# Sistemas Distribuidos

## Comunicación

## Programación con sockets

# Introducción

- Prácticas individuales: C con sockets
  - 1ª práctica: sockets *stream* (TCP)
  - 2ª práctica: sockets datagramas (UDP)
- En SS.OO. ya estudiamos este mecanismo de comunicación
  - Asumimos que ya se conocen las operaciones de los sockets
  - Pero se necesita profundizar en esta tecnología
- En este tema:
  - Revisaremos aspectos más avanzados
  - Daremos pautas sobre la programación con sockets
  - Identificaremos errores típicos
- Nos basaremos en ejecutar “experimentos”

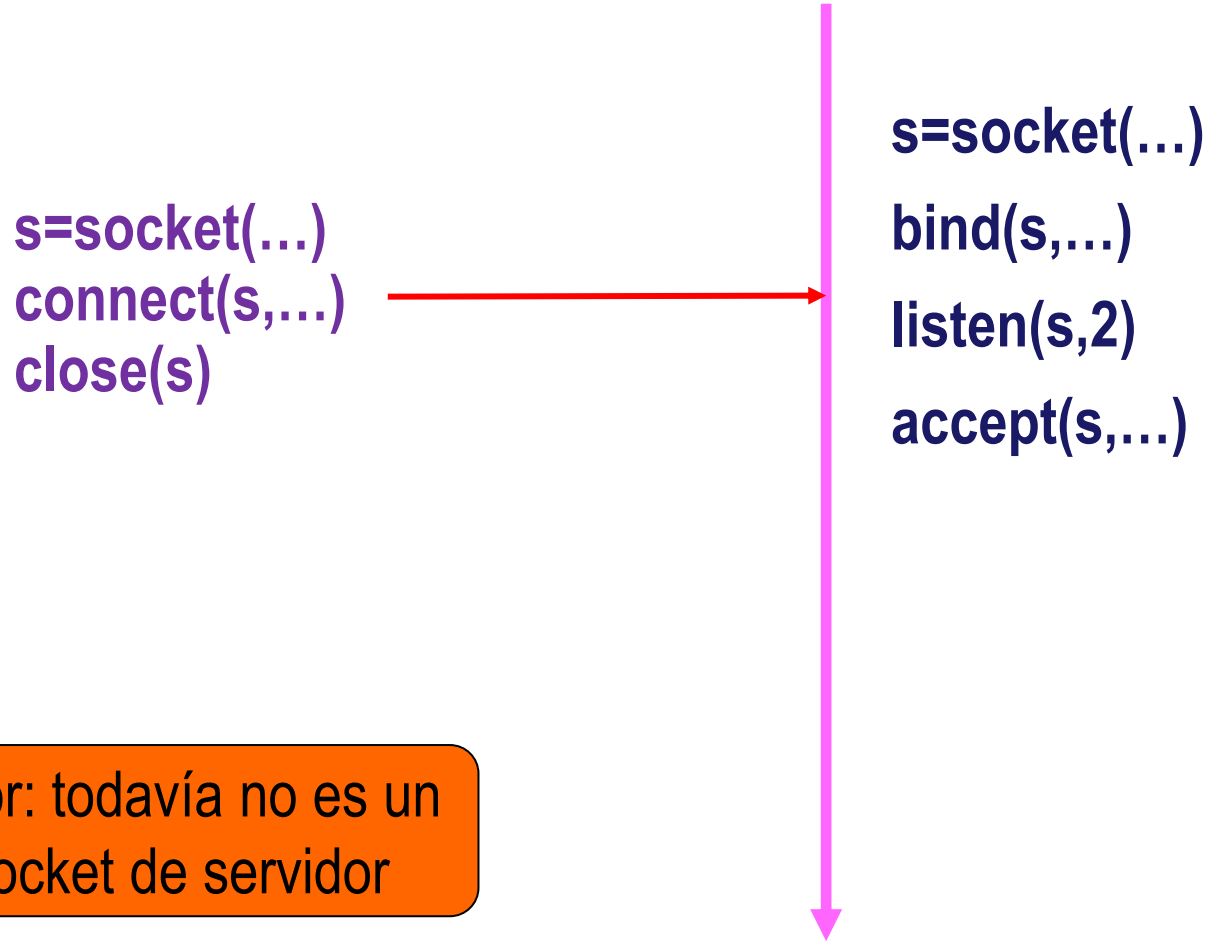
# Experimentando con las conexiones

- Ya conocemos la dinámica de las conexiones
  - *connect* del cliente y *accept* de servidor establecen la conexión
- Pero vamos a analizar más en detalle esta interacción
- Planteando distintas temporizaciones
- Experimentaremos con 2 programas

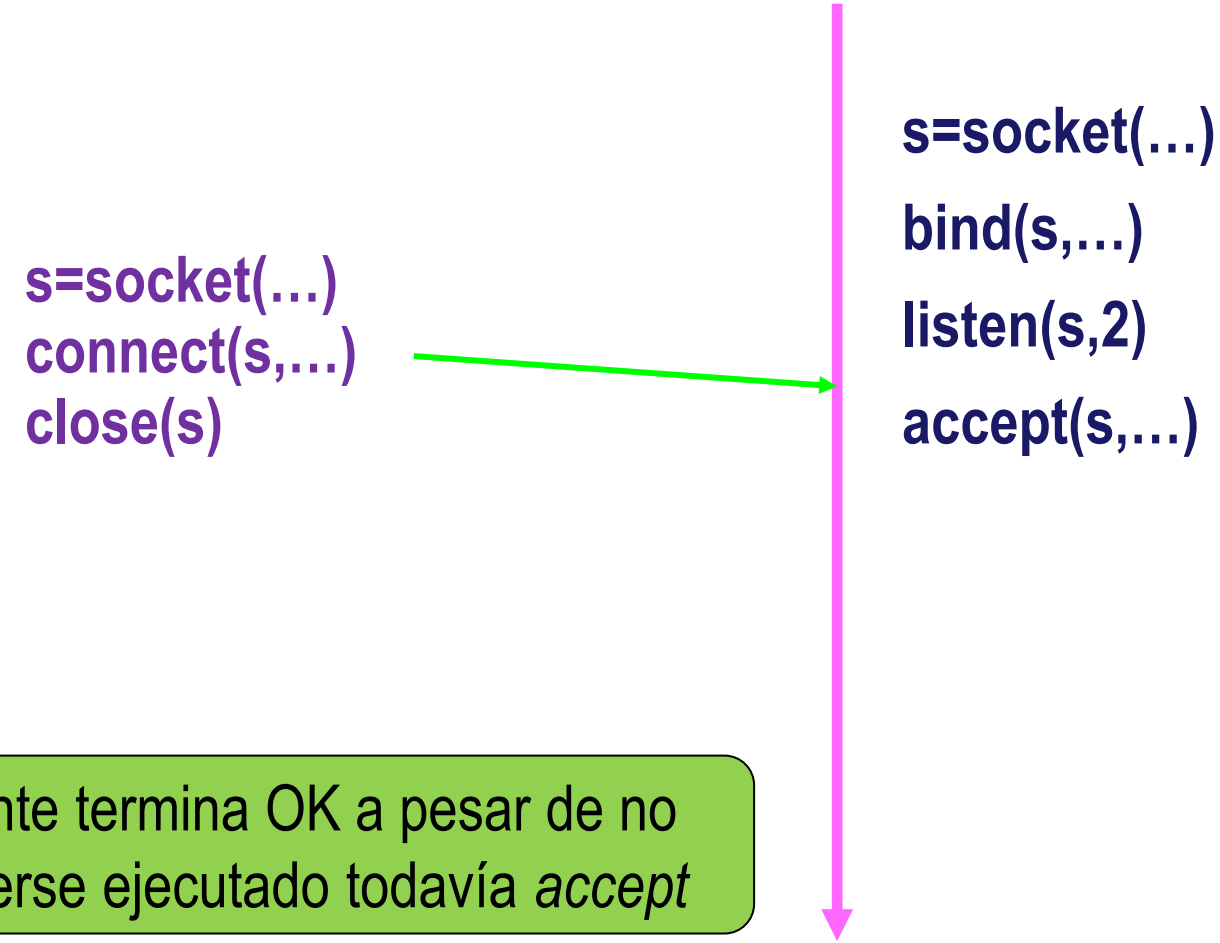
– Programa *acepta puerto val_listen*
  - bucle acepta conexiones por ese puerto con ese valor para *listen*

