Tema 4
Comunicación con sockets

Sistemas Distribuidos
Grado en Ingeniería Informática
Universidad Carlos III de Madrid

Contenido

- Conceptos básicos sobre sockets
- Modelo de comunicación
- Sockets
  - Datagrama
  - *Stream*
- API de programación
  - Sockets en C
  - Sockets en Python
  - Sockets en Java
- Guía de diseño de aplicaciones cliente-servidor con sockets

Sockets: introducción

- Mecanismo de IPC que proporciona comunicación entre procesos que ejecutan en máquinas distintas
- La primera implementación apareció en 1983 en UNIX BSD 4.2
  - Intento de incluir TCP/IP en UNIX
  - Diseño independiente del protocolo de comunicación

- Un **socket** es un descriptor de un punto final de comunicación (dirección IP y puerto)

- Abstracción que:
  - Representa un extremo de una comunicación bidireccional con una dirección asociada
  - Ofrece **interfaz de acceso a la capa de transporte** del protocolo TCP/IP
    - Protocolo TCP y UDP

Sockets: introducción

- API formalmente especificado en el estándar POSIX.1g (2000)
- Actualmente disponibles en:
  - Prácticamente todos los sistemas operativos:
    - Unix, Linux
    - WinSock: API de sockets de Windows
    - Macintosh
  - En Java como clase nativa

Sockets UNIX

- Dominios de comunicación
  - AF_UNIX
  - AF_INET
  - AF_INET6
- Tipos de sockets
  - Stream (SOCK_STREAM)
  - Datragrama (SOCK_DGRAM)
- Direcciones de sockets

Dominios de comunicación

- Un dominio representa una familia de protocolos
  - Un socket está asociado a un dominio desde su creación
  - Sólo se pueden comunicar sockets del mismo dominio
  - Los servicios de sockets son independientes del dominio

- Ejemplos de dominios:
  - AF_UNIX: comunicación dentro de una máquina
  - AF_INET: comunicación usando protocolos TCP/IP (IPv4)
  - AF_INET6: comunicación usando protocolos TCP/IP (IPv6)

Tipos de sockets

- **Stream** (SOCK_STREAM)
  - Protocolo TCP
- **Datagrama** (SOCK_DGRAM)
  - Protocolo UDP
- **Raw** (SOCK_RAW)
  - Sockets sin protocolo de transporte (IP)

Sockets *stream*

- Protocolo TCP
- Flujo de datos bidireccional
- Orientado a conexión
  - Debe establecerse una conexión extremo-a-extremo antes del envío y recepción de datos
  - Flujo de bytes (no preserva el límite entre mensajes)
  - Proporciona fiabilidad
    - Paquetes ordenados por secuencia, sin duplicación de paquetes, libre de errores, notifica errores

- Ejemplos de protocolos que usan TCP:
  - HTTP, Telnet, FTP, SMTP

Sockets *datagrama*

- Protocolo **UDP**
- Flujo de datos bidireccional
- No orientado a conexión
  - No se establece/mantiene una conexión entre los procesos que comunican
  - Un datagrama es una entidad autocontenida
    - Se preservan los límites entre mensajes
  - Longitud máxima de un datagrama (datos **y** cabeceras) es **64 KB**
    - Cabecera IP+cabecera UDP = 28 bytes
  - Mantiene separación entre paquetes
  - No proporcionan fiabilidad
    - Paquetes desordenados, duplicados, pérdidas
- Ejemplo de protocolo que usa UDP:
  - DNS

Direcciones de sockets

- Las direcciones se usan para:
  - Asignar una dirección local a un socket (bind)
  - Especificar una dirección remota (connect o sendto)

- Las direcciones son dependientes del dominio
  - Se utiliza la estructura genérica struct sockaddr
  - Cada dominio usa una estructura específica
    - Direcciones en AF_UNIX (struct sockaddr_un)
      - Nombre de fichero
    - Direcciones en AF_INET(struct sockaddr_in)
      - Cada socket debe tener asignada una dirección única
        - Dirección de host (32 bits) + puerto (16 bits) + protocolo
  - Es necesario la conversión de tipos (*casting*) en las llamadas

Puertos

- Un puerto identifica un proceso destino en un computador
  - Los puertos se asocian a procesos,
    - permiten que la transmisión se dirija a un proceso específico en el computador destino
  - Un puerto tiene un único receptor y múltiples emisores (excepto *multicast*)
  - Toda aplicación que desee enviar y recibir datos debe abrir un puerto

  - Número entero de 16 bits
    - 2^{16} puertos en una máquina ~ 65536 puertos posibles
    - Reservados por la IANA para aplicaciones de Internet: 0-1023 (también llamados ***well-known*** puertos)
    - Puertos entre 1024 y 49151 son puertos registrados para ser usados por los servicios
    - Puertos por encima de 65535 para uso privado

  - El espacio de puertos para streams y datagramas es independiente
    - http://www.iana.org/assignments/port-numbers

Información asociada a una comunicación

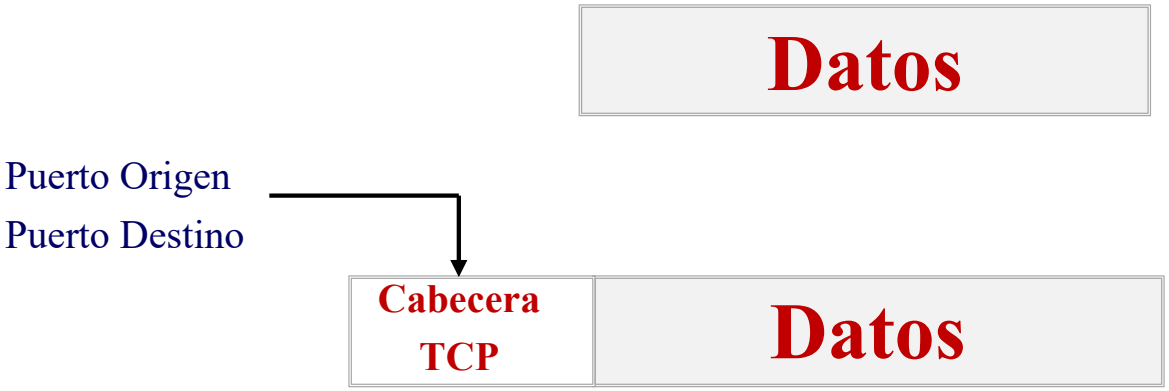
- Protocolo
  - TCP, UDP
- Dirección IP local (origen)
- Puerto local (origen)
- Dirección IP remota (destino)
- Puerto remoto (destino)

(Protocolo, IP-local, P-local, IP-remoto, P-remoto)

