante |
|---|---|---|
| envío no sincronizado | × | sondeo emisor |
| envío asíncrono | sondeo receptor | comunicación asíncrona |
| envío síncrono | sondeo receptor | cita |

Observación: el **envío asíncrono** requiere **almacenamiento temporal** para los mensajes $\Longrightarrow$ posible vulnerabilidad.

# Mejorando el paso de mensajes

- Supongamos que un emisor envía un mensaje de forma síncrona.
- ¿Qué podría pasarle al emisor si el receptor no está presente?
  - El emisor **espera indefinidamente**.
  - Ejemplo de inanición distinto del interbloqueo.

- ¿Cómo podríamos arreglar este problema?
  - Añadiendo un mecanismo de **período de espera máximo**.

# Período de espera máximo (“*timeout*”)

- Con un período de espera máximo podemos especificar cuanto tiempo estamos dispuestos a esperar hasta que la comunicación tenga lugar.
- Deberíamos ser capaces de intercambiar mensajes dentro de un cierto intervalo de tiempo, incluso en sistemas con carga elevada.
- Podríamos mejorar el envío síncrono de esta forma:

  envío_síncrono(receptor, mensaje, espera, resultado)

- Si el receptor no recibe un mensaje en `espera` tiempo, el emisor puede ser informado mediante la variable `resultado` de forma que pueda decidir como actuar.

# Tipos de direccionamiento

Direccionamiento **directo**:

- `enviar(hebra, mensaje)`
- `recibir(hebra, mensaje)`
- `enviar(filtro(hebra), mensaje)`
- `recibir(filtro(hebra), mensaje)`

Direccionamiento **indirecto**:

- `enviar(buzón, mensaje)`
- `recibir(buzón, mensaje)`
- `enviar(puerto, mensaje)`
- `recibir(puerto, mensaje)`

# Direccionamiento directo (1)

Identificación mediante máscaras y/o comodines:

- `enviar(hebra1, mensaje)`
- `recibir(hebra2, mensaje)`
- `enviar(hebra1 ∧ hebra2, mensaje)`
- `recibir(hebra1 ∨ hebra2, mensaje)`
- `enviar(*, mensaje)`
- `recibir(*, mensaje)`

# Direccionamiento directo (2)

Propiedades de un enlace de comunicaciones temporal:

- Los enlaces se establecen automáticamente.
- El enlace se asocia exactamente a un par de hebras.
- El enlace puede ser unidireccional o bidireccional en función del patrón de comunicaciones:
  - **Notificación**.
  - **Petición**.

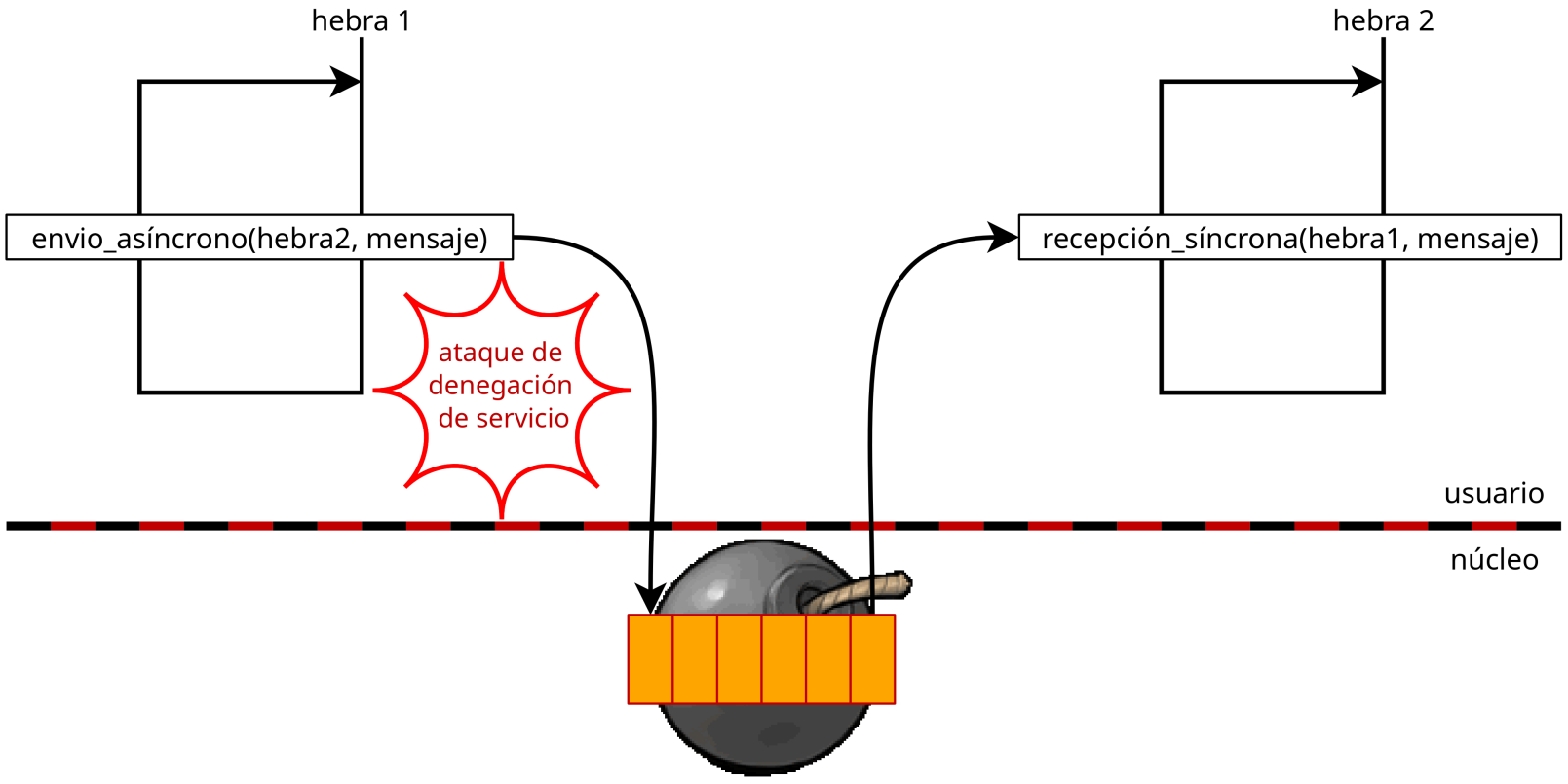
# Notificación

- Una notificación es un mensaje de **un sentido** de un emisor a un receptor.
- La transacción que representa el mensaje **finaliza** con la **entrega** del mensaje al receptor.
- La **interpretación** del mensaje en el receptor **no forma parte** de la comunicación entre procesos.
- Equivalente a las señales en sincronización.

# Petición

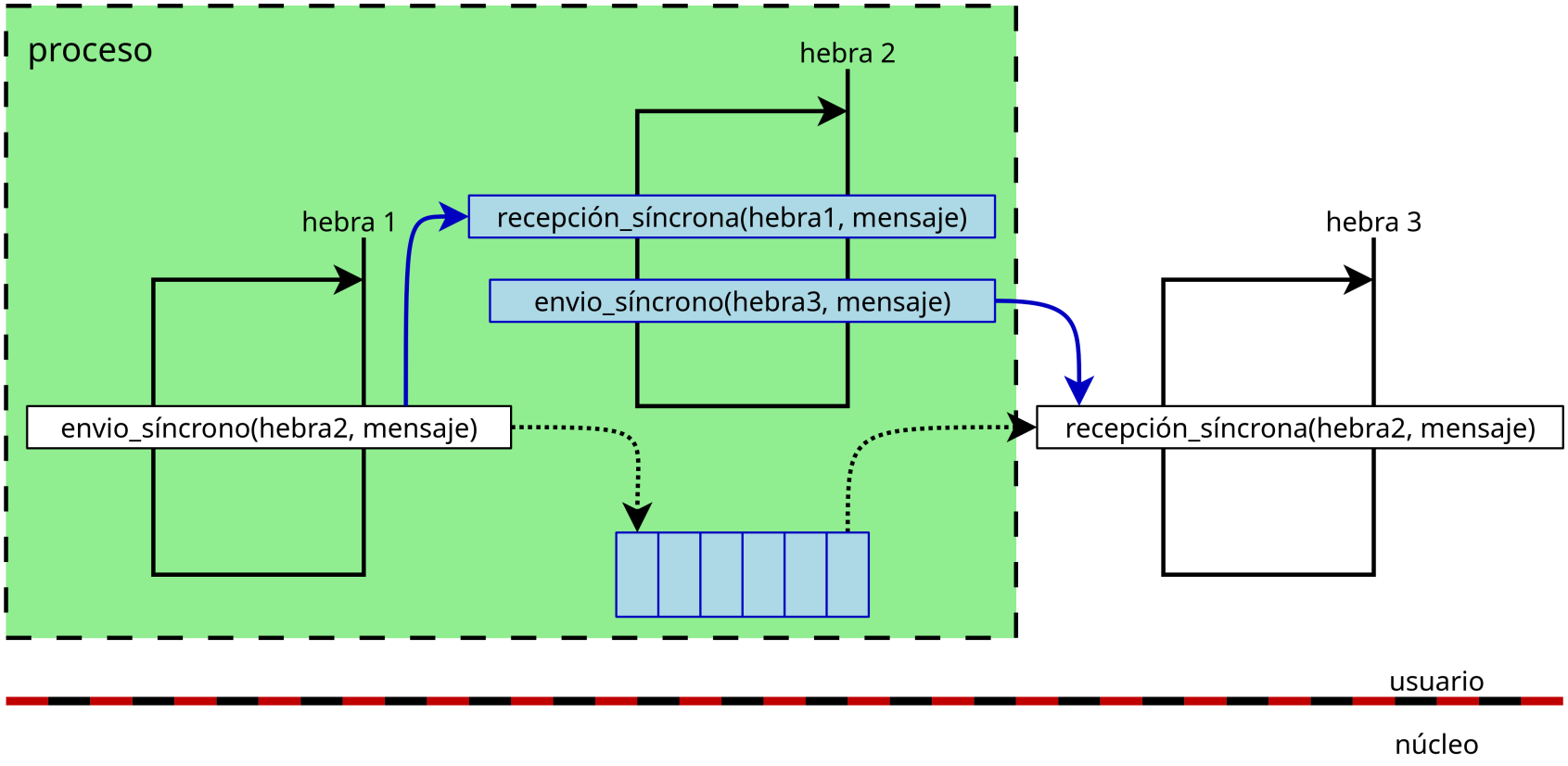
- Una petición es un mensaje de **doble sentido** iniciada desde un emisor y respondida por el receptor.
- Se inicia mediante el envío de la **petición** al receptor y finaliza con el envío de un mensaje de **respuesta**, y resultado de la petición, desde el receptor al emisor.
- Entre ambos mensajes el receptor (servidor) ha proporcionado el servicio requerido.