– Programa *conecta máquina puerto*:
  - realiza una conexión

# *connect* antes de *listen*

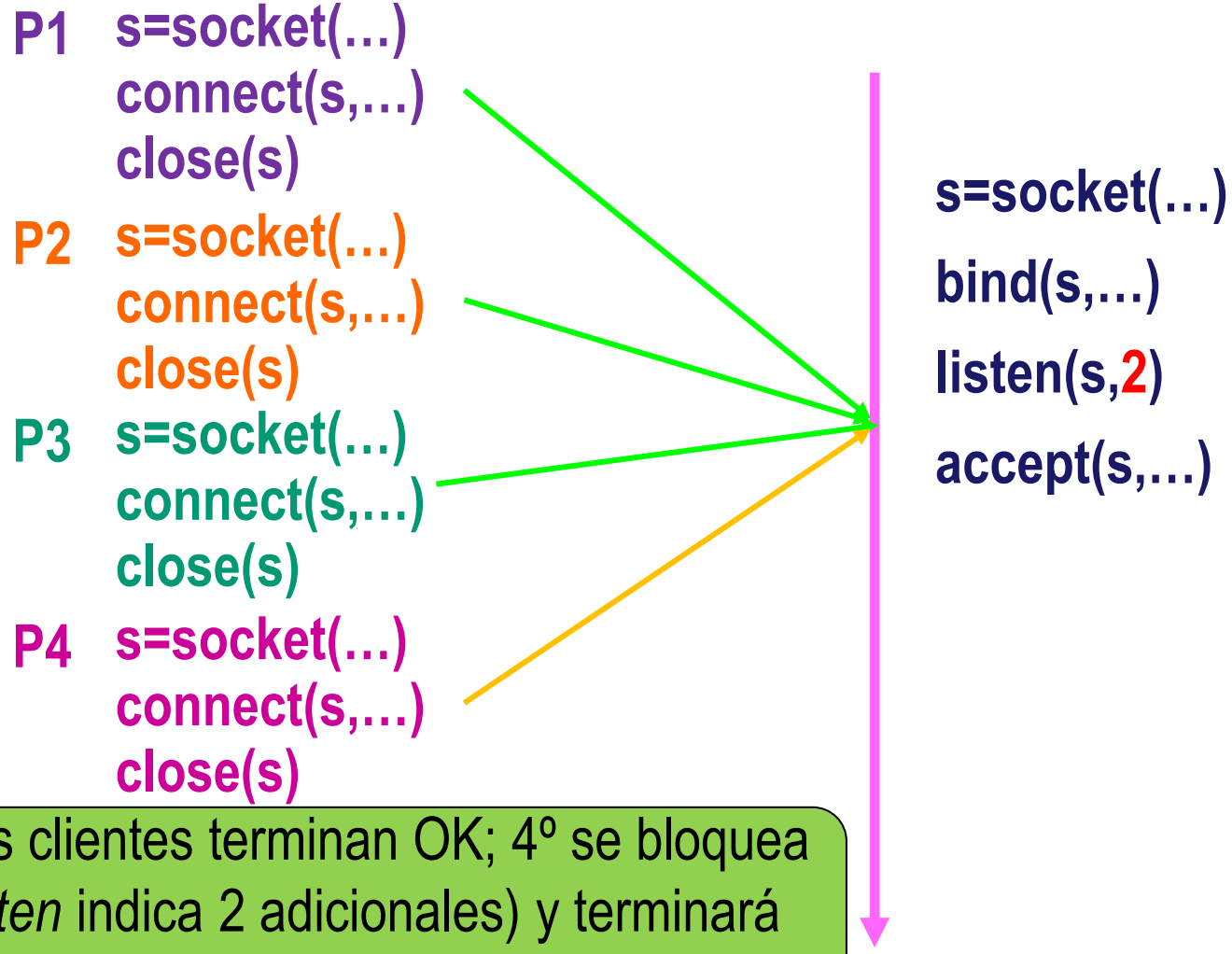


# *connect* antes de *accept*




cliente termina OK a pesar de no haberse ejecutado todavía *accept*

# Varios *connect* antes de *accept*




3 primeros clientes terminan OK; 4º se bloquea (**2** del *listen* indica 2 adicionales) y terminará cuando servidor vaya aceptando conexiones

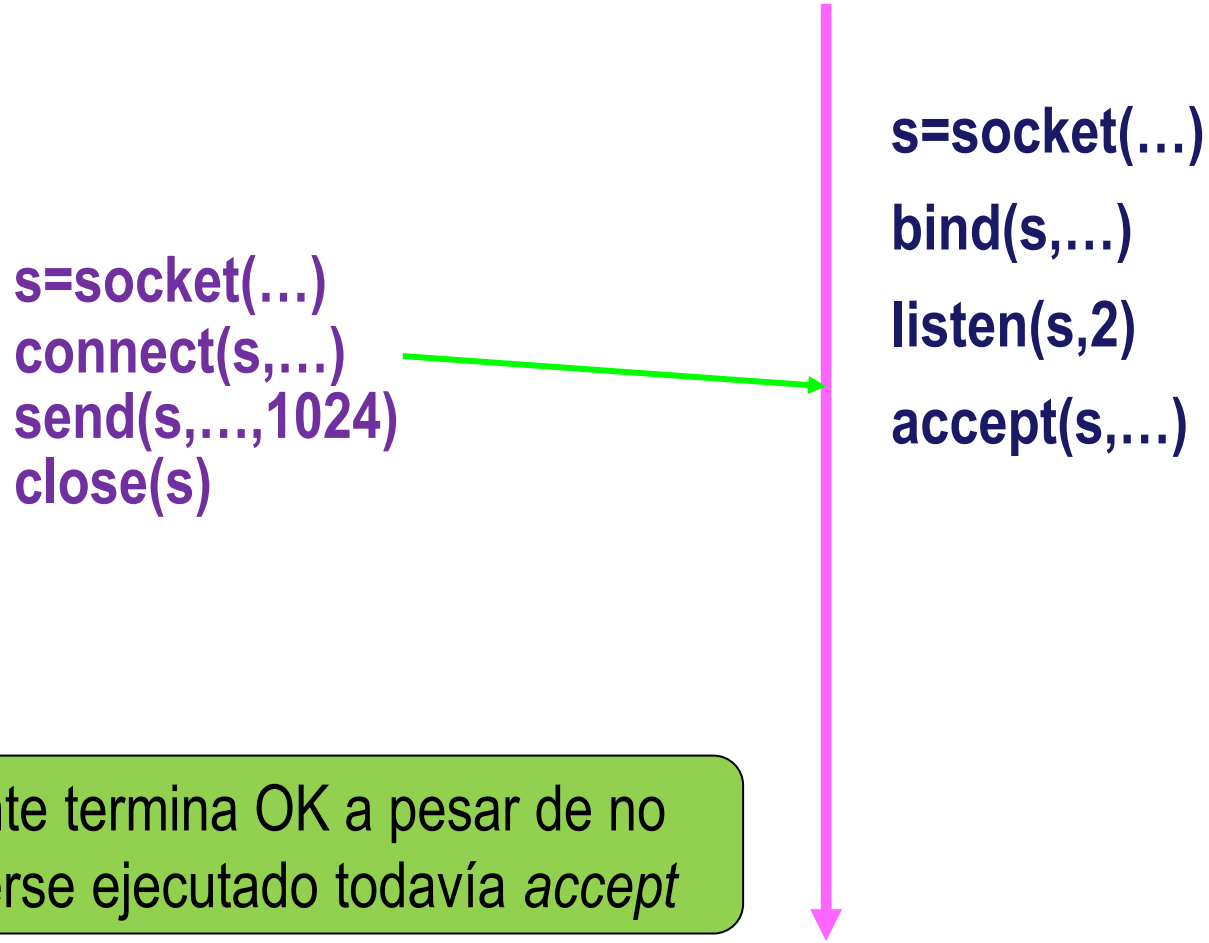
# Experimentando con envío por *stream*

- Experimentaremos con 2 programas

– Programa *receptor puerto tam_recibir*
  - bucle acepta conexiones por ese puerto
  - una vez conectado, bucle lee del socket solicitando ese tamaño
    - hasta que emisor cierra la conexión (*read* o *recv* devuelven 0)