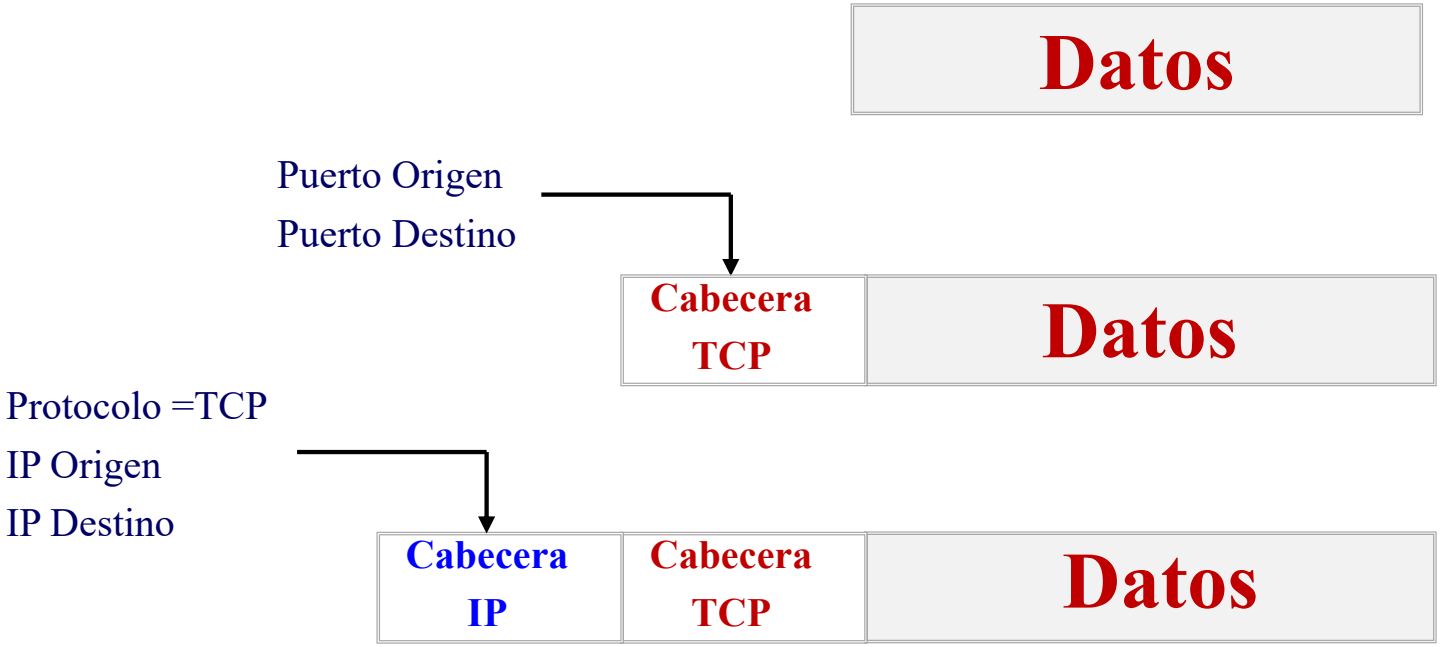
Encapsulación de un paquete TCP

Datos

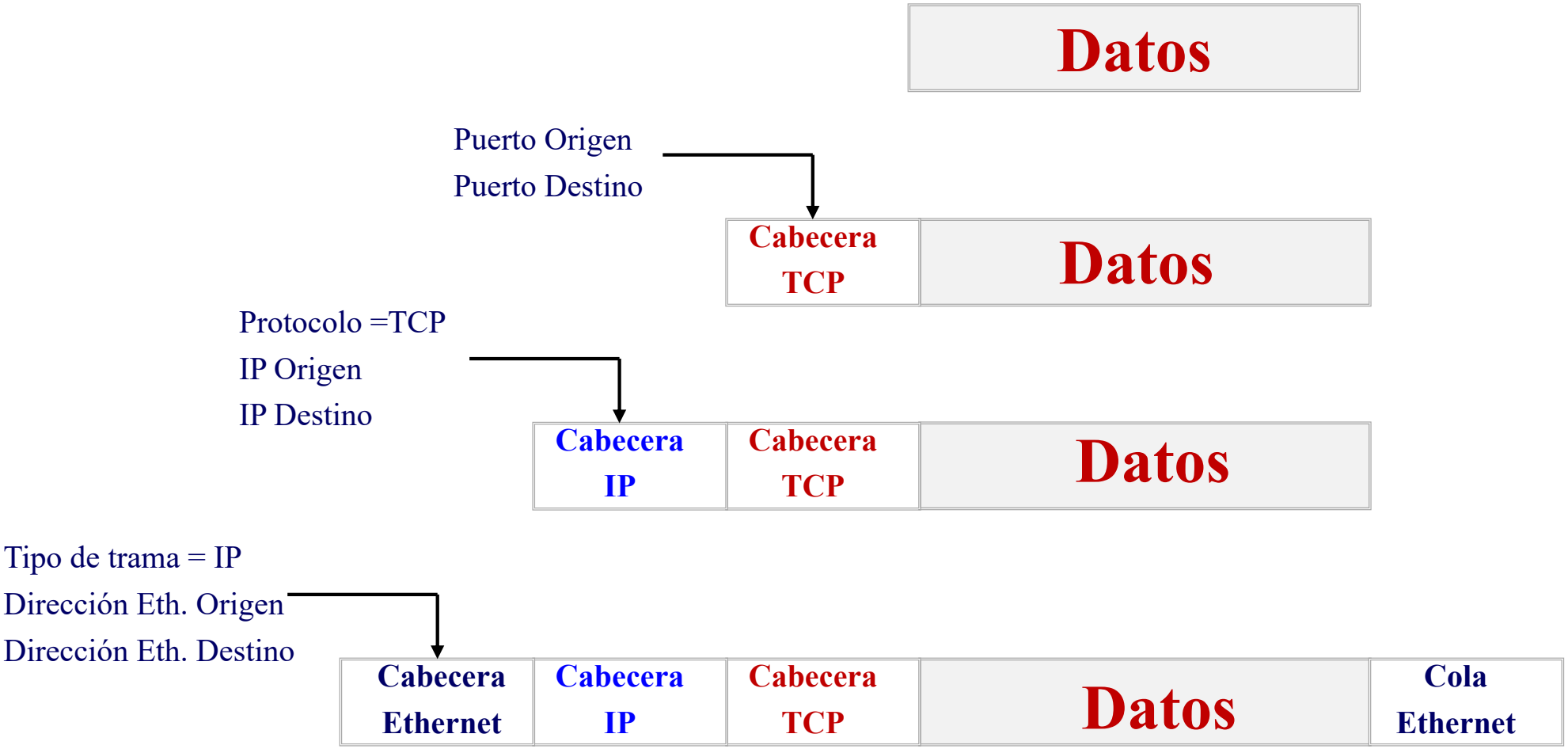
Encapsulación de un paquete TCP



Encapsulación de un paquete TCP



Encapsulación de un paquete TCP



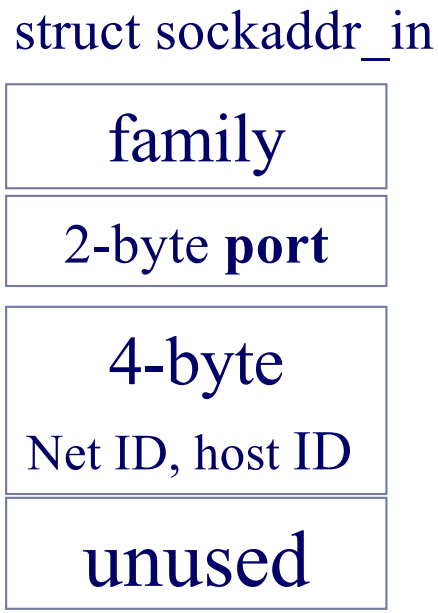
Direcciones IP

- Una dirección IP se almacena en una estructura de tipo `in_addr`

```
#include <netinet/in.h>

typedef uint32_t in_addr_t;

struct in_addr {
    in_addr_t s_addr; /*entero sin signo de 32 bits*/
};
```

Direcciones de sockets en AF_INET

- Estructura struct sockaddr_in
  - Debe iniciarse a 0
  - La estructura tiene tres campos importantes
    - `sin_family:` dominio (AF_INET)
    - `sin_port:` puerto
    - `sin_addr:` dirección del host

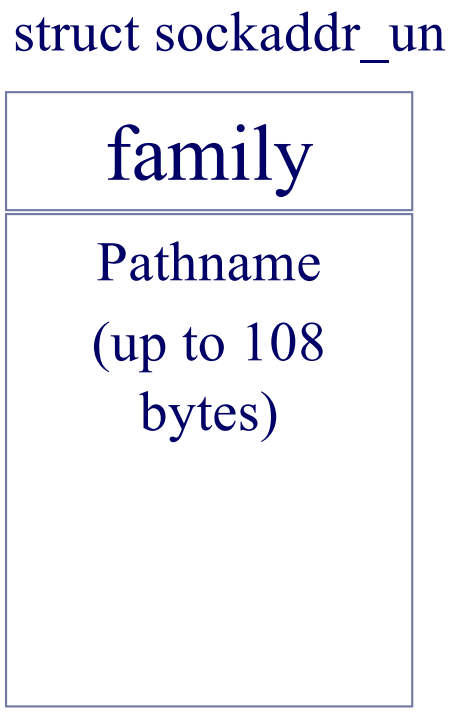



```
#include <netinet/in.h>
struct sockaddr_in  {
    short               sin_family;
    in_port_t           sin_port;  /* 16 bits sin signo*/
    struct in_addr      sin_addr;
    unsigned char       sin_zero[8];
};
```

Direcciones de sockets en AF_UNIX

- La estructura struct sockaddr_un describe la dirección de un socket AF_LOCAL o AF_UNIX

```
#include <sys/un.h>
struct sockaddr_un  {
          short         sun_family;
          char          sun_path[108];
};
```



Servicios sobre direcciones

- Obtener el nombre de un host
- Transformar direcciones
- Obtener la dirección de un host
- Representación de datos
  - Interna
  - Externa