# El problema del envío asíncrono




- Ataque de denegación de servicio (*"Denial of Service" - DoS*) provocado por el desbordamiento de un búfer en el núcleo.
- DoS: un servicio o recurso deja de ser accesible a usuarios legítimos.

# Simulando envío asíncrono mediante envío síncrono




- Almacén temporal en espacio de usuario + cambiar envío asíncrono por síncrono $\Rightarrow$ elimina el problema de DoS.

# Direccionamiento indirecto (1)

Los mensajes se envían a, o se reciben desde, buzones, canales y puertos.

- Cada buzón tiene un **identificador único**.
- Las hebras sólo se pueden comunicar si **comparten** un buzón.

Propiedades del enlace de comunicación:

- El enlace sólo se establece si las hebras comparten un buzón.
- Un enlace puede asociarse con múltiples hebras.
- Cada hebra puede compartir varios enlaces de comunicaciones.
- Los enlaces pueden ser unidireccionales o bidireccionales.

# Direccionamiento indirecto (2)

Operaciones:

- crear nuevo buzón.
- enviar y recibir mensajes a través del buzón.
- eliminar buzón.

Primitivas para utilizar buzones:

- `enviar(buzón, mensaje)`: envía mensaje a buzón.
- `recibir(buzón, mensaje)` recibe mensaje de buzón.
- `conectar(buzón, hebra)`: conecta hebra a buzón.
- `desconectar(buzón, hebra)` desconecta hebra de buzón.

# Direccionamiento indirecto (3)

Compartición de buzones:

- Tres hebras $H_1$, $H_2$ y $H_3$ comparten un buzón.
- Supongamos que $H_2$ y $H_3$ han ejecutado `recibir()` sobre el buzón.
- Si ahora $H_1$ envía un mensaje al buzón,... ¿Qué hebra recibirá el mensaje, $H_2$ o $H_3$?

Posibles soluciones:

- Añadir un campo a las primitivas de envío y recepción que identifique al emisor y/o receptor.
- Permitir al sistema seleccionar un receptor al azar y notificar su identidad al emisor.

Los patrones de comunicación de alto nivel suelen estar basado en el uso de buzones.

# Resumen sobre direccionamiento indirecto

Ventajas:

- Igual de fácil de utilizar y más flexible que el direccionamiento directo.
- Adecuado cuando desconocemos la identidad de la otra parte.
- Cada buzón puede disponer de una política de seguridad personalizada.
- Los buzones pueden perdurar más tiempo que las hebras que los crearon.

Inconvenientes:

- Sobrecarga espacial debido a las estructuras de datos necesarias.
- Cada mensaje puede requerir ser copiado varias veces.
- ¿Qué hacer con las hebras conectadas a un buzón cuando su propietario lo elimina?
- Si una hebra conectada a un buzón debe ser finalizada corremos el riesgo de dejar mensajes colgados y atascar el buzón.

# Relación hebra/mensaje

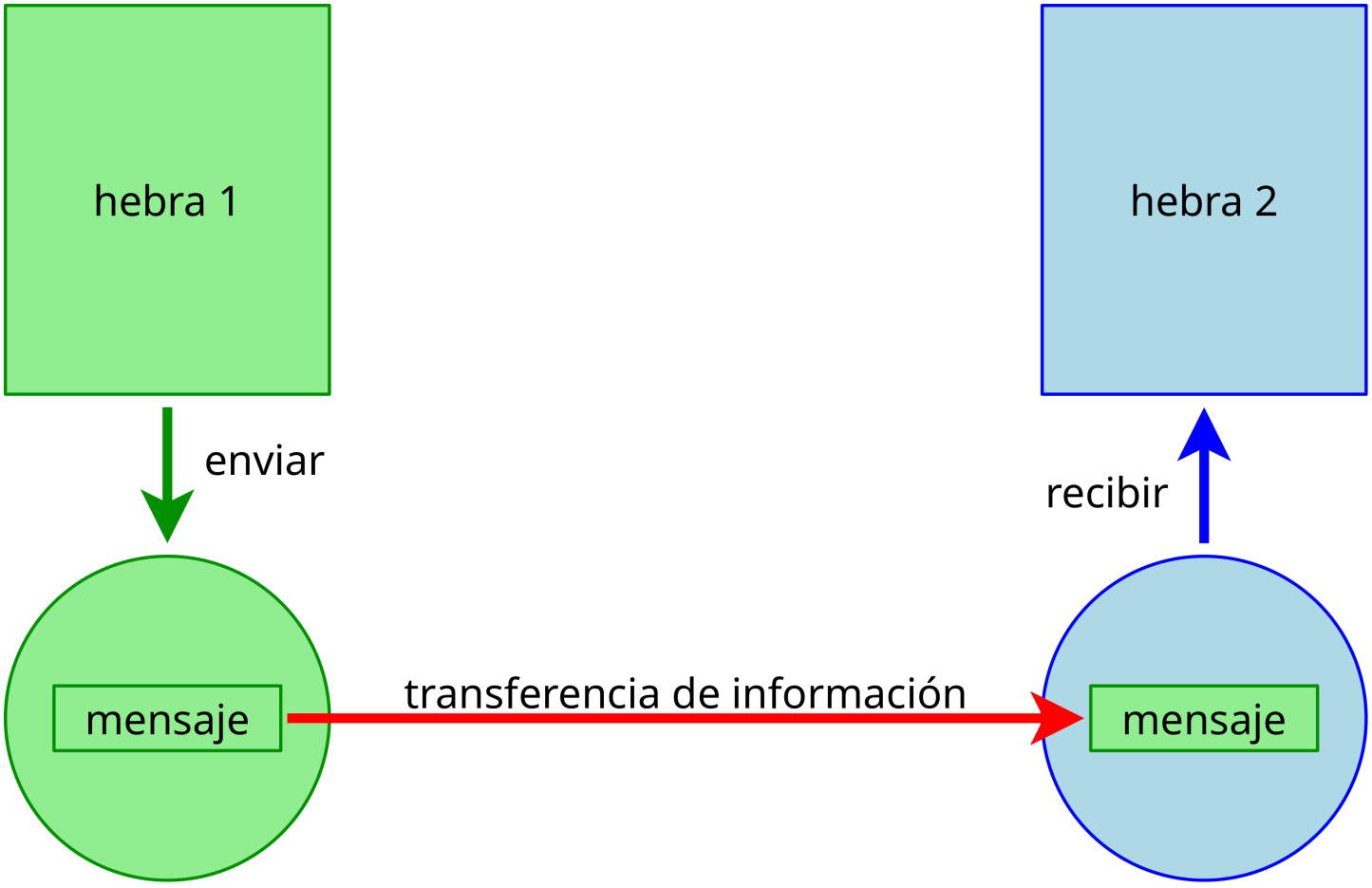
**Acoplamiento/atraque** (*"docking"*): relación de la hebra involucrada en el proceso de comunicación con el mecanismo de comunicaciones:

Estático: la hebra posee un objeto IPC (ej: puerto) sólo durante su tiempo de vida.

Dinámico: las hebras pueden...

- crear un nuevo buzón.
- conectarse y desconectarse de un buzón.
- eliminar un buzón.