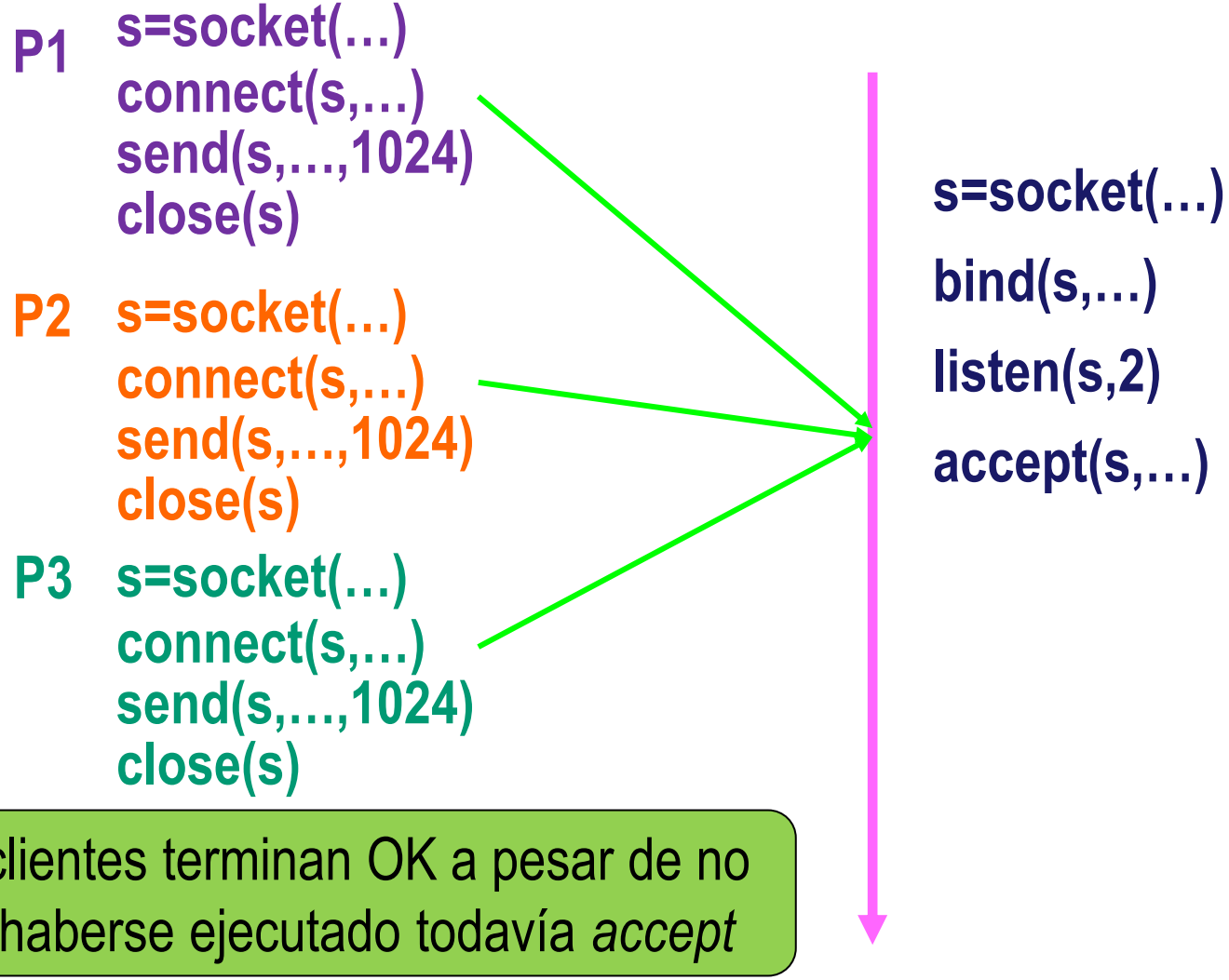
– Programa *emisor máquina puerto tam_enviar*
  - realiza una conexión y envía un mensaje de ese tamaño