Obtener el nombre de un host

- Función que facilita el nombre de la máquina (formato dominio punto) en la que se ejecuta:

```
int gethostname(char *name, size_t namelen);
```

donde:

name referencia al buffer donde se almacena el nombre

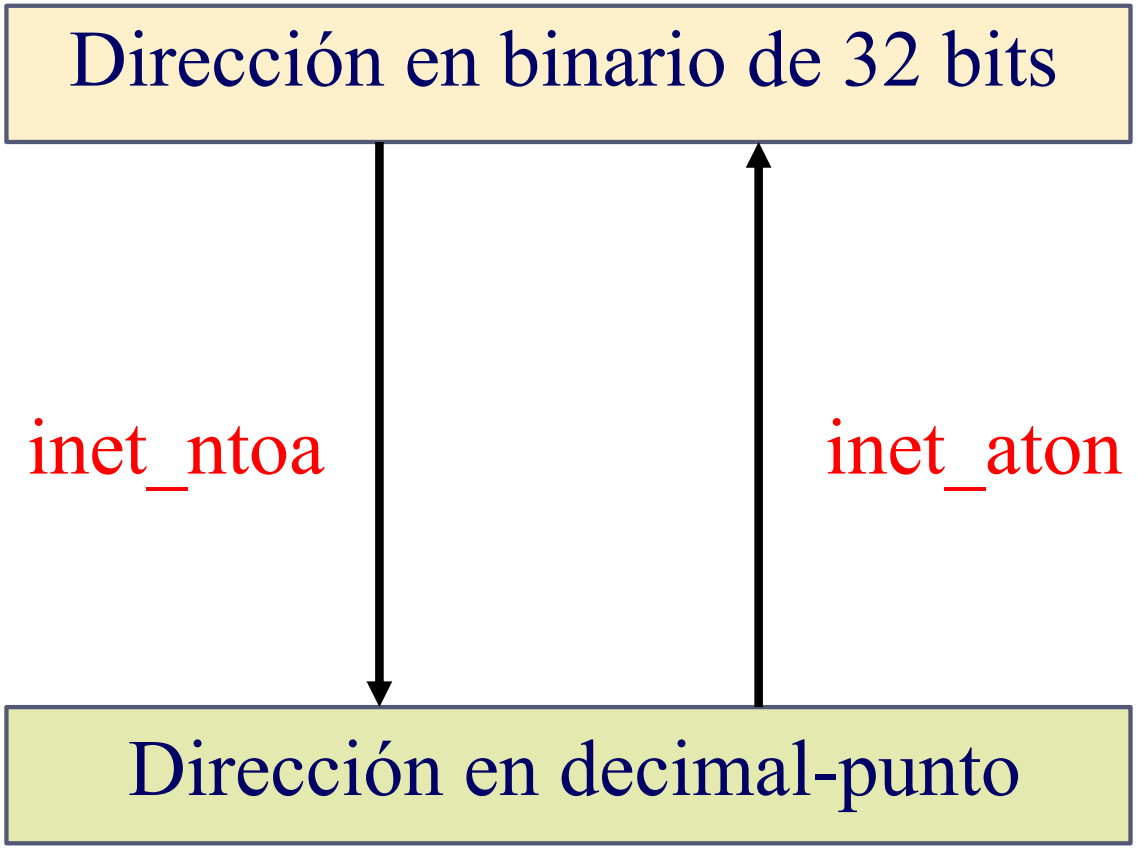
namelen longitud del buffer

Ejemplo: obtener el nombre de un host en formato dominio-punto

gethostname.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main ()
{
        char maquina[256];
        int err;

        err = gethostname(maquina, 256);

        if (err != -1)
                printf("Ejecuto en la maquina %s\n", maquina);

        exit(0);
}
```

Transformación de direcciones



Obtención de la dirección de host (IPv4)

- Usuarios manejan direcciones en forma de texto:
  - Notación decimal-punto (ej: 138.100.8.100)
  - Notación dominio-punto (ej. www.uc3m.es)

- Dos funciones de transformación de direcciones:
  - Devuelve una dirección IP en notación decimal-punto

```
char *inet_ntoa(struct in_addr in);
```

  - Obtiene una dirección IP (binaria) a partir de notación decimal punto

```
int inet_aton(const char *cp,
              struct in_addr *in);
```

Obtención de la dirección de un host (IPv4, IPv6)

- Dos funciones de transformación de direcciones:
  - Devuelve una dirección IP en notación entendible (decimal-punto, dirección hexadecimal separada por ::

```
const char *inet_ntop(int domain,
                      const void *addrptr,
                      char *dst_str, size_t len);
```

  - Obtiene una dirección IP (binaria) a partir de notación entendible

```
int inet_pton(int domain, const char *src_str,
              void *addrptr);
```

Ejemplo: Manejo de direcciones

direcciones.c

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int  main(int argc, char **argv){
        struct in_addr in;

        if (argc != 2) {
                printf("Uso: %s <decimal-punto>\n", argv[0]);
                exit(0);
        }
        if (inet_aton(argv[1], &in) == 0) {
                printf("Error en la dirección\n");
                exit(0);
        }

        printf("La dirección es %s\n", inet_ntoa(in));

        exit(0);
}
```

Obtener la dirección de host (IPv4)

- Obtiene la información de un host a partir de una dirección en formato dominio-punto (`getaddrinfo` para IPv4 e IPv6)

```
struct hostent *gethostbyname(char *str);
```

Argumentos:

str es el nombre de la máquina

- Obtiene la información de un host a partir de una dirección IP (`getaddrinfo` para IPv4 e IPv6)

```
struct hostent *gethostbyaddr(const void *addr,
                              int len,
                              int type);
```

Argumentos:

addr es un puntero a una estructura de tipo *struct in_addr*

len es el tamaño de la estructura

type es AF_INET

Estructura *struct hostent*

```
struct    hostent
{
          char          *h_name ;
          char          **h_aliases ;
          int           h_addrtype ;
          int           h_length ;
          char          **h_addr_list ;
} ;
```

Ejemplo: Conversión dominio-punto a decimal-punto

dns.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <netdb.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int main(int argc, char **argv) {
            struct hostent *hp;
            struct in_addr in;

            hp = gethostbyname("www.uc3m.es");
            if (hp == NULL) {
               printf("Error en gethostbyname\n");     exit(0);
            }
            memcpy(&in.s_addr,*(hp->h_addr_list),sizeof(in.s_addr));
            printf("%s es %s\n", hp->h_name, inet_ntoa(in));
           return 0;
}
```

Ejemplo: Conversión dominio-punto a decimal-punto

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <netdb.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int main(int argc, char **argv) {
            struct hostent *hp;
            struct in_addr in;

            hp = gethostbyname("www.uc3m.es");
            if (hp == NULL) {
               printf("Error en gethostbyname\n");      exit(0);
            }
            memcpy(&in.s_addr,*(hp->h_addr_list),sizeof(in.s_addr));
            printf("%s es %s\n", hp->h_name, inet_ntoa(in));
           return 0;
}
```

```
struct    hostent
{
          char          *h_name ;
          char          **h_aliases ;
          int           h_addrtype ;
          int           h_length ;
          char          **h_addr_list ;
} ;
```

Ejemplo: conversión decimal-punto a dominio-punto

obtener-dominio.c

```
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(int argc, const  char **argv)  {
     struct in_addr  addr;          struct hostent *hp;
     char **p;                      struct in_addr in;
     char **q;                      int err;


     if (argc != 2) {
        printf("USO: %s Direccion-IP\n", argv[0]);
        return (1);
     }
     err = inet_aton(argv[1], &addr);

     if (err == 0) {
         printf("Direccion IP en formato a.b.c.d\n");
         return (2);
     }
```

Ejemplo: conversión decimal-punto a dominio-punto (cont)

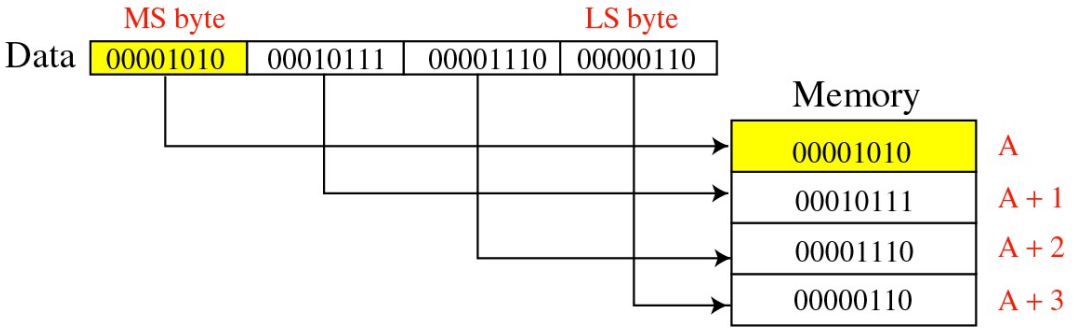
obtener-dominio.c

```
      hp=gethostbyaddr((char *) &addr, sizeof (addr), AF_INET);

      if (hp == NULL) {
          printf("Error en ....\n");
          return (3);
      }
      for (p = hp->h_addr_list; *p!=0;  p++){
         memcpy(&in.s_addr, *p, sizeof(in.s_addr));
         printf("%s es \t%s \n", inet_ntoa(in), hp->h_name);
         for (q=hp->h_aliases;  *q != 0; q++)
               printf("%s\n", *q);
      }

      return(0);
}
```