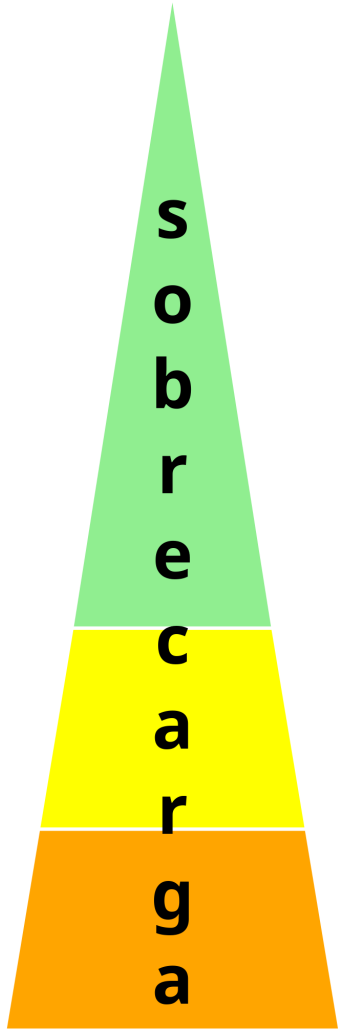
# Transferencia de datos (1)




Pregunta: ¿Tendremos que copiar el mensaje en cada transferencia?

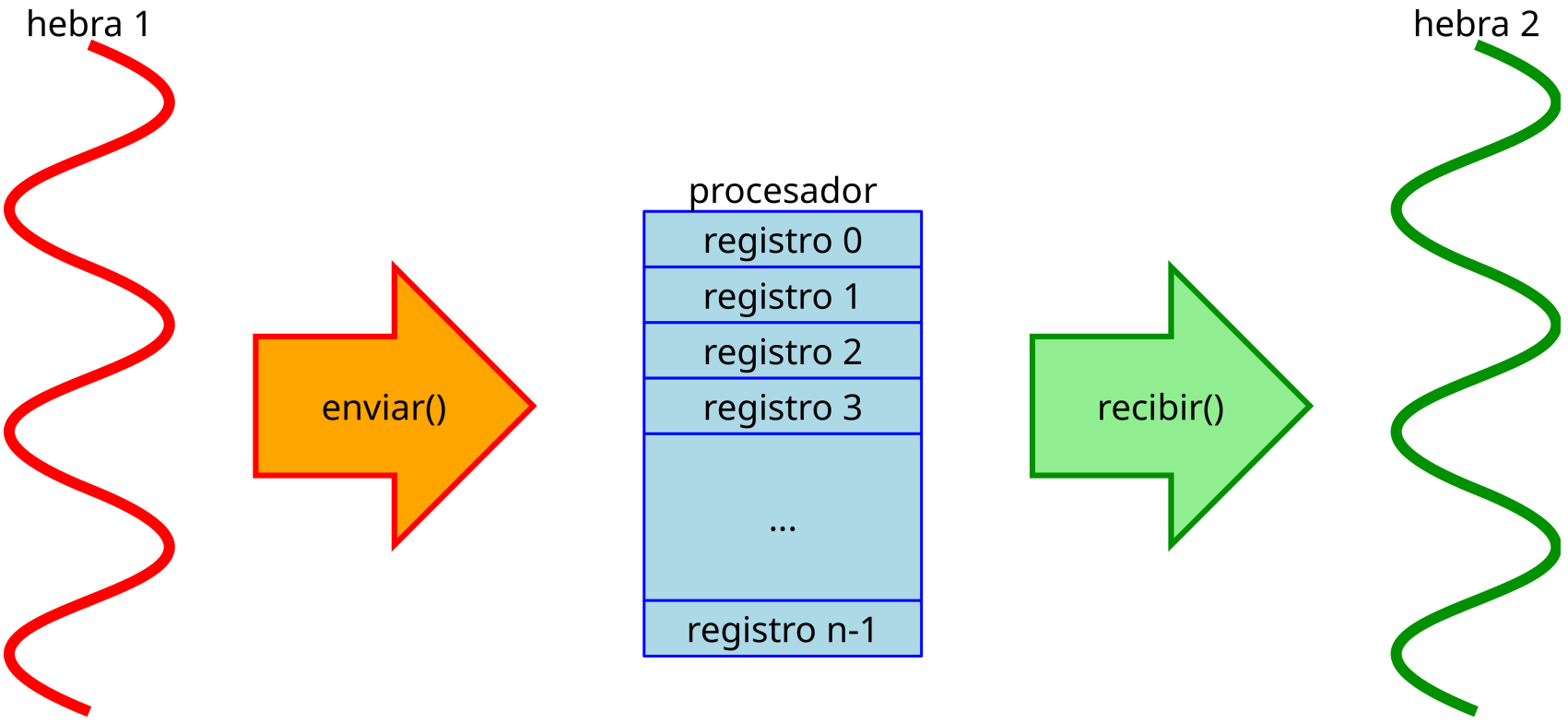
# Transferencia de datos (2)

- Registros:
  - mensajes **cortos**.
  - 0 copias.
- Memoria compartida:
  - mensajes de longitud arbitraria.
  - 0 copias.
  - registros/memoria del núcleo empleada para transmitir la **dirección** del mensaje.
- Mapeo temporal:
  - mensajes de longitud arbitraria.
  - 1 copia.
- Búfer en el núcleo:
  - mensajes de longitud arbitraria.
  - 2 copias.



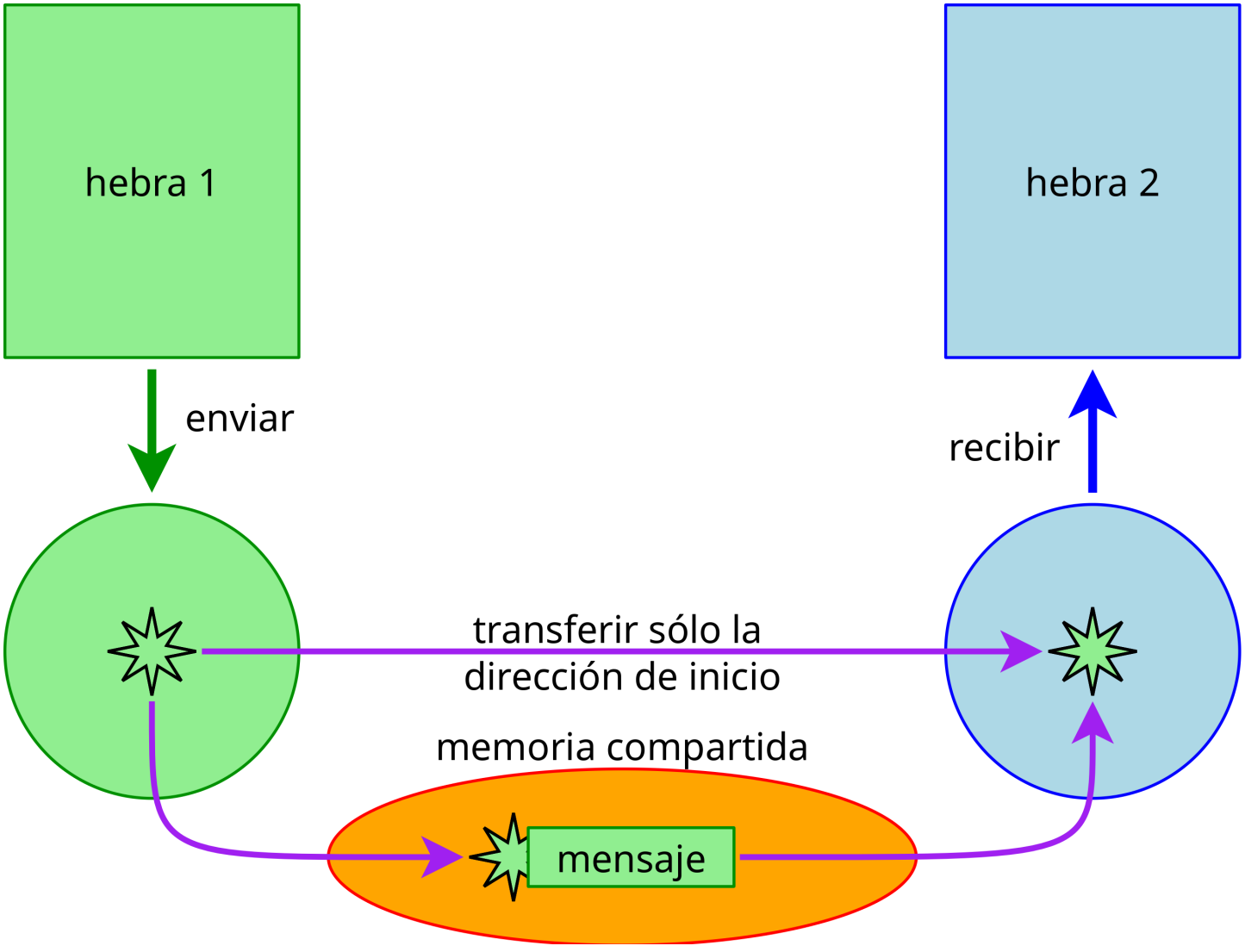

¿Qué implicaciones sobre sincronización tiene cada método?