# *send* antes de *accept*



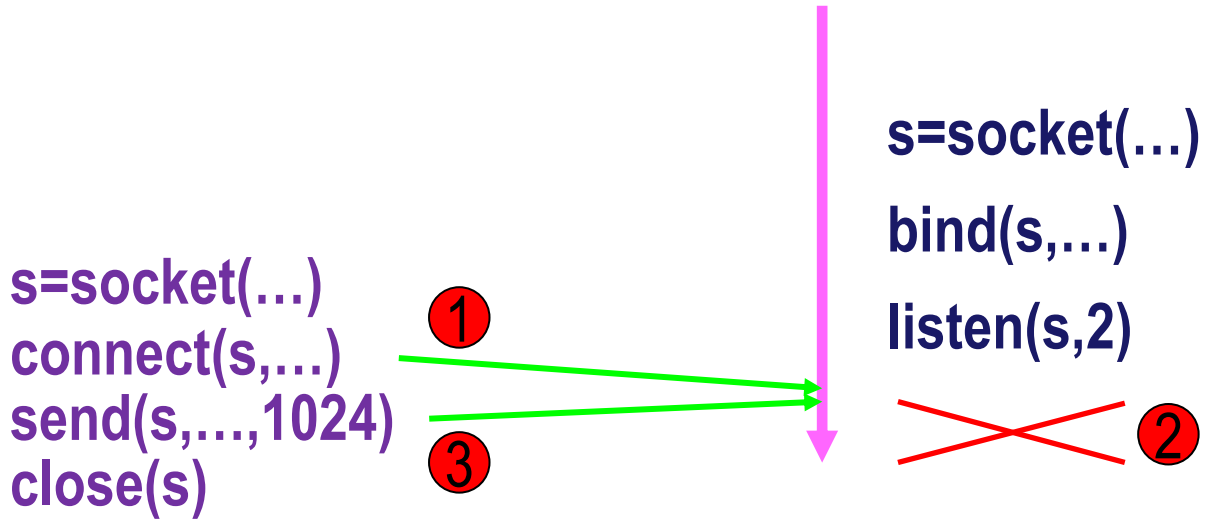

cliente termina OK a pesar de no haberse ejecutado todavía *accept*

# Varios *send* antes de *accept*



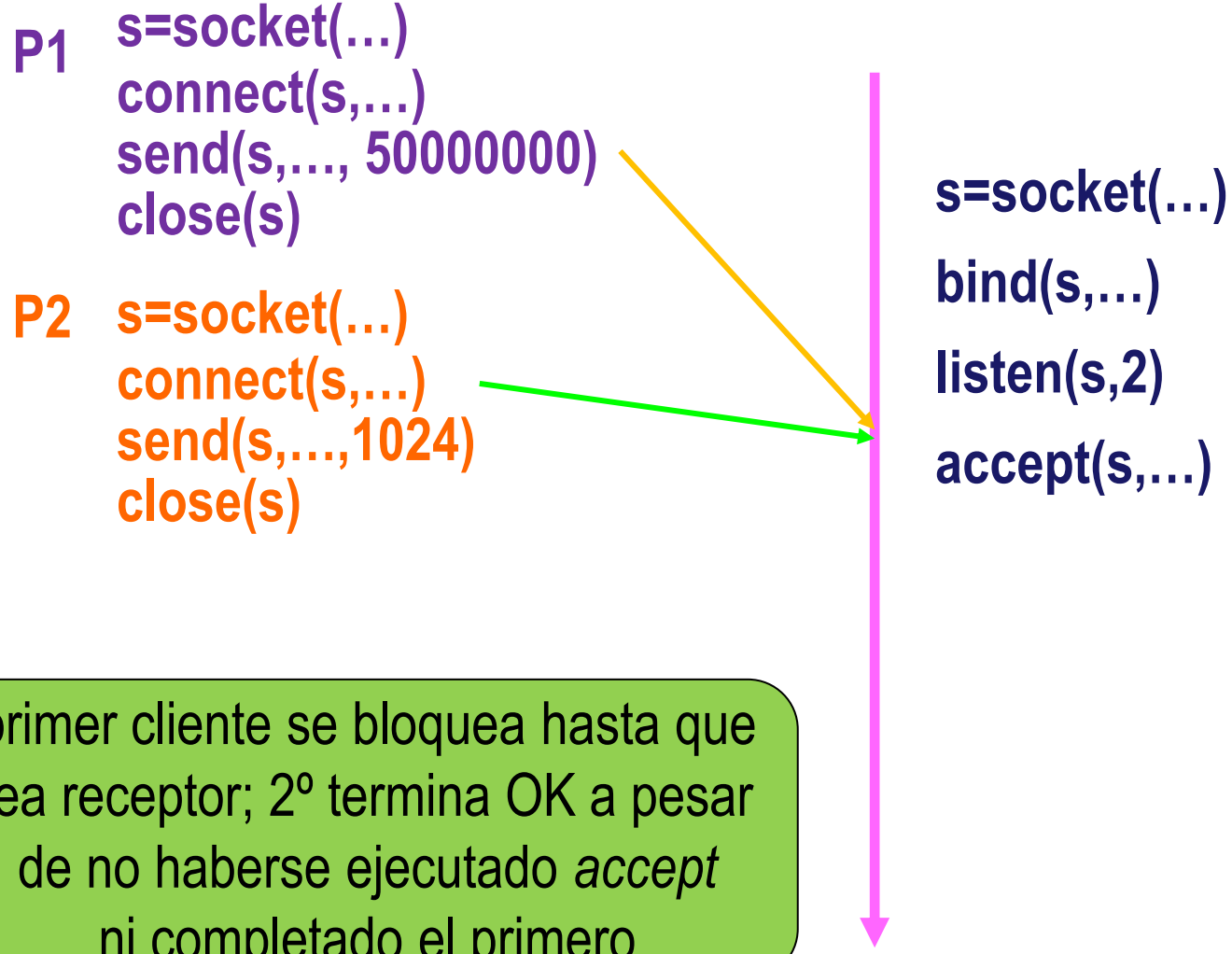

clientes terminan OK a pesar de no haberse ejecutado todavía *accept*

# *send* antes de *accept* y caída servidor



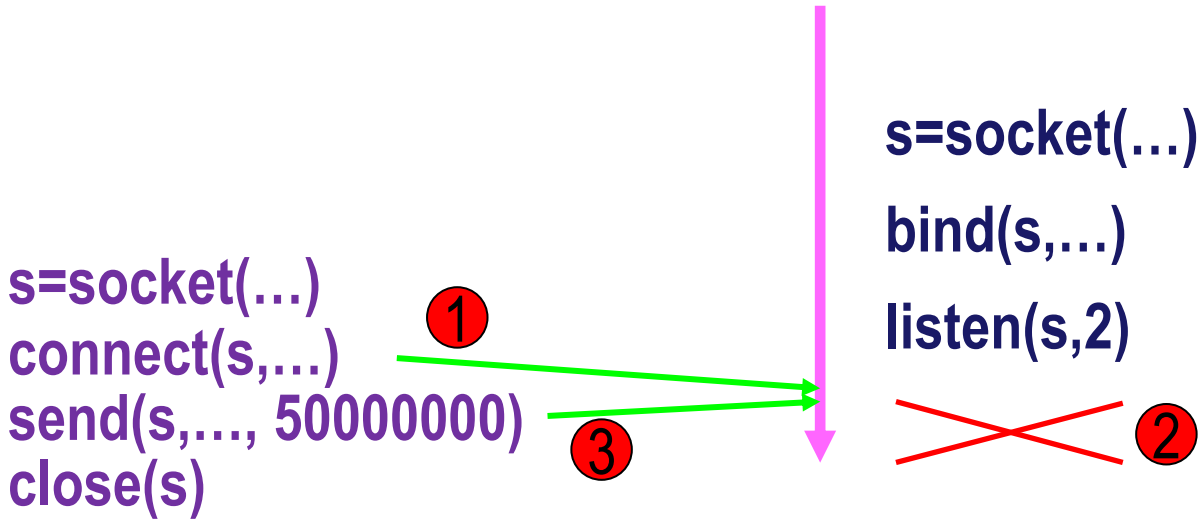

cliente termina OK a pesar de que servidor se cae después de *connect* pero antes de *accept*

# *send* grande antes de *accept*




primer cliente se bloquea hasta que lea receptor; 2º termina OK a pesar de no haberse ejecutado *accept* ni completado el primero

# *send* grande y caída del servidor




*send* devuelve menos de 50000000
pero no retorna un error

# Envío de datos por socket *stream*

- Semántica similar a tubería
  - Por cada socket, SO usa un *buffer* de transmisión
- Envío (*send*): copia datos en ese *buffer*
  - Si hay sitio suficiente, retorna inmediatamente
    - Como solo copia, puede no devolver error aunque destino caído
  - Si no, bloqueo hasta que pueda copiar **todo**
    - Según vayan transmitiéndose datos va quedando espacio
    - Se desbloqueará cuando se haya copiado todo al *buffer*
- *send* normalmente devuelve valor = nº de bytes pedidos
- Pero puede devolver valor < en ciertas situaciones patológicas:
  - envío grande (bloqueante) y receptor cae o cierra socket sin leer todo
  - No da error en ese envío, pero si emisor realiza envío adicional:
    - Error ECONNRESET o error EPIPE y señal SIGPIPE
    - Por seguridad, comprobar errores en *send*/*write* e ignorar SIGPIPE

# Envío no bloqueante

- Modo de operación no bloqueante (*fcntl* con *O_NONBLOCK*)
  - Copia solo lo datos que caben en *buffer* y retorna
    - Devuelve cuántos bytes se han copiado
  - Si no cabe nada, error (-1 *errno* = EAGAIN o EWOULDBLOCK)
- Ejemplo *emisor_nb*

```
int flags= fcntl(s, F_GETFL, 0); // hay que añadirlo a valores previos
fcntl(s, F_SETFL, flags | O_NONBLOCK);
```

- Experimente con un envío grande
- ¿Cuándo volver a intentar escribir lo que falta?
  - Bucle que lo intenta continuamente realiza espera activa
  - Se complementa con mecanismo que avisa cuándo volver a escribir
    - *select*, *poll*, *epoll*…

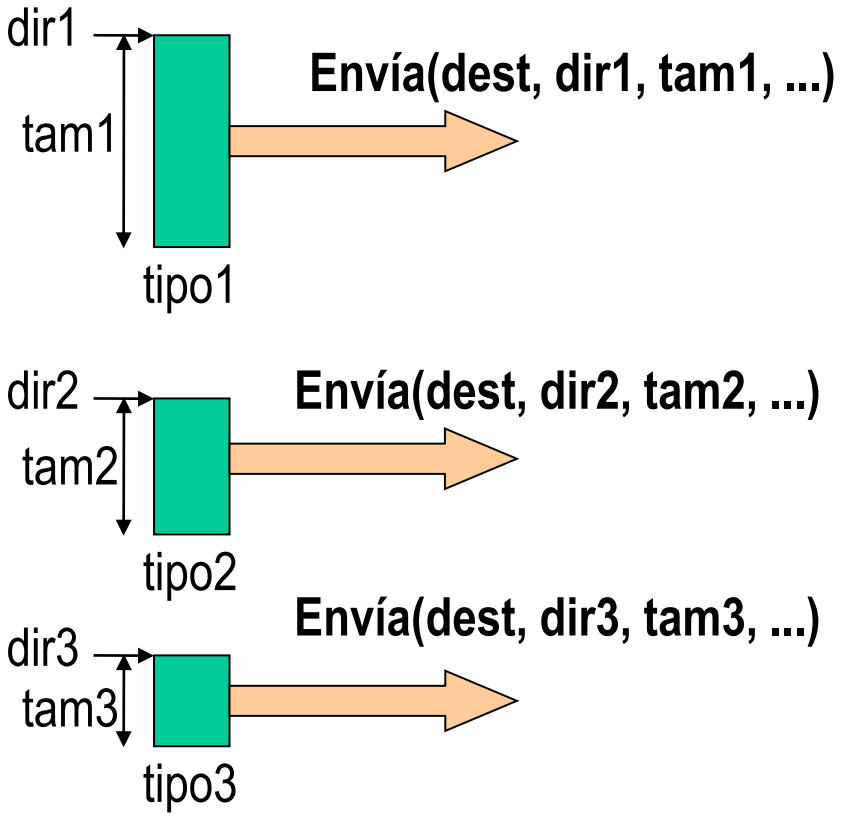
# Envío asíncrono

- Modo de operación asíncrono (*aio_write*) ≠ no bloqueante
- SO toma nota y servicio retorna inmediatamente sin copia
- Proceso continúa ejecución mientras SO realiza transferencia
  - Puede especificarse mecanismo de notificación fin de OP (p.e. señal)
  - Programa no puede reusar *buffer* hasta fin de OP
    - El SO está accediéndolo directamente para ir enviando los datos
- E/S asíncrona en Linux deficiente:
  - Uno de los puntos débiles de ese SO
  - Implementada en biblioteca en modo usuario con un *thread*
- Soluciones basadas en eventos prefieren envíos no bloqueantes

# Envío de múltiples datos

- Escenario habitual; Alternativas:
  - Un envío (una llamada al sistema) por cada dato
    - Sobrecarga de cambios de modo usuario a sistema y viceversa
    - Envíos separados pueden acabar en mensajes independientes
  - Copiar datos en una zona continua (en una estructura) y enviarla
    - Coste de la copia de los datos
  - Uso funciones *gather*: *writev* (uso general); *sendmsg* (solo sockets)
    - Permiten especificar múltiples *buffers* en una sola llamada
- Aplicación debe evitar hacer copias de datos
- *Zerocopy*: en trasmisión de datos
  - Reducir al mínimo (≈ a cero) copias entre zonas de memoria
  - S.O. y hardware de comunicación colaboran para intentarlo
  - Aplicación no debería estropearlo