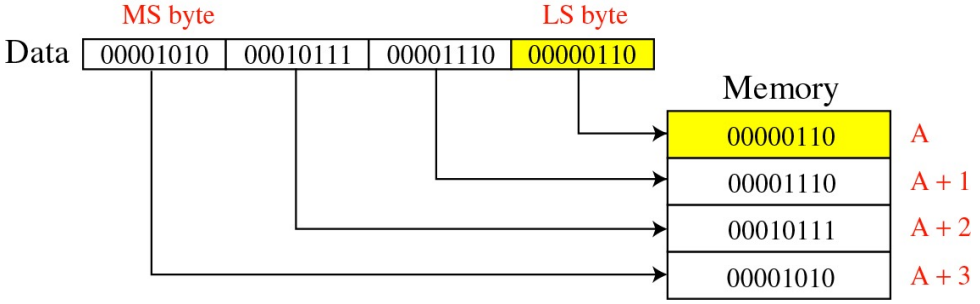
```
struct    hostent
{
          char          *h_name ;
          char          **h_aliases ;
          int           h_addrtype ;
          int           h_length ;
          char          **h_addr_list ;
} ;
```

Orden de los bytes

- Big-endian es el estándar para el ordenamiento de los bytes usado en TCP/IP
  - También llamado Network byte order
- ***Big-endian***

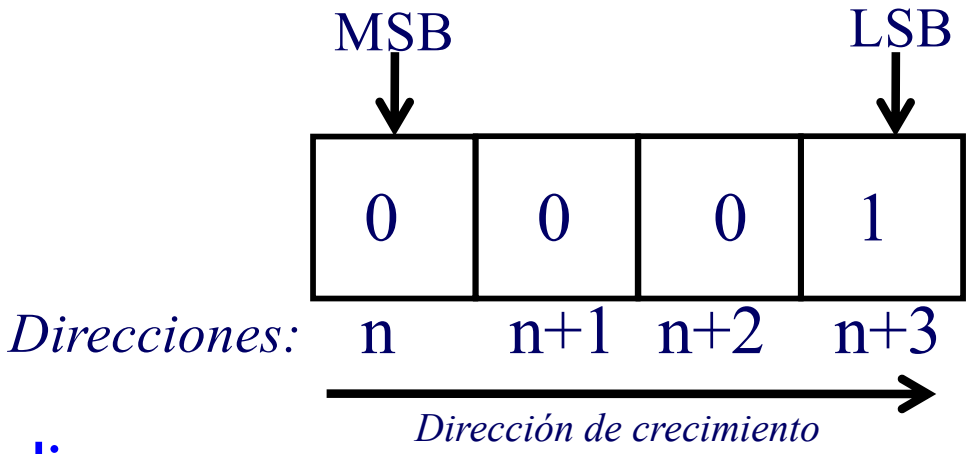


- *Little-endian*

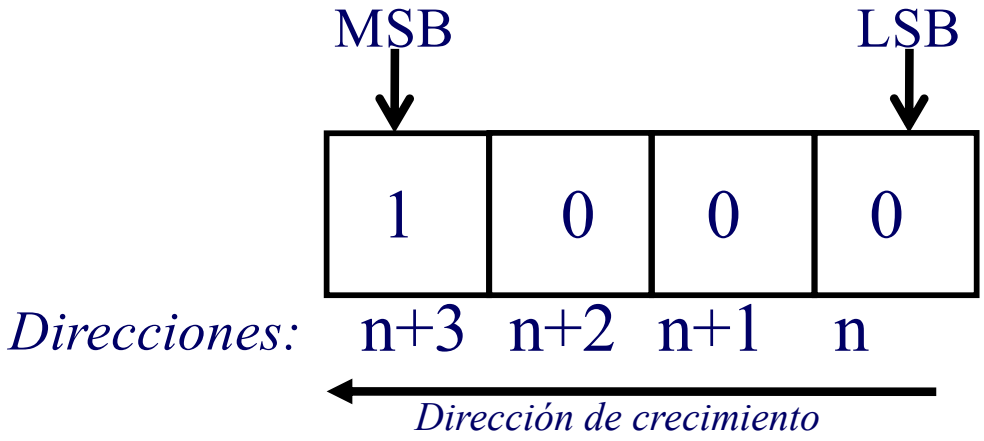


Ejemplo: Representación del 1

- Big Endian

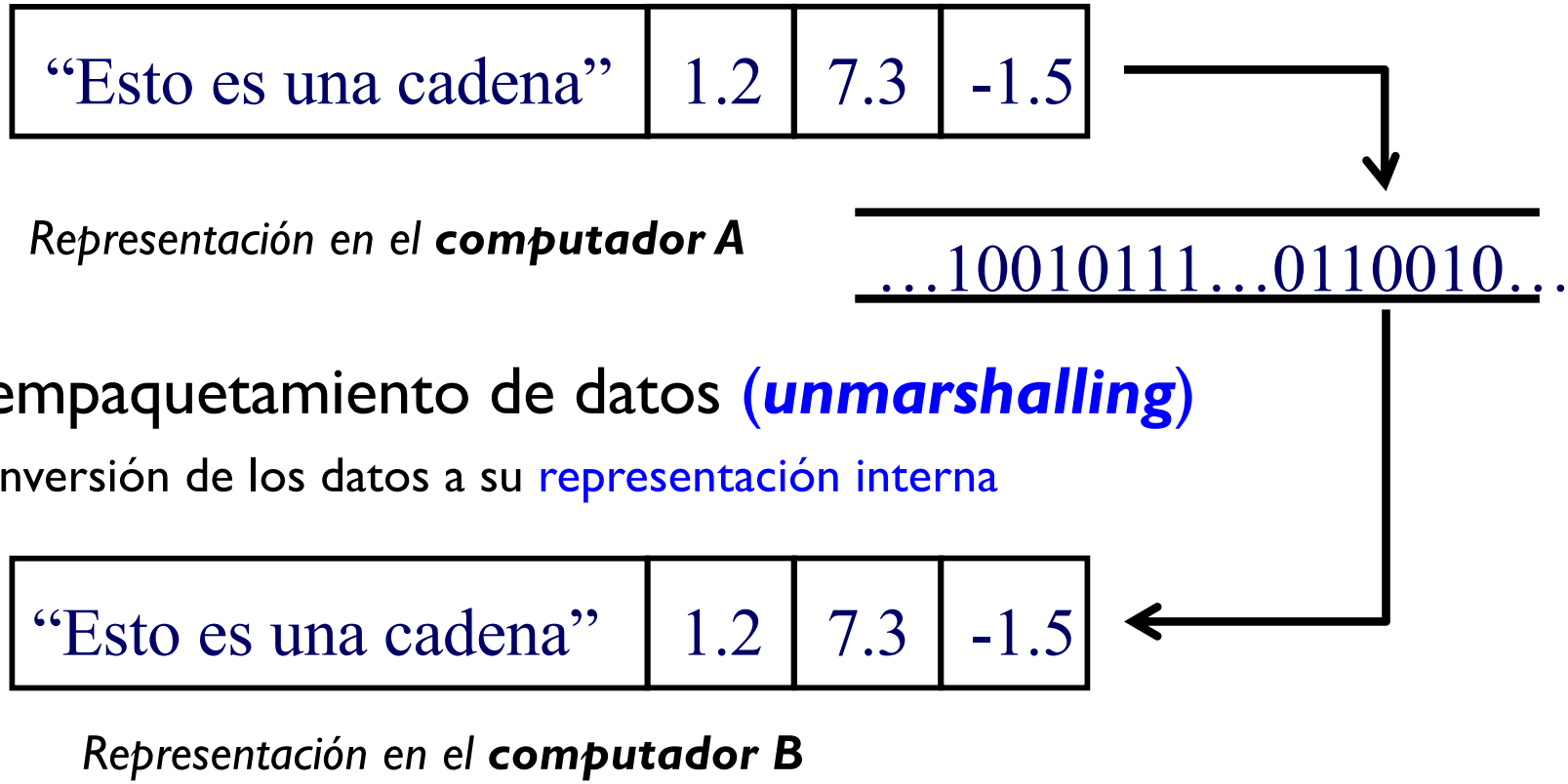


- Little Endian



Representación de datos

- Empaquetamiento de datos (***marshalling***):
  - Serialización de las estructuras de datos y conversión de los valores de los datos a su representación externa