# Transferencia de datos a través de registros

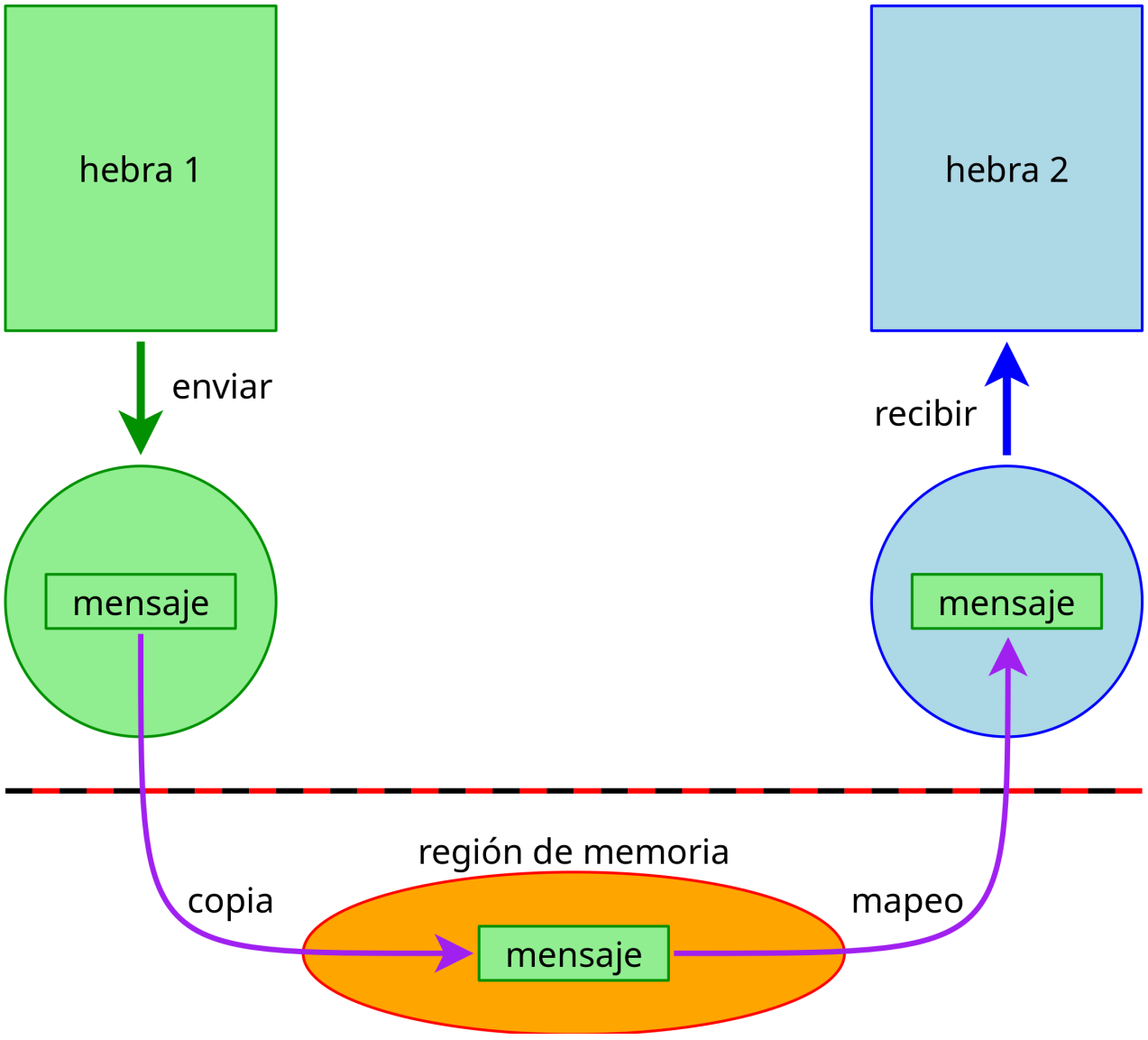



- ¿Funciona en cualquier sistema? $\Longrightarrow$ No, sólo UMA.
- ¿Ventajas? $\Longrightarrow$ velocidad.
- ¿Inconvenientes? $\Longrightarrow$ capacidad y restricciones de sincronización.

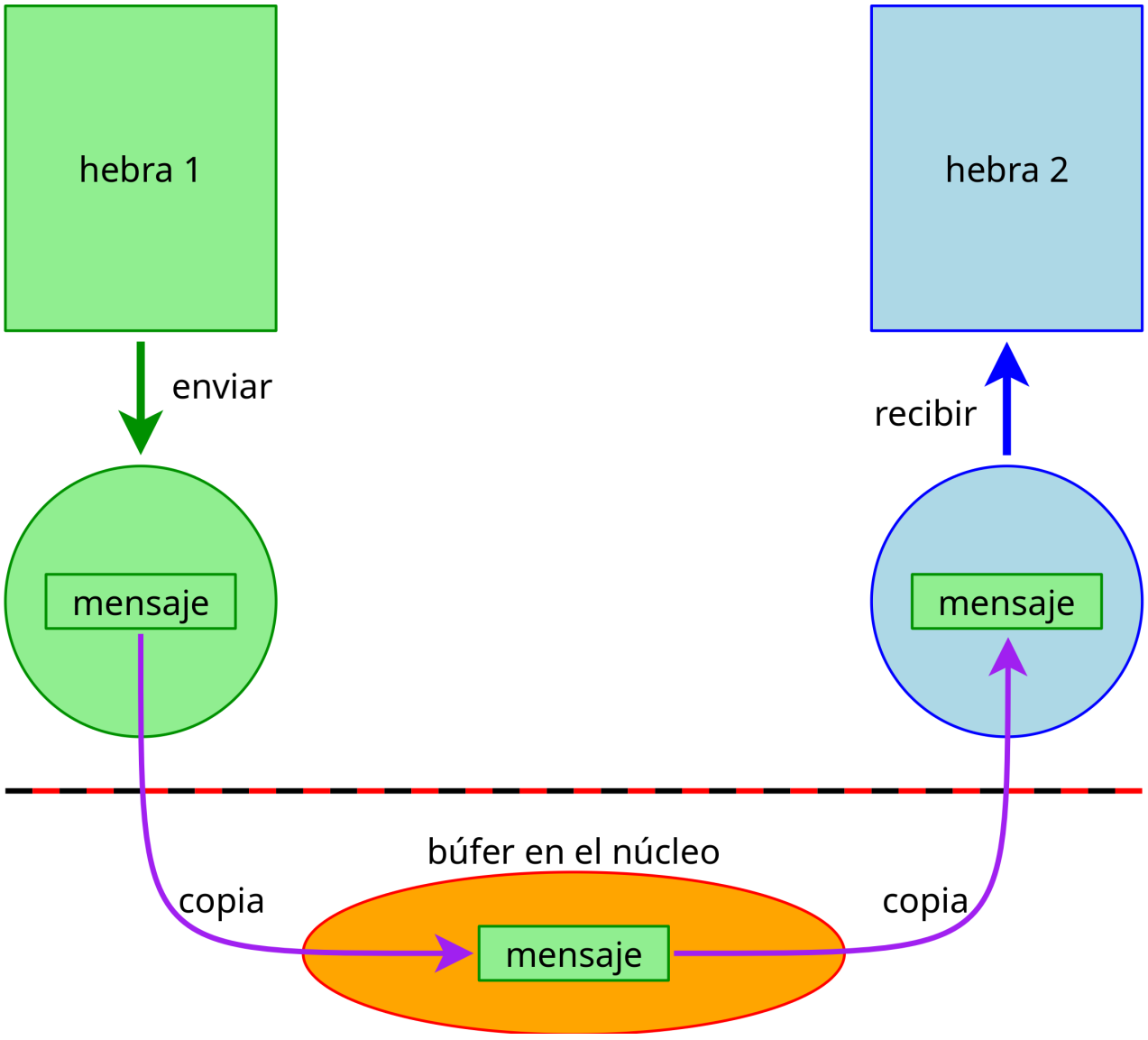
# Transferencia de datos a través de memoria compartida