# Datos múltiples: varios envíos




sobrecarga de llamadas + fragmentación de mensajes

# Datos múltiples: varios envíos

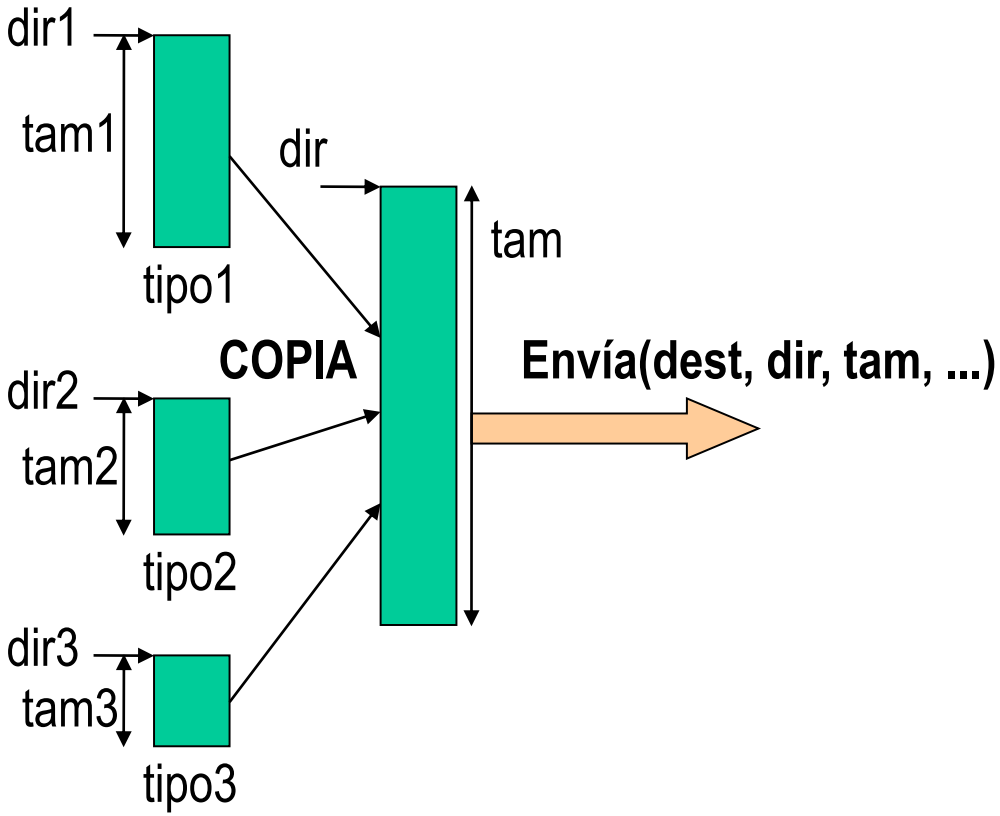
- Experimento: cliente envía *argv[3]* y *argv[4]*
- Ejemplo *envio_multiple_2send*

```
write(s, argv[3], strlen(argv[3]));
write(s, argv[4], strlen(argv[4]));
```

- Optimización: para reducir fragmentación de mensajes
  - Informar al SO de que se van a enviar más mensajes (*MSG_MORE*)
  - SO espera por envíos adicionales para que el paquete esté más lleno
- Ejemplo *envio_multiple_2send_more*

```
send(s, argv[3], strlen(argv[3]), MSG_MORE);
send(s, argv[4], strlen(argv[4]), MSG_MORE);
```

# Datos múltiples: envío con copia




sobrecarga por copias

# Datos múltiples: envío con copia

- Solo una llamada pero sobrecarga por copia
- Necesidad de enviar tamaño máximo
- Ejemplo *envio_multiple_copia*

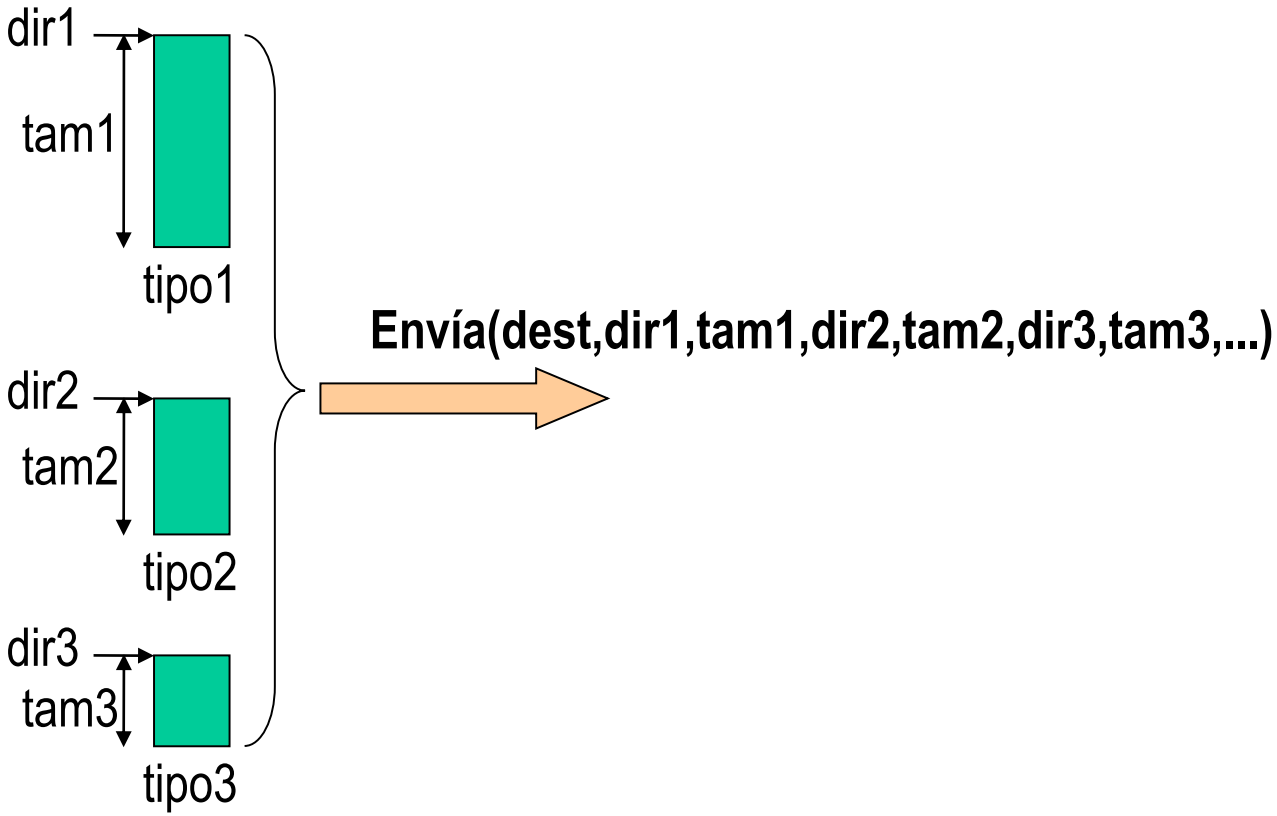
```
#define MAX_DATO 16
struct mensaje {
        char dato1[MAX_DATO];
        char dato2[MAX_DATO];
};
struct mensaje m;
strcpy(m.dato1,argv[3]);
strcpy(m.dato2,argv[4]);
write(s, &m, sizeof(m));
```

# Datos múltiples: ¿envío sin copia?

- **No funciona**: envía las direcciones de los datos, no sus valores
- Ejemplo *envio_multiple_error*

```
struct mensaje {
        char *dato1;
        char *dato2;
};
struct mensaje m;
m.dato1=argv[3];
m.dato2=argv[4];
write(s, &m, sizeof(m));
```

# Datos dispersos: envío *gather*




Uso de *writev*

# Datos múltiples: envío con writev

- Solo una llamada sin sobrecarga por copia
- Ejemplo *envio_multiple_writev*

```
struct iovec iov[2];
iov[0].iov_base=argv[3];
iov[0].iov_len=strlen(argv[3]);
iov[1].iov_base=argv[4];
iov[1].iov_len=strlen(argv[4]);
writev(s, iov, 2);
```