- Desempaquetamiento de datos (***unmarshalling***)
  - Conversión de los datos a su representación interna

Funciones de transformación: *network* ←→ *host*

- En computadores que **no** utilicen Big-endian es necesario emplear funciones para traducir números entre el formato que utiliza TCP/IP (Big-endian) y el empleado por el propio computador (Little-endian) :
  - Host (Little-Endian) → Network (Big-Endian)

    Para traducir un número de 32/16 bits representado en el formato del computador al formato de red (TCP/IP):

    ```
    u_long  htonl(u_long  hostlong)
    u_short htons(u_short hostshort)
    ```

  - Network (Big-Endian) → host (Little-Endian)

    Para traducir un número de 32/16 bits representado en el formato de red (TCP/IP) al formato del computador

    ```
    u_long  ntohl(u_long  netlong)
    u_short ntohs(u_short netshort)
    ```

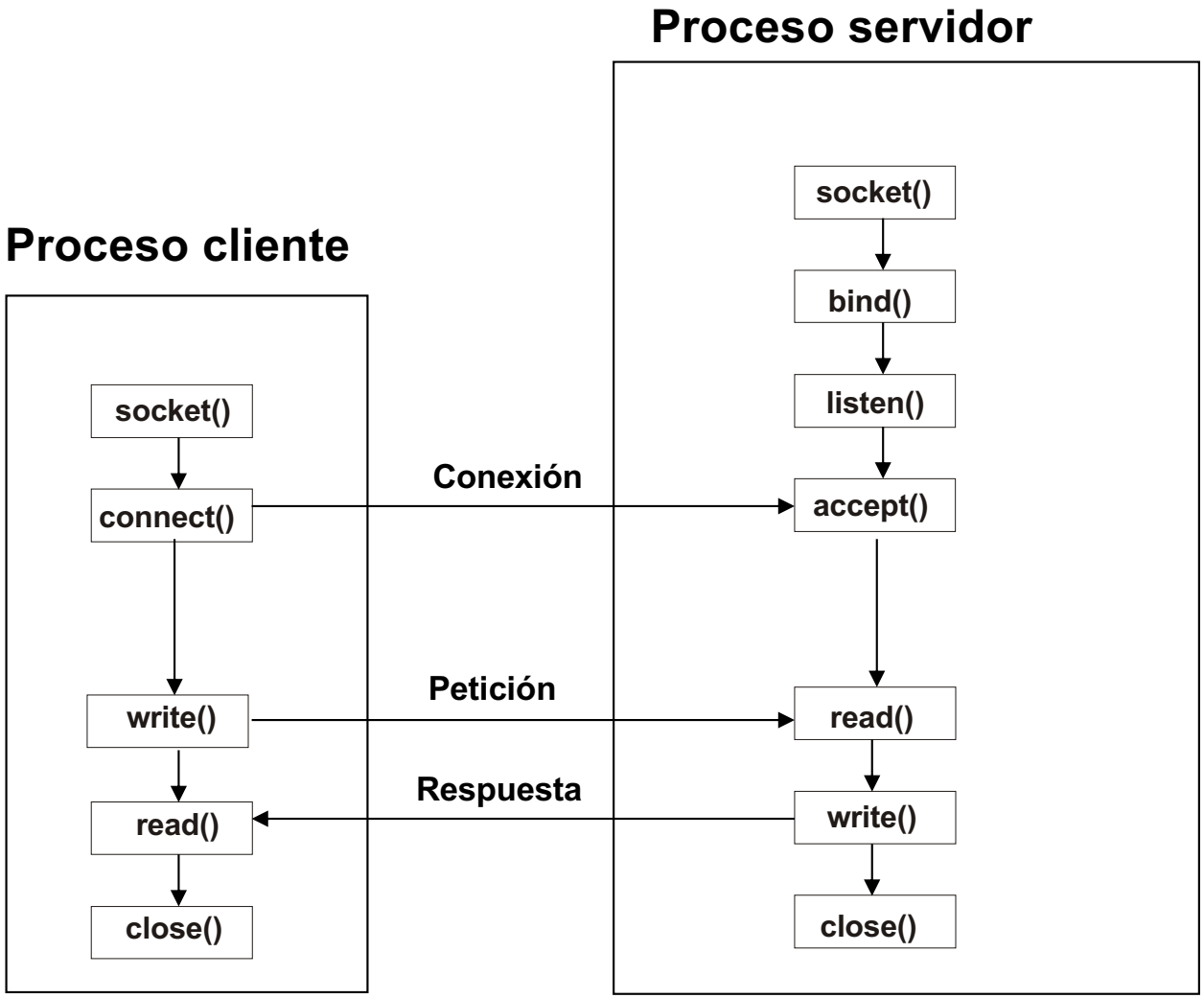
  - `#include <arpa/inet.h>`

Modelos de comunicación

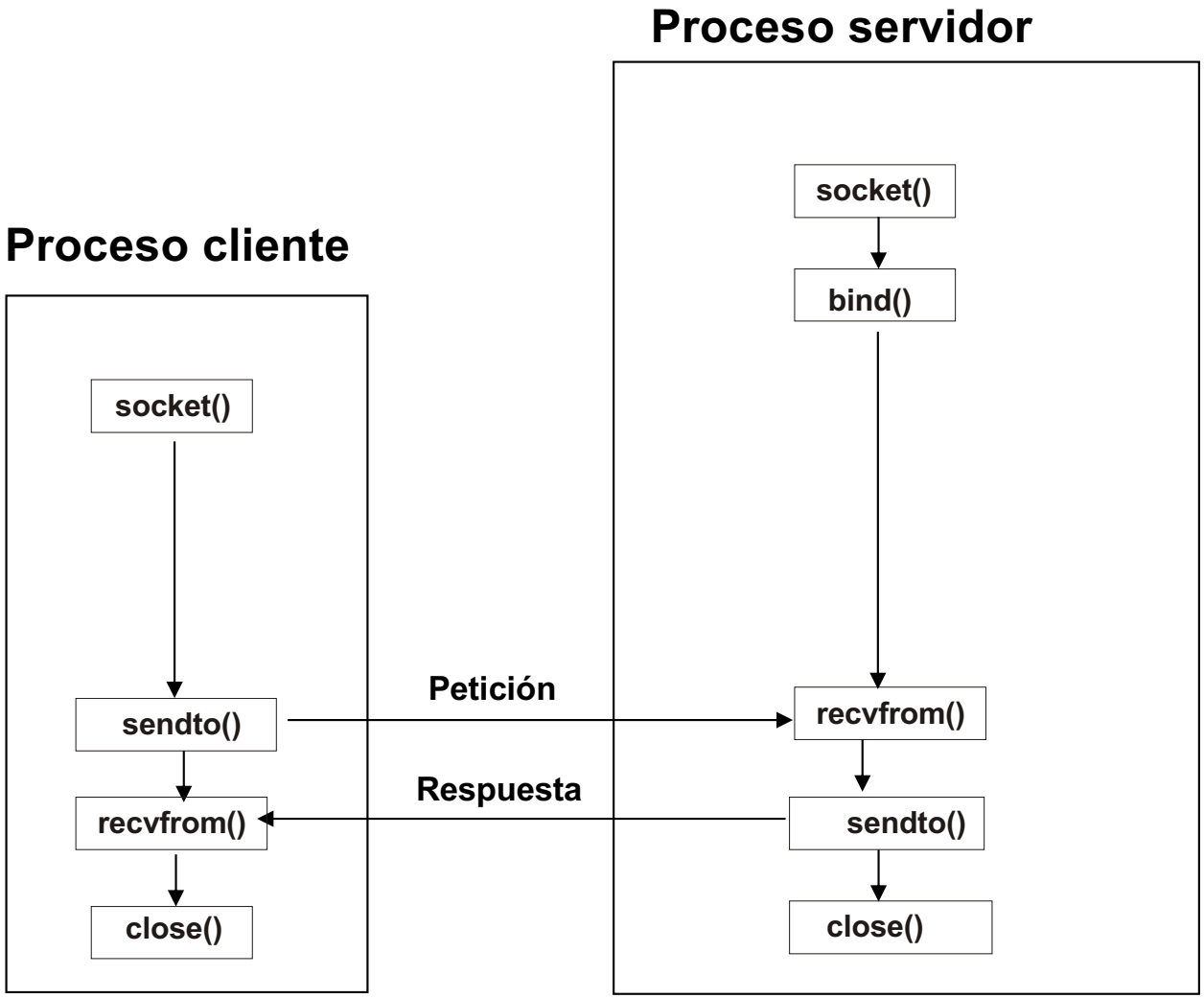
- Sockets **stream** (SOCK_STREAM)
  - Orientado a conexión
  - **TCP**