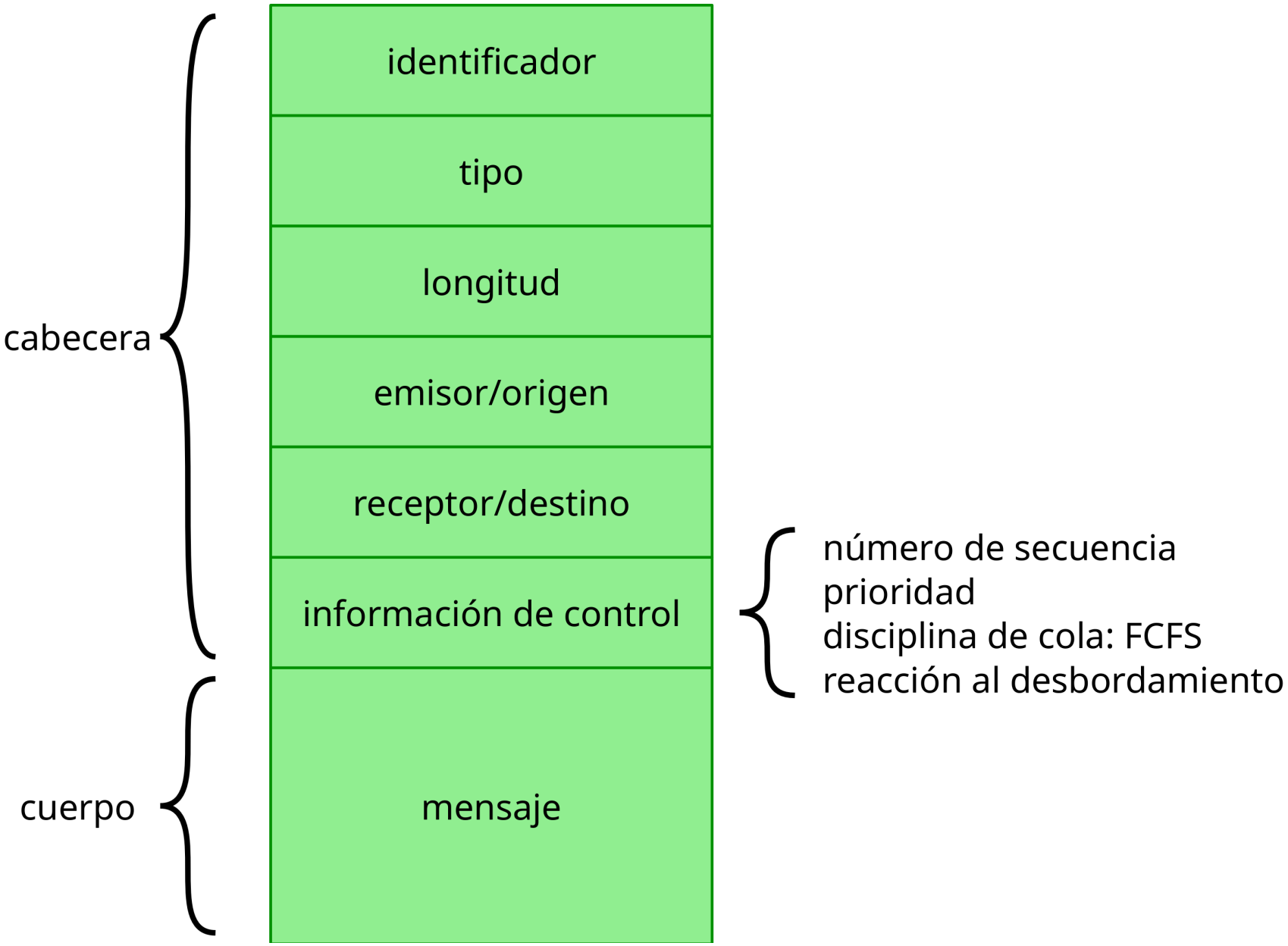
# Transferencia de datos a través de mapeo de memoria



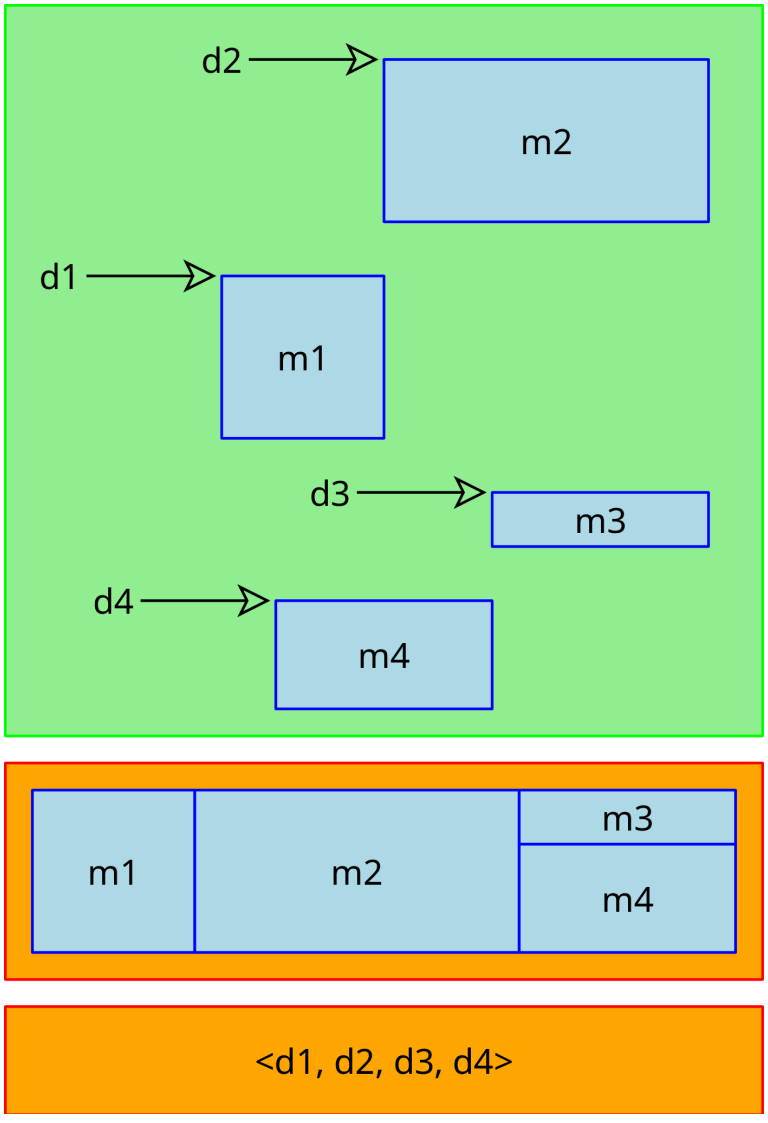
# Transferencia de datos a través de un búfer en el núcleo



# Formato de los mensajes



# Mensajes no contiguos




- Problema: enviar un mensaje **disperso** en memoria.
- Soluciones:
  - copiar las partes del mensaje m1, m2, m3 y m4 en un búfer único y enviarlo.
  - copiar las direcciones iniciales de los componentes del mensajes d1, d2, d3 y d4 a un búfer y enviar dicho búfer $\longrightarrow$ requiere memoria compartida.

# Comunicación cliente/servidor

- Sistemas locales:
  - Servidores locales.
  - Protocolo D-Bus[1]: implementado en espacio de usuario (libdbus, GDBus y sd-bus) y núcleo (kdbus y BUS1).

- Sistemas distribuidos:
  - *Sockets.*
  - Llamada a procedimiento remoto
    RPC - *"Remote Procedure Call"*
  - Invocación de método remoto
    RMI - *"Remote Method Invocation"*

[1]Desktop Bus: sistema IPC/RPC entre aplicaciones.

# IPC con un servidor

| Cliente | |
|---|---|
| . | enviar(servidor, petición) |
| solicitar servicio de un servidor: | |
| . | recibir(servidor, resultado) |

Ventajas:

- No necesitamos mecanismos adicionales.

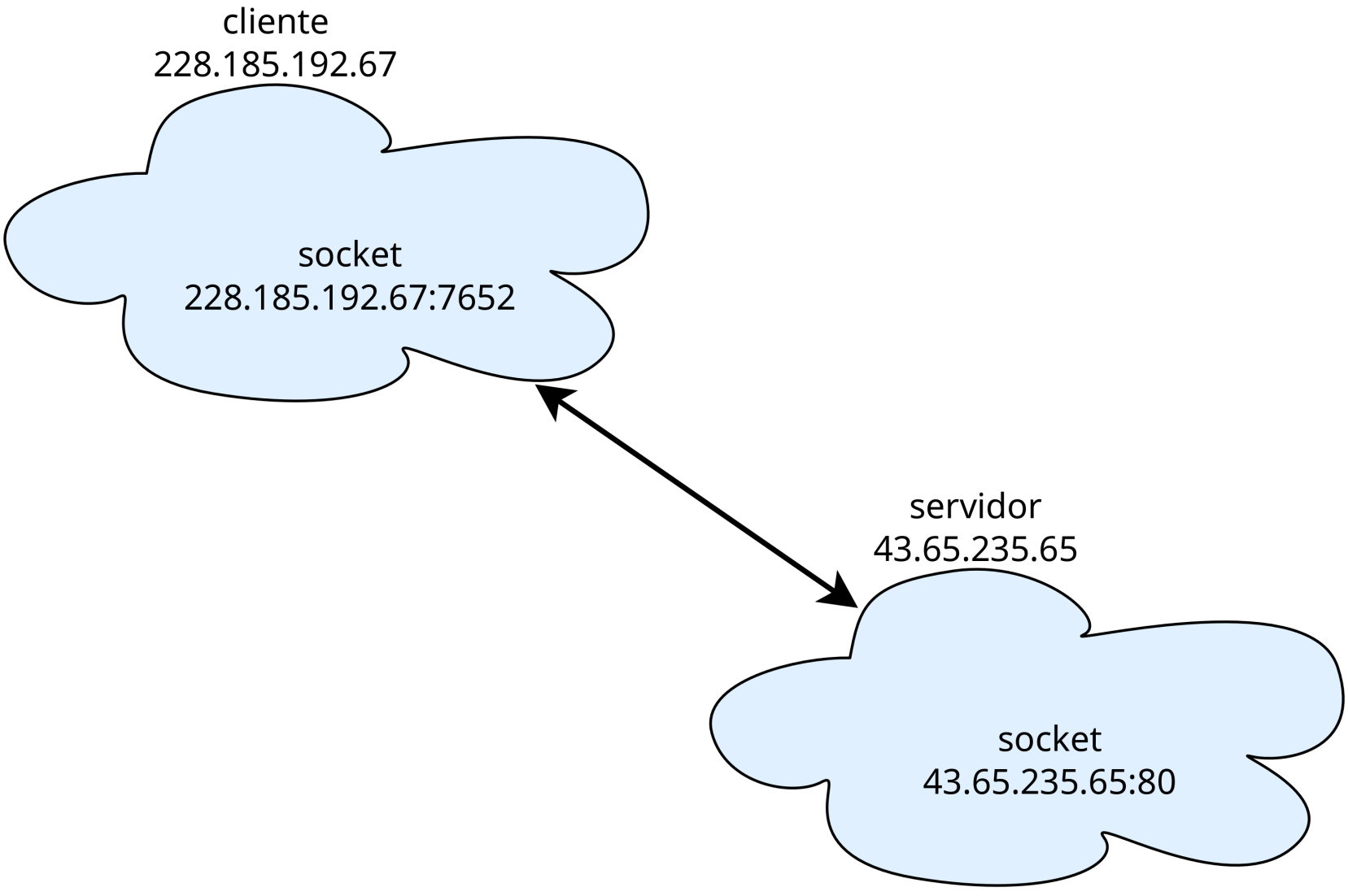
Inconvenientes:

- Requiere 2 llamadas al sistema $\longrightarrow$ **sobrecarga**.
- Si el activador se ejecuta entre las dos llamadas el servidor no completa su operación y retrasa al cliente.

# *Sockets*

- Abstracción para designar un **punto de conexión** del enlace de comunicaciones.
- Unión de una dirección **IP** y un número de **puerto**, ej: 1.2.3.4:5.
- El socket 1.2.3.4:5 hace referencia al puerto 5 en la máquina con dirección IP 1.2.3.4.
- La comunicación tiene lugar entre un **par** de sockets.

# Comunicación entre sockets



# Llamada a procedimiento remoto (RPC)

Cliente

.

solicitar servicio: RPC(servidor, petición)

.

Ventajas:

- Sólo es necesaria una llamada al sistema.

Inconvenientes:

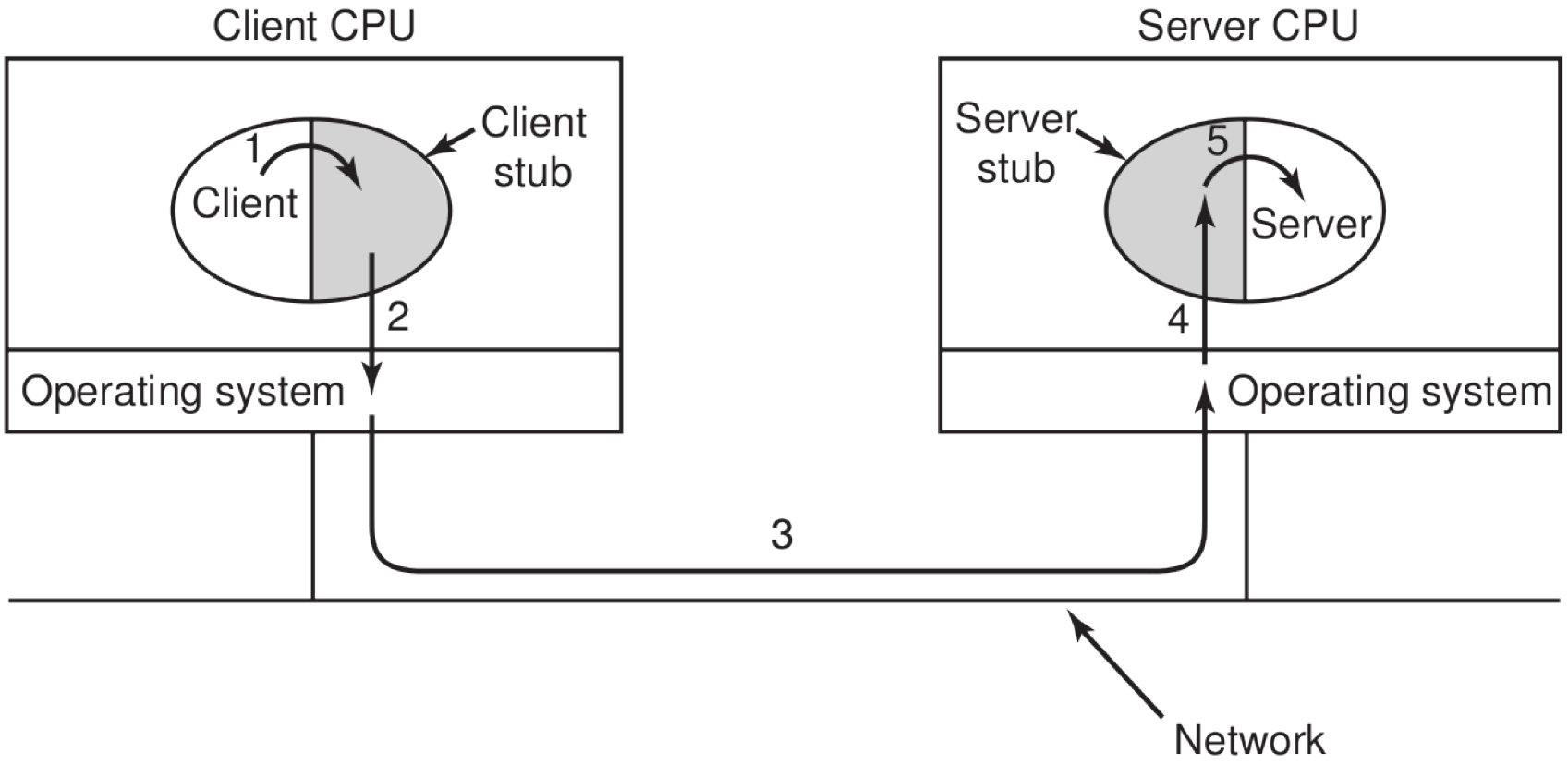
- El proceso invocador debe esperar el resultado.
- Aunque imprescindible en sistemas distribuidos es un mecanismo adicional que debe ser implementado.

Dentro de un mismo sistema se denomina llamada a procedimiento local (LPC - *"Local Procedure Call"*).

# Llamada a procedimiento remoto (RPC)

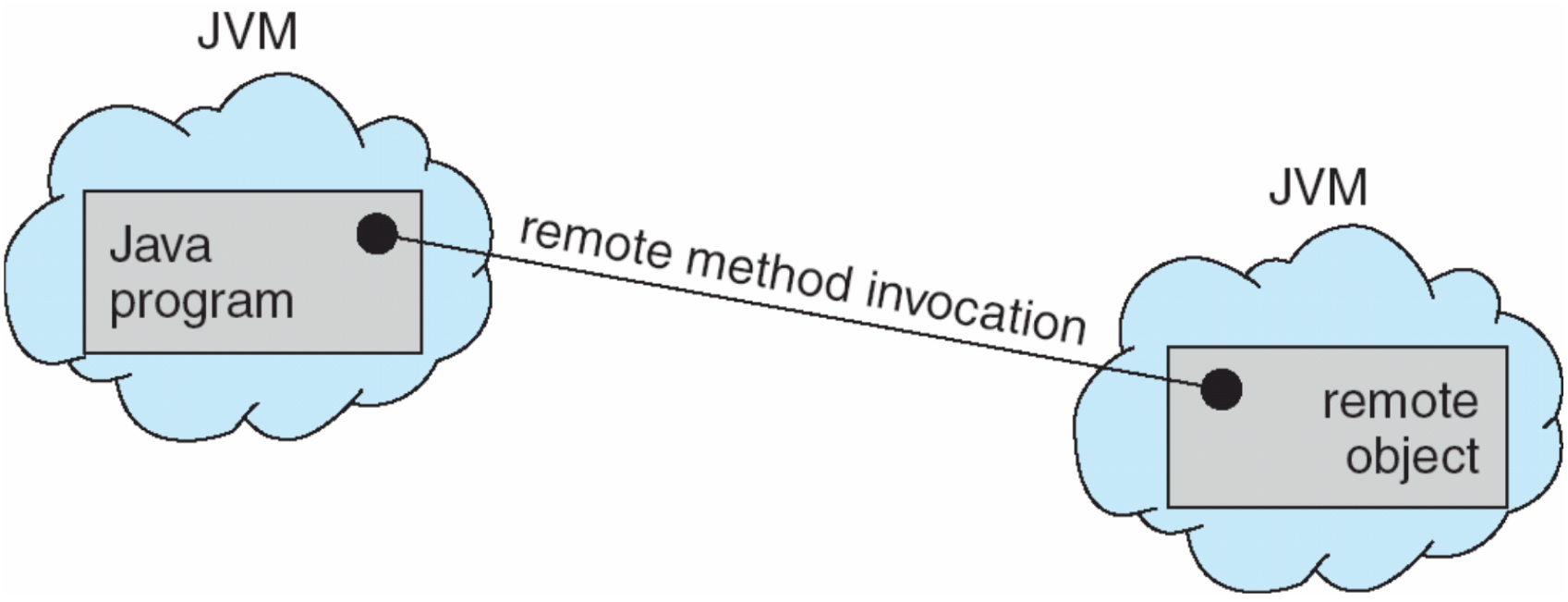
- RPC es una abstracción de llamada a procedimiento entre procesos en sistemas en red.
- Una RPC se utilizan como procedimiento local gracias a los *"stubs"* que ocultan los detalles de la comunicación:
  - cuando el usuario solicita una RPC el sistema localiza el stub adecuado y le pasa los parámetros.
  - el stub empaqueta los parámetros (*"marshalling"*) y realiza el paso de mensaje hacia el servidor.
  - el servidor dispone de otro stub que hace el trabajo de forma inversa: desempaqueta parámetros (*"unmarshalling"*) y llama al procedimiento local.
- Problemas:
  - tipo y codificación de la información (ej: BE/LE).
  - problemas introducidos por la red: eliminación, duplicación,...