# Experimentando con la recepción

- Ejecutemos el programa receptor y emisor con mismo tamaño
  **./receptor 12345 50000000**
  **./emisor maq_receptor 12345 50000000**
- El receptor ha necesitado varias operaciones para recibir todo
- ¿Por qué el *read*/*recv* no espera a recibir todo?
- Esa es la esencia de los sockets *stream*:
  - **Están orientados a transmitir un flujo de datos**
  - Se considera que aplicación está interesada en ser desbloqueada
    - Incluso aunque no haya llegado todavía todo lo pedido
    - Para que puede ir trabajando con esos datos
  - En el destino se funden los mensajes
  - Lectura puede obtener parte de un mensaje o de varios
- Es la misma semántica que la lectura de una tubería

# Recepción de datos por socket *stream*

- Puede devolver menos de los pedidos
- Como en las tuberías, asimetría entre escritura y lectura
  - Escritura no se completa hasta que se haya copiado todo en el *buffer*
  - Lectura devuelve los datos presentes aunque sean menos de lo pedido
- Devuelve 0 cuando el otro extremo cierra la conexión
- Recepción no bloqueante (*fcntl* con *O_NONBLOCK*)
  - Si no hay datos, error (-1 *errno* = EAGAIN o EWOULDBLOCK)
  - Se complementa con mecanismo que avisa cuándo hay datos
    - *select*, *poll*, *epoll*…
- Recepción asíncrona (*aio_read*)
  - Mismas consideraciones que sobre el envío asíncrono
- Soluciones basadas en eventos: recepciones no bloqueantes

# Recepción datos con tamaño conocido

- Muchas aplicaciones no encajan en el modelo *stream*
  - No pueden procesar un dato hasta que no lo tengan completo
- Alternativas para asegurar que se reciben *N* bytes
  - Bucle que repite la llamada de recepción y va acumulando hasta *N*
    - Implementación "popular": función *readn*
    - Ejemplo *receptor_completo_readn*
  - Uso de funciones de la biblioteca de un lenguaje
    - Ejemplo *receptor_completo_fread*
  - Uso del *flag MSG_WAITALL* en *recv*
    - Ejemplo *receptor_completo_waitall*

**recv**(s_conec, buf, tam, **MSG_WAITALL**)

# Recepción con funciones del lenguaje

- Facilita la programación
  - Permite esperar por todos los datos (en C *fread*)
    - Internamente llama a *recv/read* todas las veces que sean necesarias
  - Y usar la funcionalidad de la E/S del lenguaje
    - P.e. leer del socket hasta que llegue un fin de línea (en C *fgets*)

- Ejemplo *receptor_completo_fread*:

```
FILE *soc_desc = fdopen(s_conec, "r");
while ((leido=fread(buf, 1, tam, soc_desc))>0) {
        printf("leido %d\n", leido);
        total+=leido;
}
fclose(soc_desc);
```

# Manejo de datos con tamaño variable

- Algunas alternativas
- Enviar longitud del dato
  - Para evitar dos mensajes o copias, se puede usar *writev*
- Enviar un carácter separador
  - Hay que asegurarse de que no aparece en el dato a enviar
  - Receptor lee hasta que lo encuentra
- Cerrar la conexión después del envío
  - Provoca que *read*/*recv* devuelva 0
    - Pero si es un cliente, necesita dejarlo abierto para recibir la respuesta
  - Puede usar *shutdown* en vez de *close* y solo cerrar el envío
    - Pero impide que cliente pueda mantener conexión persistente
- Web usa una mezcla de las dos primeras alternativas
  - Una línea vacía para indicar cuándo termina la cabecera
  - Campo *content-length* en la cabecera indica el tamaño del cuerpo

# Datos tamaño variable: emisor

- Ejemplo *envio_tam_variable_writev*:
  - Envía 2 datos de t. variable (*argv[3]* y *argv[4]*) pero antes su longitud

```
 struct cabecera {
        int long1; int long2;
};
struct cabecera cab;
cab.long1=htonl(strlen(argv[3])); // transforma de local a formato de red
cab.long2=htonl(strlen(argv[4])); // transforma de local a formato de red
struct iovec iov[3];
iov[0].iov_base=&cab; iov[0].iov_len=sizeof(cab);
iov[1].iov_base=argv[3]; iov[1].iov_len=strlen(argv[3]);
iov[2].iov_base=argv[4]; iov[2].iov_len=strlen(argv[4]);
writev(s, iov, 3);
```

# Datos de tamaño variable: receptor

- Ejemplo *receptor_tam_variable*:

```
struct cabecera cab;
recv(s_conec, &cab, sizeof(cab), MSG_WAITALL);
int tam1=ntohl(cab.long1); // transforma de formato de red a local
int tam2=ntohl(cab.long2); // transforma de formato de red a local
char *dato1 = malloc(tam1+1);
char *dato2 = malloc(tam2+1);
recv(s_conec, dato1, tam1, MSG_WAITALL);
recv(s_conec, dato2, tam2, MSG_WAITALL);
dato1[tam1]='\0'; // asegura que es un string bien formado
dato2[tam2]='\0'; // asegura que es un string bien formado
```

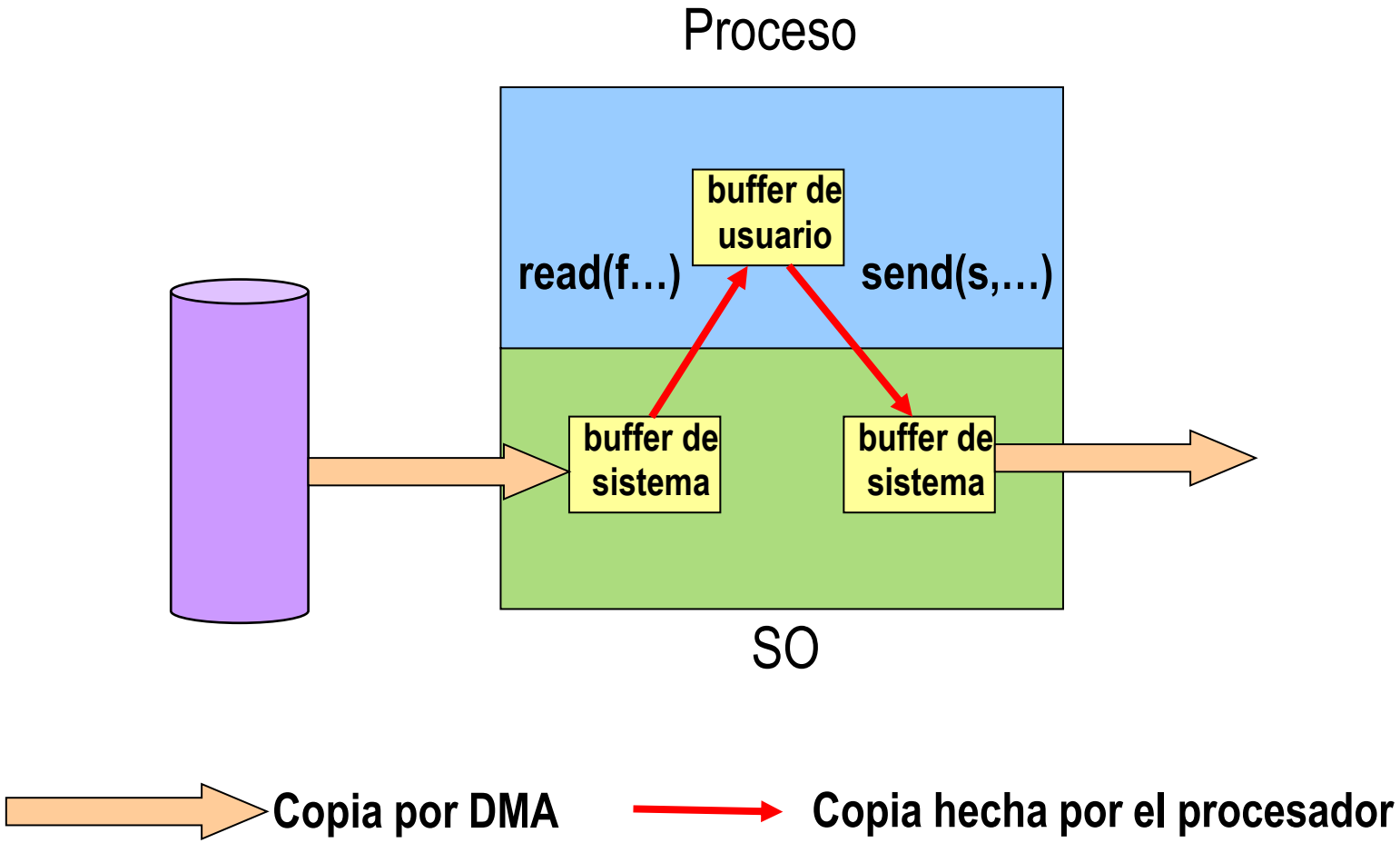
- Cuidado con las transferencias de cadenas de caracteres:
  - Si emisor envía *strlen*, no está incluido el carácter nulo
  - Tendrá que añadirlo el receptor para que la cadena sea válida