- Sockets **datagrama** (SOCK_DGRAM)
  - No orientado a conexión
  - **UDP**

Modelo de comunicación con sockets *stream*



Modelo de comunicación con sockets datagrama



Servicios POSIX para utilizar sockets

- Creación de un socket (**socket**)
- Asignación de direcciones (**bind**)
- Preparar para aceptar conexiones (**listen**)
- Aceptar una conexión (**accept**)
- Solicitud de conexión (**connect**)
- Obtener la dirección de un socket
- Transferencia de datos
  - *Streams*
  - Datagramas
- Cerrar un socket (**close**)

Creación de un socket *(socket)*

- Crear un socket:

```
#include <sys/socket.h>
int socket(int dominio, int tipo, int protocolo)
```

Argumentos:

dominio	AF_UNIX, AF_INET
tipo	SOCK_STREAM, SOCK_DGRAM
protocolo	dependiente del dominio y tipo

  - 0 en el caso general
  - Especificados en */etc/protocols*

devuelve:

si éxito, un **descriptor de socket**
si error, **-1**

- El socket creado **no** tiene dirección asignada

Ejemplo: Crear un socket TCP

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>

int main()
{
        /* VARIABLES DEL PROGRAMA */
        int sockfd;

        if ((sockfd = socket(AF_INET, SOCK_STREAM, 0))==-1) {
                printf("Error en la creación del socket");
                return(-1);
        }
        /* CONTINUACIÓN DEL PROGRAMA */

        return(0);
}
```

Asignación de direcciones *(bind)*

- Asignar dirección a un socket:

```
#include <sys/socket.h>

int bind(int sd,   const struct sockaddr *addr,
                                    socklen_t addrlen)
```

Argumentos:

sd	descriptor devuelto por socket
addr	dirección a asignar
addrlen	longitud de la dirección/tamaño de la estructura *sockaddr*

devuelve **–1** si error y **0** si se ejecutó con éxito

- Direcciones en dominio AF_INET (*struct sockaddr_in*)
  - Host: una dirección local IP
    - INADDR_ANY: elige cualquiera de la máquina
  - Puertos:
    - 65535 puertos disponibles, de los cuales 0..1023 están reservados
    - Si 0, el sistema elige uno

- Si no se asigna dirección al socket (típico en **clientes**)
  - Se le asigna automáticamente (puerto efímero) en la primera utilización (*connect* o *sendto*)

Obtener la dirección de un socket

- Obtener la dirección a partir del descriptor:

```
int getsockname(int sd,
                struct sockaddr *dir, socklen_t *len)
```

sd	descriptor devuelto por socket
dir	dirección del socket devuelta
len	parámetro valor-resultado

- Obtener la dirección del socket en el otro extremo de la conexión:

```
int getpeername(int sd,
                struct sockaddr *dir, socklen_t *len)
```

sd	descriptor devuelto por socket
dir	dirección del socket remoto
len	parámetro valor-resultado

Preparar para aceptar conexiones (*listen*)

- Habilita un socket para recibir conexiones en el servidor **TCP**
- Realizada en el servidor *stream* después de *socket* y *bind*

```
#include <sys/types.h>
#include <sys/socket.h>
int listen(int sd, int baklog)
```

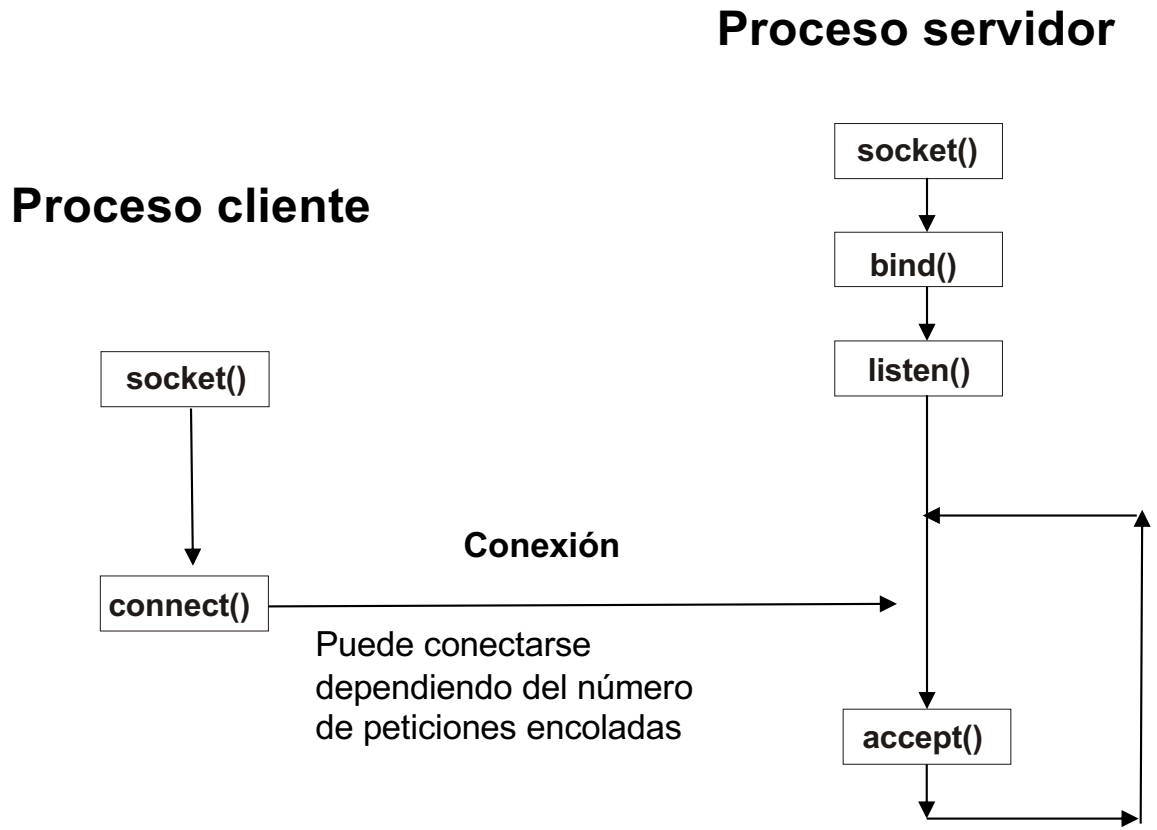
Argumentos:

sd: descriptor devuelto por socket

backlog: número máximo de peticiones que se encolarán.
Máximo definido en SOMAXCONN (`<sys/socket.h>`).

devuelve **–1** si error y **0** si se ejecutó con éxito

Preparar para aceptar conexiones (*listen*)



Aceptar una conexión *(accept)*

- Realizada en el servidor *stream* después de *socket*, *bind* y *listen*

- Bloquea al servidor hasta que se produce la conexión; el servidor obtiene:
  - La dirección del socket del cliente
  - Un nuevo descriptor que queda conectado al socket del cliente

- Después de la conexión quedan activos dos sockets en el servidor:
  - El **original** para seguir aceptando nuevas conexiones
  - El **nuevo** para enviar/recibir datos por la conexión establecida

Aceptar una conexión *(accept)*

```
#include <sys/socket.h>

int accept(int sd, struct sockaddr *dir, socklen_t *len)
```

Argumentos:

sd	descriptor devuelto por socket
dir	dirección del socket del cliente
len	parámetro valor-resultado: 1) Antes de la llamada: tamaño de `dir` 2) Después de la llamada: tamaño de la dirección del cliente que se devuelve en `dir`

- Devuelve **un nuevo descriptor** de socket asociado a la conexión
  - **-1** en caso de error

Solicitud de conexión *(connect)*

- Realizada en el cliente (TCP) para conectarse al servidor