# Stubs

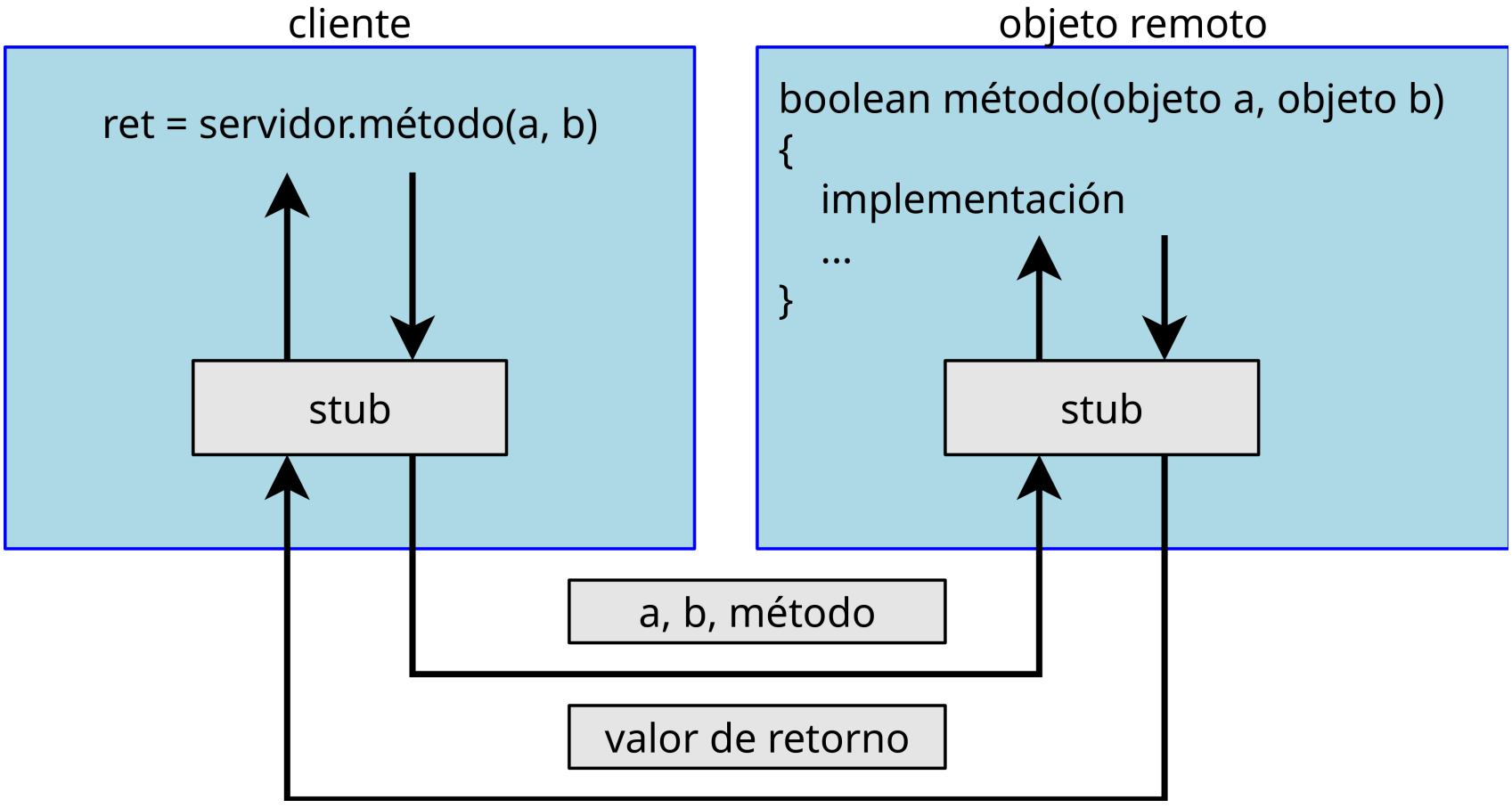

Client CPU
Server CPU
Client
stub
1
Client
2
Server
stub
5
Server
4
Operating system
Operating system
3
Network

# Invocación de método remoto (RMI)

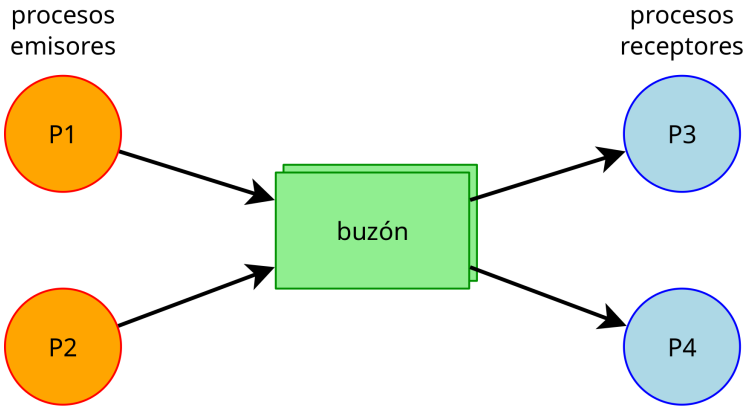
- RMI es un mecanismo de Java similar a las RPC.
- RMI permite a un programa Java invocar la ejecución de un método de un objeto sobre otra máquina remota.

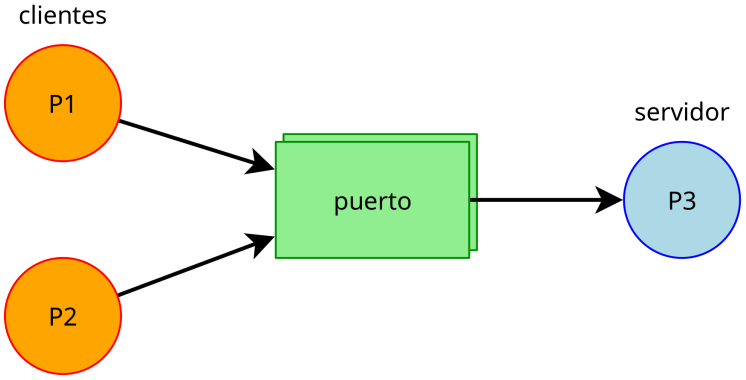


# Empaquetado de parámetros (*"marshalling"*)



# Buzones/canales frente a puertos




- Un buzón puede ser privado a un par de emisores/receptores.
- El mismo **buzón** puede ser compartido por **múltiples emisores y receptores**.
- Algunos SO permiten distinguir los mensajes por su tipo (Unix).
- **Un puerto** en cambio es un buzón asociado a un **único receptor** y a cualquier número de emisores.
- Los puertos suele emplearse en aplicaciones cliente/servidor en las que el único receptor es el servidor.

# Exclusión mutua mediante RPC

Truco: sólo una hebra ejecuta la sección crítica.

**Hebra cliente$_i$**

```
while (true)
{
  RPC(hebra_servidora);
  sección_no_crítica;
}
```

**Hebra servidora**

```
while (true)
{
  recibir(cualquiera);
  sección_crítica;
  enviar(cliente, "hecho");
}
```

¿Cuáles son las ventajas e inconvenientes de este método?