# Ejemplo: servidor web

- Permite ilustrar algunas de las soluciones planteadas
- Modo de operación en descarga de una página:
  - Recibe petición, prepara cabecera, envía cabecera y contenido fichero
- Proponemos tres alternativas
  - Uso de *read* y *send* (***srv_web_read_send***)
  - Uso de *mmap* y *writev* (***srv_web_mmap_writev***)
  - Uso de *sendfile* (***srv_web_sendfile***)
- Experimento

```
truncate -s 1G BIG.html # crea fichero de 1G vacío
./srv_web_read_send 23456 # se lanza una de las versiones
wget -O - localhost:23456/BIG.html > /dev/null # descarga fichero
```

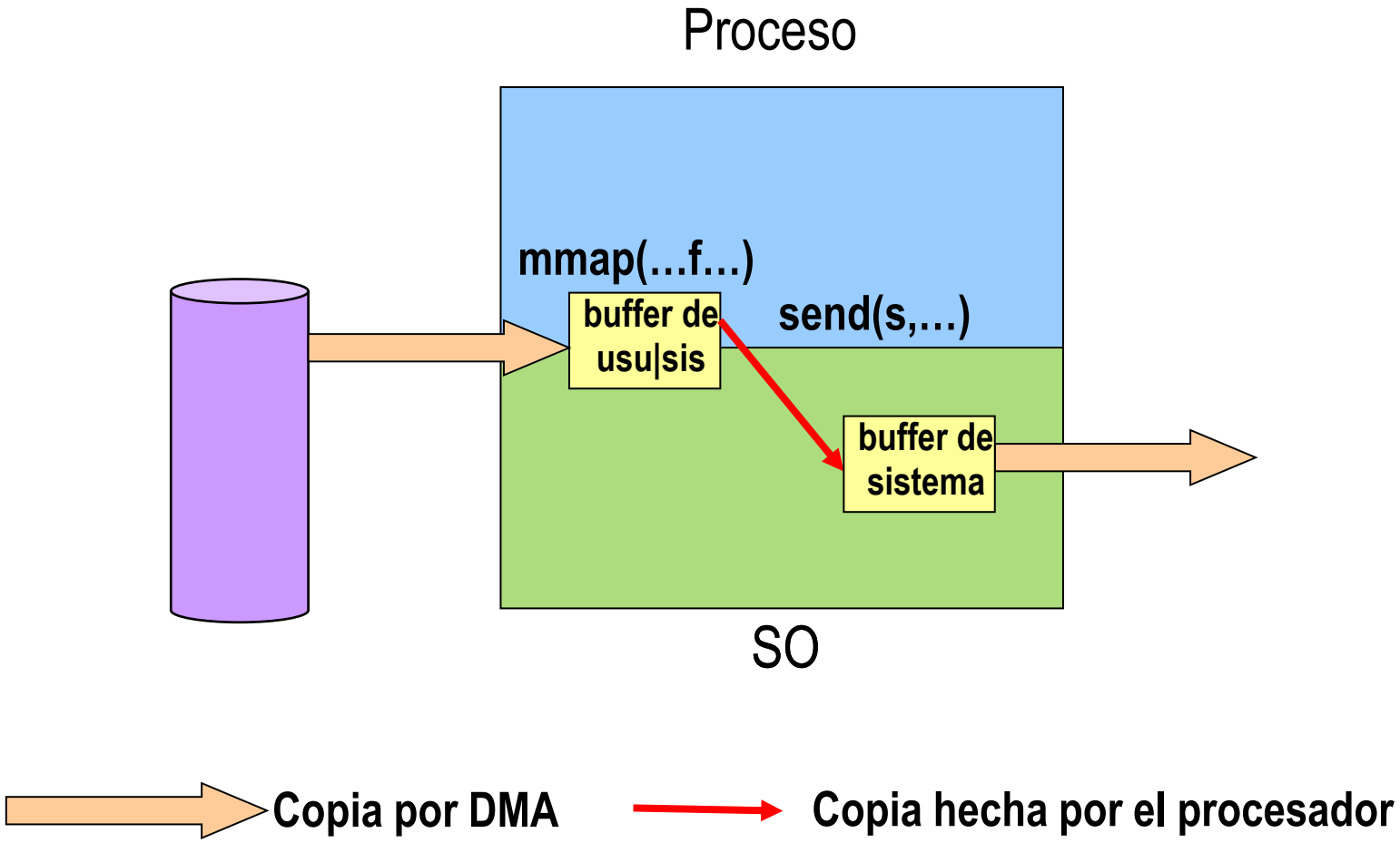
# Envío convencional de fichero



# Servidor web con *read* y *send*

```
while (1) {
    s_conec=recibir_peticion(s, pagina);
    f=abrir_fichero(pagina, &tam_fich, &tam_acceso_recom);
    preparar_cabecera(tam_fich, cabecera);
    char buf[tam_acceso_recom];
    send(s_conec, cabecera, strlen(cabecera), MSG_MORE);
    while ((leido=read(f, buf, tam_acceso_recom))>0)
        send(s_conec, buf, leido, MSG_MORE);
    close(s_conec);
    close(f);
}
```

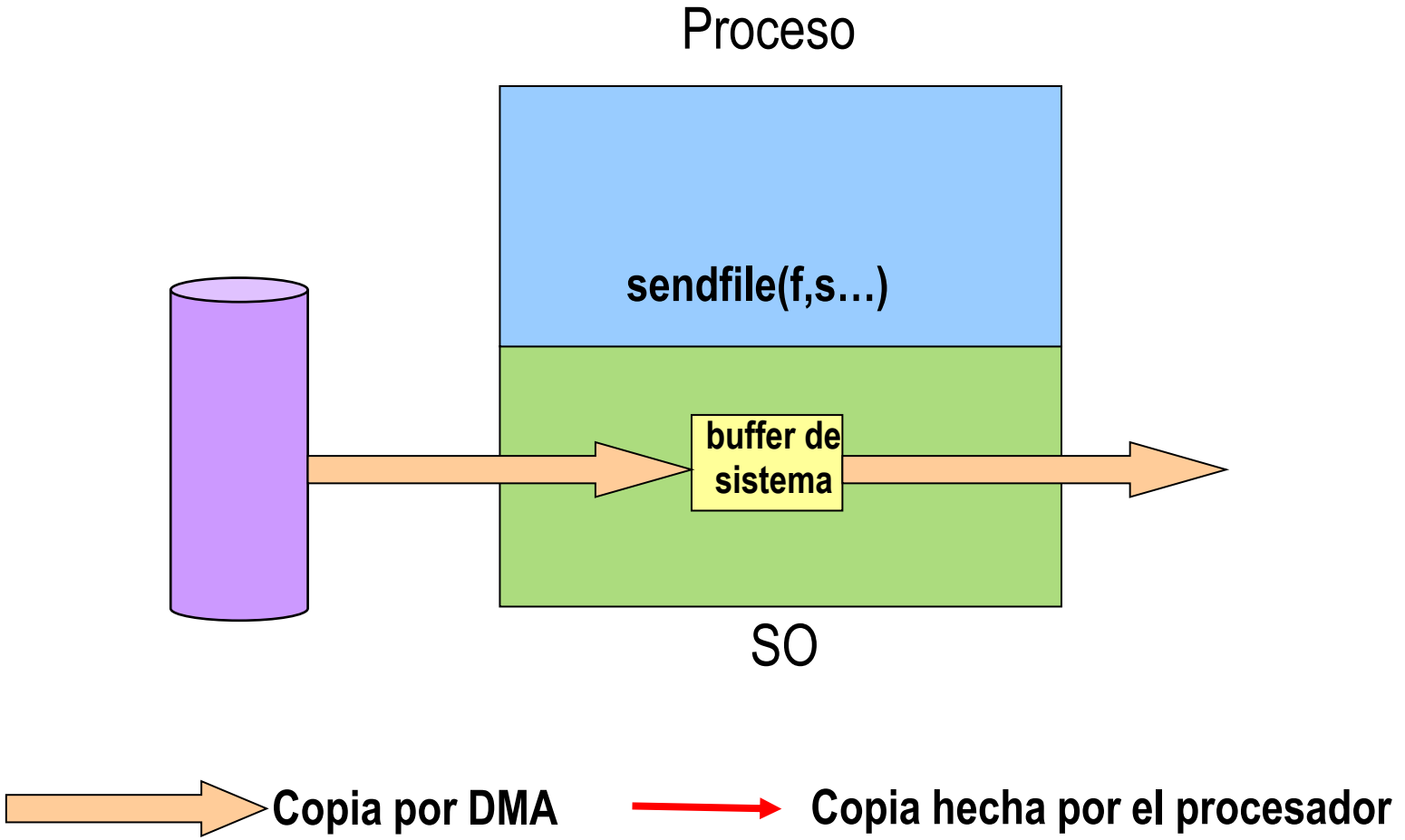
# Envío con proyección de fichero



# Servidor web con *mmap* y *writev*

```
while (1) {
    s_conec=recibir_peticion(s, pagina);
    f=abrir_fichero(pagina, &tam_fich, &tam_acceso_recom);
    preparar_cabecera(tam_fich, cabecera);
    p = mmap(NULL, tam_fich, PROT_READ, MAP_PRIVATE, f, 0);
    close(f);
    iov[0].iov_base = cabecera; iov[0].iov_len = strlen(cabecera);
    iov[1].iov_base = p; iov[1].iov_len = tam_fich;
    writev(s_conec, iov, 2);
    close(s_conec);
    munmap(p, tam_fich);
}
```

# Envío zero-copy de fichero



# Servidor web con *sendfile*

```
while (1) {
    s_conec=recibir_peticion(s, pagina);
    f=abrir_fichero(pagina, &tam_fich, &tam_acceso_recom);
    preparar_cabecera(tam_fich, cabecera);
    send(s_conec, cabecera, strlen(cabecera), MSG_MORE);
    sendfile(s_conec, f, NULL, tam_fich);
    close(s_conec);
    close(f);
}
```

# Diseño de un ejemplo de servicio

- Servidor recibe como petición un bloque de datos
  - Le da la vuelta (*hola* → *aloh*) y lo envía al cliente
- No encaja con modelo *stream*
  - Hasta que no se reciba toda la petición no puede procesarse
  - Se necesitan gestionar datos de tamaño variable
- Proponemos cuatro alternativas (como vimos en tema 2)
  - Secuencial (***srv_rec_sec***)
  - Concurrente con procesos dinámicos (***srv_rec_prc***)
  - Concurrente con *threads* dinámicos (***srv_rec_thr***)
  - Basado en eventos (***srv_rec_evn***): solución más compleja
    - Solo se muestra para resaltar esa complejidad
- Cliente (***cln_rev***)
  - Recibe ficheros como argumentos
  - Realiza una petición por cada uno manteniendo la conexión

# Cliente *cln_rev*

```
connect(s, (struct sockaddr *)&dir, sizeof(dir));
for (int i=3; i<argc; i++) {
    f = open(argv[i], O_RDONLY); fstat(f, &st);
    int tam=st.st_size; int tamn=htonl(tam);
    void *p = mmap(NULL, tam, PROT_READ|PROT_WRITE,
            MAP_PRIVATE, f, 0); close(f);
    struct iovec iov[2];
    iov[0].iov_base=&tamn; iov[0].iov_len=sizeof(tamn);
    iov[1].iov_base=p; iov[1].iov_len=tam;
    writev(s, iov, 2);
    recv(s, p, tam, MSG_WAITALL);
    write(1, p, tam);
munmap(p, tam); } // fin del for
close(s); // fin del cliente
```

# Servidor secuencial *srv_rev_sec*

```
while(1) {
    tam_dir=sizeof(dir_cliente);
    s_conec=accept(s, (struct sockaddr *)&dir_cliente, &tam_dir);
    // nuevo cliente
    while (recv(s_conec, &tam, sizeof(tam), MSG_WAITALL)>0) {
        // nueva petición del cliente
        int tamn=ntohl(tam); char *dato = malloc(tamn);
        recv(s_conec, dato, tamn, MSG_WAITALL);
        revierte(dato, tamn);
        send(s_conec, dato, tamn, 0);
    } // fin while interno: cliente ha terminado de pedir
    close(s_conec);
} // fin while externo: servidor ha terminado
```

# Servidor procesos *srv_rev_prc*

```
// ignorar señal SIGCLD para evitar que hijos queden zombis si no wait
while(1) {
    tam_dir=sizeof(dir_cliente);
    s_conec=accept(s, (struct sockaddr *)&dir_cliente, &tam_dir);
    if (fork()==0) { // se crea hijo para server a ese cliente
        close(s); // hijo no usa ese socket
        while (recv(s_conec, &tam, sizeof(tam), MSG_WAITALL)>0) {
            int tamn=ntohl(tam); char *dato = malloc(tamn);
            recv(s_conec, dato, tamn, MSG_WAITALL);
            revierte(dato, tamn); send(s_conec, dato, tamn, 0);
        }
        close(s_conec); exit(0); } // fin hijo: cliente ha terminado de pedir
    close(s_conec); // padre no usa ese socket
} // fin del servidor
```

# Servidor *threads srv_rev_thr*

```
void *servicio(void *arg){ // thread de servicio: 1 por cliente
   int s_srv, tam;  s_srv=(long) arg;
   while (recv(s_srv, &tam, sizeof(tam), MSG_WAITALL)>0) {
      int tamn=ntohl(tam); char *dato = malloc(tamn);
      recv(s_srv, dato, tamn, MSG_WAITALL);
      revierte(dato, tamn); send(s_srv, dato, tamn, 0);} // fin while: fin cliente
   close(s_srv);  return NULL; }
```

---

```
// main
pthread_attr_init(&atrib_th); // evita pthread_join
pthread_attr_setdetachstate(&atrib_th,PTHREAD_CREATE_DETACHED);
while(1) {
   tam_dir=sizeof(dir_cliente);
   s_conec=accept(s, (struct sockaddr *)&dir_cliente, &tam_dir);
   pthread_create(&thid, &atrib_th, servicio, (void *)(long)s_conec);}
```

# Gestión multiplexada de eventos E/S

- En operaciones no bloqueantes identificamos necesidad de
  - mecanismo que avise cuando se puede leer o escribir sin bloqueo
- Servicio de gestión de eventos de E/S
  - Notifica cuándo hay datos en un descriptor de entrada
    - En socket inicial de servidor, cuando hay conexión pendiente de aceptar
    - En socket conectado, cuando hay datos pendientes de recibir
  - Notifica cuándo se puede escribir sin bloqueo en descriptor de salida
  - Permite manejar múltiples descriptores simultáneamente
  - Espera por eventos temporizada
- Servicios UNIX: *select/poll* ineficientes con muchos descriptores
  - *epoll*: solución eficiente específica de Linux; 2 modos de operación
    - flanco: avisa cuando se produce el evento
    - nivel: avisa mientras se mantenga el evento
- Soporte de las soluciones basadas en eventos

# Modo operación servidor eventos

- Inicio: pide ser notificado cuando lleguen peticiones de conexión
- Modo de operación: bucle espera de eventos (*select/poll/epoll*)
- En cada iteración, tratamiento de los eventos activos:
  - Si petición de conexión, la acepta y pide ser notificado al llegar datos
  - Si llegada de datos por socket, se leen con operación no bloqueante
    - Si esa lectura completa la petición, se procesa
    - Si procesado requiere op. bloqueante (p.e. lee fichero), uso E/S asíncrona
    - Completado el procesado hay que enviar la respuesta:
      - Solicita ser notificado cuando el socket permite escritura sin bloqueo
  - Si socket permite escritura sin bloqueo (hay sitio en el *buffer* interno)
    - Envío no bloqueante; si completo, pide no ser notificado eventos escritura
  - Si se detecta fin de entrada de datos por un socket
    - pide no ser notificado de eventos de lectura en ese socket
- Más complejo que soluciones concurrentes (***srv_rec_evn***)

# Sockets datagrama

- Nos hemos centrado en *stream* por uso más frecuente
- Mucho de lo comentado puede aplicarse también a datagramas
  - Pero, obviamente, teniendo en cuenta que no hay conexiones
- Una diferencia importante: la integridad de los mensajes
  - En *stream* los mensajes se funden
    - *read*/*recv* puede obtener partes de distintos mensajes
  - En datagrama se mantiene la separación de los mensajes
    - Si mensaje > solicitado por *recvfrom* → el resto se pierde
    - *recvfrom* de 100 bytes no puede recibir mensajes de 2 *sendto* de 50
- Otra diferencia: establece un tamaño máximo para enviar
- Experimente con este tipo de sockets usando:
  - *receptor_dgram* y *emisor_dgram*