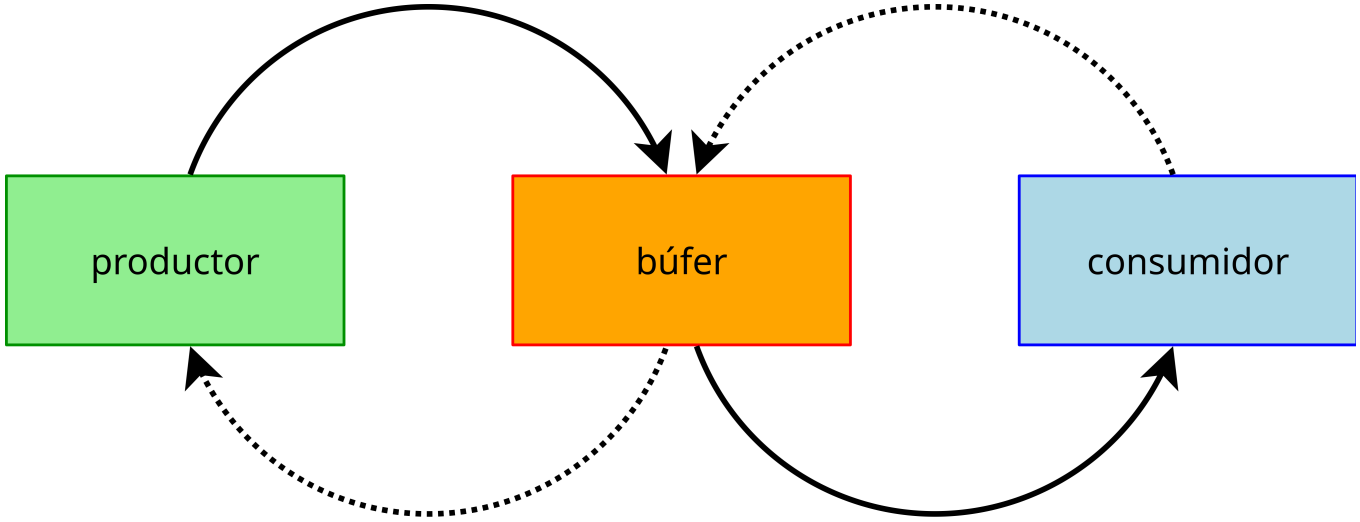
# Productor/consumidor mediante RPC (1)

## productor

```
while (true)
{
     mensaje_t mensaje;
     producir(mensaje);
     RPC(búfer, mensaje);
}
```

## consumidor

```
while (true)
{
     mensaje_t mensaje;
     RPC(búfer, mensaje);
     consumir(mensaje);
}
```



# Productor/consumidor mediante RPC (2)

```
class búfer_rpc
{
public:
  void trabajo()
  {
    estado = esperando_trabajo;
    while (true)
    {
      recibir(cliente, mensaje);
      if (cliente == productor)
        meter(mensaje);
      else // cliente == consumidor
        sacar(mensaje);
    }
  }

  void meter(mensaje)
  {
    if (búfer.lleno())
      estado = productor_en_espera;
    else
      enviar(productor, "ok");

    if (estado == consumidor_en_espera)
    {
      enviar(consumidor, mensaje);
      estado = esperando_trabajo;
    }
    else
      búfer.almacenar(mensaje);
  }

  void sacar(mensaje)
  {
    if (búfer.no_vacio())
      enviar(consumidor, mensaje);

    if (estado == productor_en_espera)
    {
      enviar(productor, "ok");
      estado = esperando_trabajo;
    }
    else
      estado = consumidor_en_espera;
  }

private:
  ...
};
```

# Exclusión mutua mediante IPC

**variables compartidas**

```
buzón mutex;
enviar(mutex, "turno");
```

**$Hebra_i$**

```
mensaje_t mensaje;
while (true)
{
    recibir(mutex, mensaje);
    sección crítica;
    enviar(mutex, mensaje);
    sección no crítica;
}
```

- Crear un buzón `mutex` compartido por todas las hebras.
- Es necesaria la inicialización: `enviar(mutex, "turno")`.
- `recibir()` es bloqueante cuando el buzón `mutex` está vació.
- `enviar()` es no bloqueante.
- La primera hebra que ejecute `recibir()` podrá ejecutar su sección crítica.
- Las demás hebras permanecerán bloqueadas hasta que la que está ejecutando su sección crítica la abandone y devuelva el mensaje al buzón `mutex`.

# Mecanismos de IPC en Unix

Comunicación entre hebras y/o procesos:

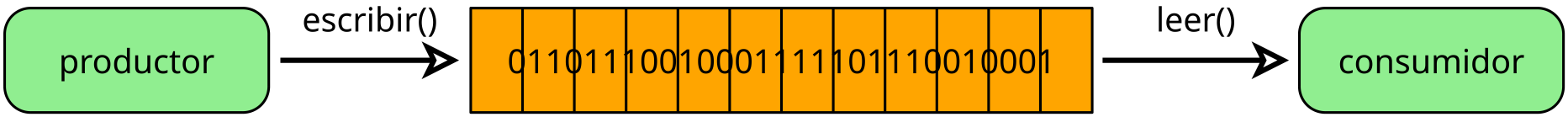
- Tuberías:
  - anónimas.
  - con nombre.
- Mensajes.
- Memoria compartida.

Para disparar acciones en otras hebras y/o procesos:

- Señales.
- Semáforos.

# Tuberías

- Dos procesos intercambian chorros de bytes en orden FCFS.
- Implementación: búfer circular de 4 a 64KB.
- Las tuberías pueden utilizarse desde...
  - los programas mediante la API del núcleo (libc).
  - el interprete de órdenes mediante el operador "|".
- Sincronización implícita en caso de búfer lleno/vacío:
  - el productor es detenido al llenar el búfer.
  - el consumidor es detenido al vaciar el búfer.
- Escribir sobre una tubería sin ningún consumidor lanza una excepción.



# Tuberías anónimas

- Sólo pueden ser utilizadas por dos procesos de la misma "familia", por ejemplo, un proceso padre e hijo.
- Habitualmente sólo se emplean de forma unidireccional.
- Se borran de forma automática tan pronto como desaparece el escritor si no existe ningún lector.

# Ejemplos de uso de tuberías en el intérprete de órdenes

```
cat alumnos.txt | sort
```

el contenido del fichero `alumnos.txt` es pasado a `sort` y mostrado por pantalla de forma ordenada.

```
cat practica.cc | sed "s/alarma(1)/alarma(8)/"> practica2.cc
```

cambia el número de segundos de alarma y guarda el fichero de código fuente `practica.cc` con el nuevo nombre `practica2.cc`.

# Ejemplo de uso de tuberías con la API del núcleo

```
int main()
{
    char bufer[10];    // bufer de recepción de datos
    int descriptor[2]; // descriptores de ficheros tubería

    pipe(descriptor);  // crear una nueva tuberia

    if (fork() == 0)                              // hijo
    {
        write(descriptor[1], "hola papi", 10); // enviar mensaje
    }
    else                                          // padre
    {
        read(descriptor[0], bufer, 10);           // recibir mensaje
        std::cout << "hijo --> padre: " << std::quoted(bufer) << "\n";
    }
}
```

# Tuberías con nombre

- Pueden ser utilizados por procesos no relacionados y de forma bidireccional (*"full duplex"*).
- A diferencia de las tuberías anónimas son objetos persistentes.
- A pesar de ser ficheros persistentes su contenido desaparece cuando lo hace el escritor si no hay ningún lector.
- Puede ser reutilizado en el futuro por cualquier proceso con privilegios para acceder al mismo.

# Ejemplo de tubería con nombre

bash

```
$ mkfifo mi_tuberia
$ xz < mi_tuberia > 1m.xz
$ rm mi_tuberia
```

bash

```
$ seq 1000000 > mi_tuberia
```

- Use dos terminales o lance `xz` en segundo plano.
- `mkfifo`: creador de tuberías.
- `xz`: compresor de ficheros.
- Las tuberías con nombre son empleadas con frecuencia para establecer relaciones cliente/servidor.

# API del núcleo sobre tuberías con nombre

`int mkfifo(const char *ruta, modo_t modo);`

- crea una tubería en `ruta`.
- `modo` especifica los permisos.

- Una vez creada una tubería cualquier proceso puede abrirla para lectura o escritura, de la misma manera que cualquier fichero normal.
- Sin embargo, tiene que ser abierto en los **dos extremos simultáneamente** antes de que se pueda proceder a cualquier operación de entrada o salida.
- Abrir un FIFO para **lectura** provoca un **bloqueo** hasta que algún otro proceso abre el mismo FIFO para **escritura**.
- Toda operación sobre una tubería es **atómica**.

# IPC en UNIX

Características:

- Creación dinámica.
- Persistencia.
- Utilizables por cualquier proceso.
- Identificables de forma no ambigua.

Semántica:

- Envío asíncrono.
- No se preserva el orden de llegada a la cola.
- Los mensajes se borran una vez repartidos.

# System V IPC de UNIX System V

`int msgget(key_t key, int msgflg)` La función devuelve el identificador de la cola de mensajes asociada a key. Se crea una nueva cola de mensajes si no existe ninguna cola de mensajes asociada a key. En otro caso se consulta msgflg para decidir que hacer.

`int msgsnd(int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg)` Envia el mensaje `msgp` a la cola `msqid`.

`size_t msgrcv(int msqid, struct msgbuf *msgp, size_t msgsz, long msgtyp, int msgflg)` Recibe en `msgp` un mensaje desde la cola `msqid`.

`int msgctl(int msqid, int cmd, struct msqid_ds *buf)` Ejecuta operación especificada por `cmd` en la cola de `msqid`.

`mqd_t mq_open(const char *name, int oflag)` crea una nueva cola de mensajes.

`int mq_close(mqd_t mqdes)` cierra el descriptor de cola de mensajes `mqdes`.

`int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned int msg_prio)` añade el mensaje apuntado por `msg_ptr` a la cola de mensajes `mqdes`.

`ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned int *msg_prio)` elimina el mensaje más viejo y de mayor prioridad de la cola de mensajes `mqdes` y lo coloca en `msg_prt`.

# Implementaciones de IPC

**Binder** Mecanismo de comunicación síncrona entre procesos utilizado por Android.

**D-Bus** Mecanismo de comunicación asíncrona entre procesos desarrollado como parte del proyecto `freedesktop.org`. Emplea sockets como mecanismo de transporte de los mensajes.