```
#include <sys/socket.h>
int connect(int sd, const struct sockaddr *dir,
                                socklen_t   len)
```

Argumentos:

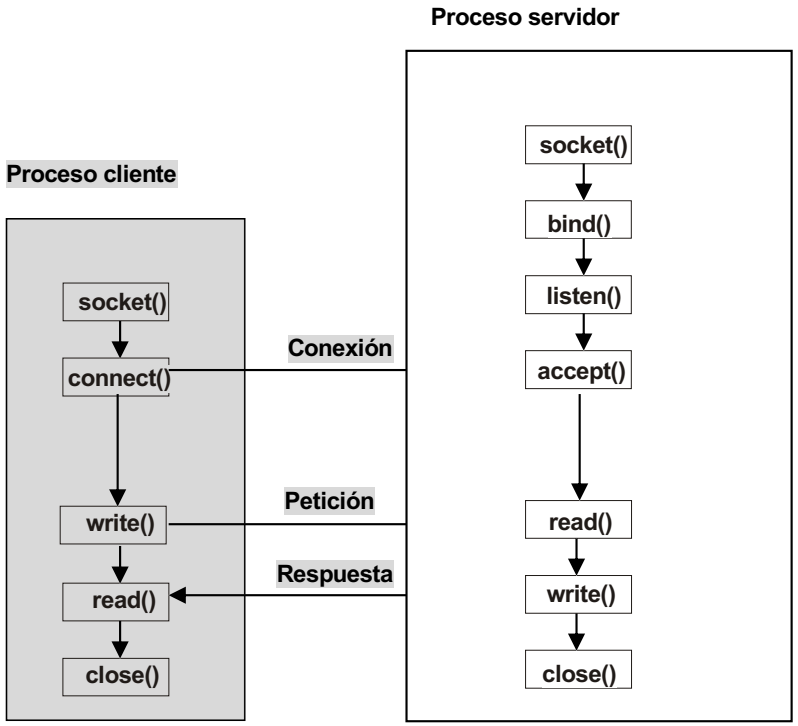
sd	descriptor devuelto por socket
dir	dirección del socket remoto
len	longitud de la dirección del socket remoto

devuelve **–1** si error y **0** si se ejecutó con éxito

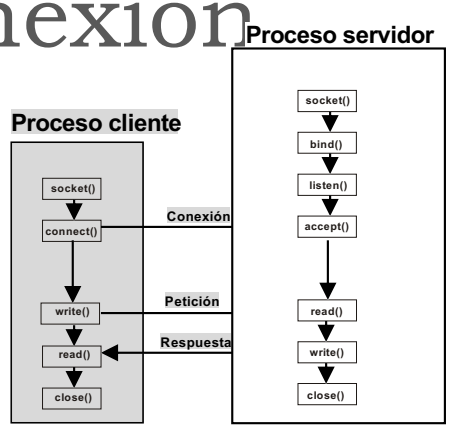
- Si el socket no tiene dirección asignada, se le asigna una automáticamente

Ejemplo: Establecimiento de conexión TCP (clientes)

- Cliente que establece una conexión con un host y puerto pasados por parámetro

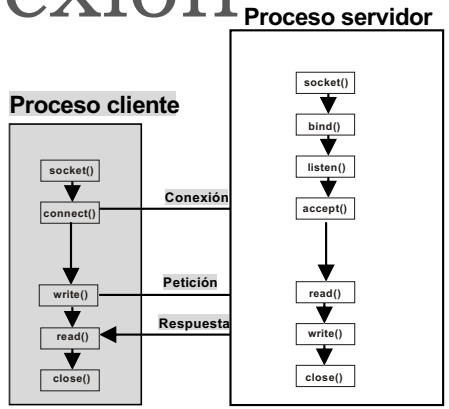


Ejemplo: Establecimiento de conexión TCP (clientes)



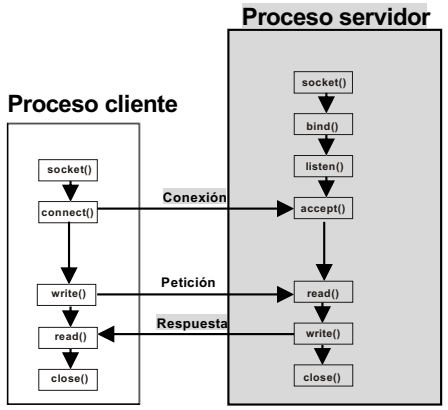

```
#include <stdio.h>
#include <stdlib.h>
#include <netdb.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>

int main(int argc, char **argv) {
        int sd;      short puerto;
        struct sockaddr_in server_addr;
        struct hostent *hp;
        if (argc != 3){
                printf("Uso: cliente <host> <puerto>\n");
                return(0);
        }
        puerto = (short) atoi(argv[2]);
```

Ejemplo: Establecimiento de conexión TCP (clientes)-cont




```
sd = socket(AF_INET, SOCK_STREAM, 0);
bzero((char *)&server_addr, sizeof(server_addr));
hp = gethostbyname (argv[1]);

memcpy (&(server_addr.sin_addr), hp->h_addr, hp->h_length);
server_addr.sin_family  = AF_INET;
server_addr.sin_port    = htons(puerto);
// se establece la conexión
if(connect(sd, (struct sockaddr *) &server_addr,
                          sizeof(server_addr)) < 0) {
     printf("Error en la conexión\n");
     return(0);
}
/* se usa sd para enviar o recibir datos del servidor*/
close(sd);
return(0);
}
```

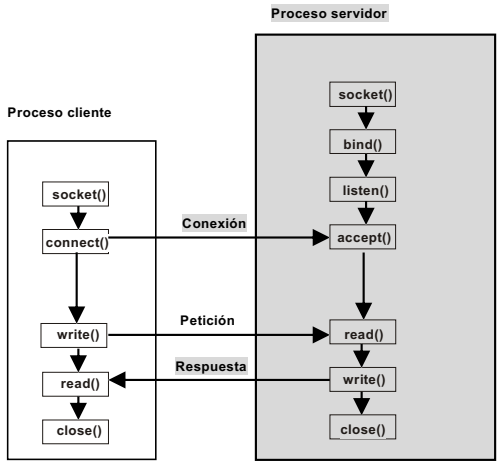
Ejemplo: Establecimiento de conexión TCP (servidores)




```
#include <stdio.h>
#include <string.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>

int main() {
     int sd, newsd;
     socklen_t size;
     struct sockaddr_in server_addr, client_addr;

    if ((sd=socket(AF_INET, SOCK_STREAM, 0))==-1){
          printf("Error en la creación del socket");
          return(-1);
    }
```

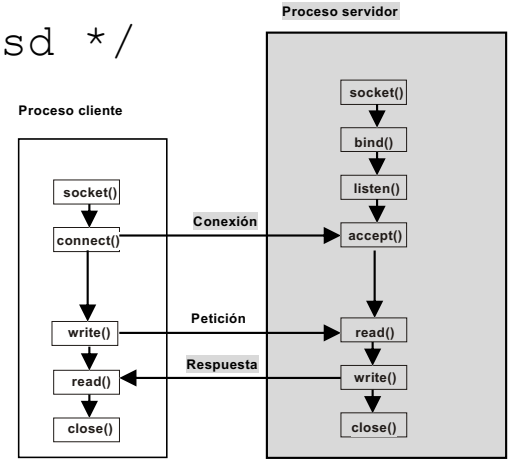
Ejemplo: Establecimiento de conexión TCP (servidores) - cont

```
bzero((char *)&server_addr, sizeof(server_addr));
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(4200);
server_addr.sin_addr.s_addr = INADDR_ANY;

/* bind */
 if (bind(sd,(struct sockaddr *)&server_addr,
                          sizeof(server_addr)) <0) {
       printf("Error en el bind\n");
       return(-1);
 }
```



Ejemplo: Establecimiento de conexión TCP (servidores) - cont

```
        listen(sd,SOMAXCONN);
        for (;;) {
          newsd=accept (sd, (struct sockaddr *) &client_addr, &size);
          if (newsd < 0) {
                printf("Error en el accept");
                return(-1);
          }
          /* transferir datos sobre newsd */
           /* procesar la petición utilizando newsd */
         close(newsd);
      }
        close(sd);
} /* fin main */
```



Obtener la IP y el puerto del cliente

```
newsd=accept (sd, (struct sockaddr *) &client_addr, &size);
if (newsd < 0) {
     printf("Error en el accept");
     return(-1);
}

// La estructura client_addr almacena la IP y el puerto del proceso
// cliente que se conecta con el servidor
printf("conexión aceptada de IP: %s   Puerto: %d\n",
             inet_ntoa(client_addr.sin_addr),
              ntohs(client_addr.sin_port));
```

Transferencia de datos con *streams*

- Una vez establecida la conexión usando sockets *stream*, ambos extremos puede transferir datos
- Envío:

```
#include <unistd.h>
ssize_t write(int sd, const char *mem, size_t long);
```

  - Devuelve el número de bytes enviados (0, no se envío nada) o -1 en caso de error

- Pueden **no** transferirse todos los datos
  - Es importante comprobar siempre el valor que devuelven estas llamadas

Transferencia de datos con *streams*

- La recepción es una llamada bloqueante (síncrona)
- Recepción:

```
#include <unistd.h>
int read(int sd, char *mem, int long);
```

  - Devuelve el **número de bytes** recibidos o **–1** si error
  - Devuelve 0 si se ha cerrado la conexión en el otro extremo.

- Pueden **no** transferirse todos los datos
  - Es importante comprobar siempre el valor que devuelven estas llamadas

Ejemplo: Transferencia de datos con *streams*

- Función que envía un bloque de datos con reintentos:

```
int sendMessage(int socket, char *buffer, int len)
{
      int r;
      int l = len;
      do {
            r = write(socket, buffer, l);
            l = l - r;
            buffer = buffer + r;
      } while ((l>0) && (r>=0));

      if (r < 0)
            return (-1);   /* fallo */
      else
            return(0);    /* se ha enviado longitud */
}
```

Ejemplo: Transferencia de datos con *streams*

- Función que recibe un bloque de datos con reintentos:

```
int recvMessage(int socket, char *buffer, int len)
{
      int r;
      int l = len;
      do {
            r = read(socket, buffer, l);
            l = l - r;
            buffer = buffer + r;
      } while ((l>0) && (r>=0));

      if (r < 0)
            return (-1);   /* fallo */
      else
            return(0);/* se ha recibido longitud bytes */
}
```

Transferencia de datos con datagramas

- **No** hay conexión
- Para usar un socket para transferir datos basta con:
  - Crearlo (*socket*)
  - Asignarle una dirección (*bind*)
    - Si no se le asigna dirección, lo hará el sistema de manera transparente
- Envío:

```
#include <sys/types.h>
#include <sys/socket.h>
int sendto(int sd, const void *buffer, size_t len, int flags,
           const struct sockaddr *dir, socklent_t addrlen)
```

  - Devuelve el **número de bytes** enviados o **–1** si error
  - Argumentos:

sd	descriptor de socket
buffer	puntero a los datos a enviar
len	la longitud de los datos
flags	opciones de envío (*man sendto*), 0 en la mayoría de los casos
dir	dirección del socket remoto
addrlen	la longitud de la dirección

Transferencia de datos con datagramas

- Recepción:

```
#include <sys/types.h>
#include <sys/socket.h>
int recvfrom(int sd, void *buffer, size_t len,
      int flags, struct sockaddr *dir, socklent_t addrlen):
```

  - Devuelve el **número de bytes** recibidos o **–1** si error
  - Argumentos:

sd	descriptor de socket
buffer	puntero a los datos a enviar
len	la longitud de los datos
flags	opciones de recepción (man *recvfrom*), 0 en la mayoría de los casos
dir	dirección del socket remoto
addrlen	la longitud de la dirección

Cerrar un socket *(close)*

- Se usa close para cerrar ambos tipos de sockets (**stream** y **datagrama**)

```
#include <unistd.h>
int close(int sd);
```

  - Cierra la conexión en ambos sentidos
  - Devuelve **–1** si error

- Se puede cerrar el canal de lectura o escritura:

```
int shutdown(int sd, int modo);
```

sd descriptor devuelto por socket
modo SHUT_RD, SHUT_RW o SHUT_RDWR

  - Devuelve **–1** si error

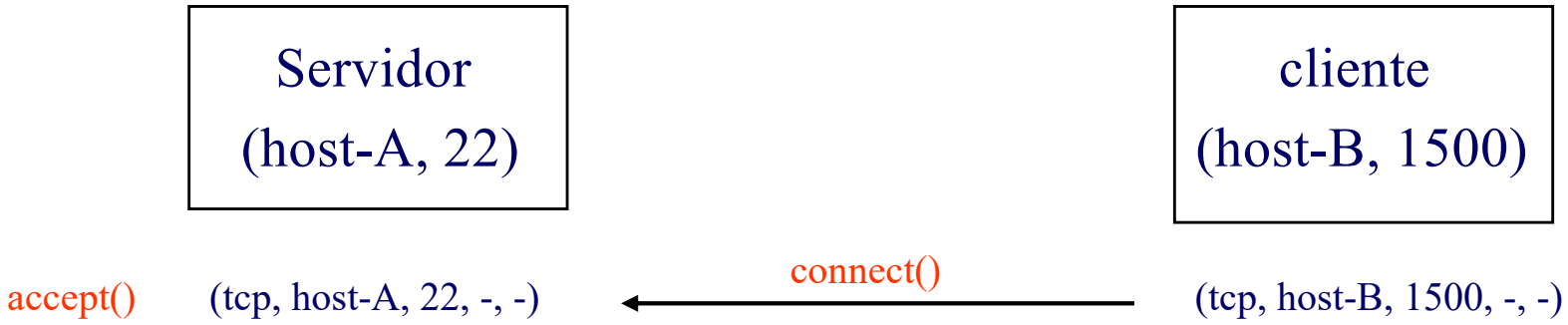
shutdown

- modo:
  - SHUT_RD: cierra el canal de lectura. Posteriores lecturas devuelve 0. Se puede seguir escribiendo datos en el socket
  - SHUT_WR: cierra el canal de escritura. Cuando el extremo de la comunicación haya leído todos los datos pendientes, `read` devolverá 0.
  - SHUT_RDWR: cierra ambos canales.

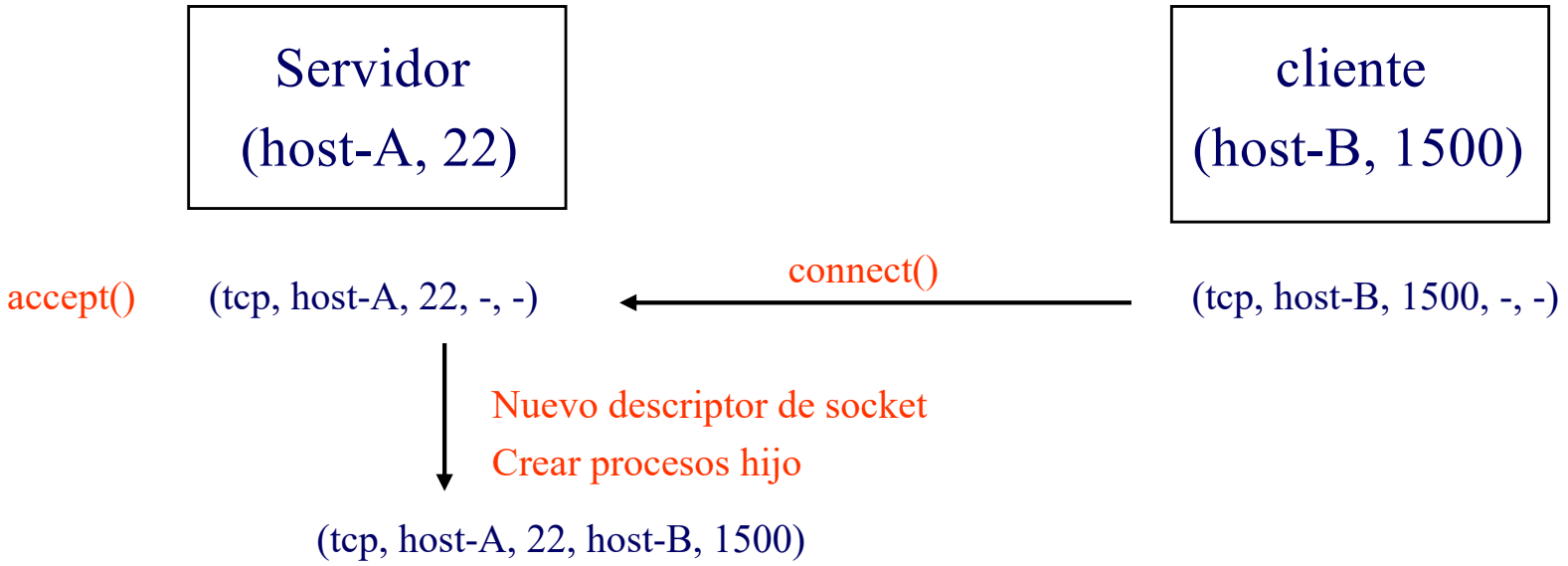
Conexión con TCP

Conexión con TCP

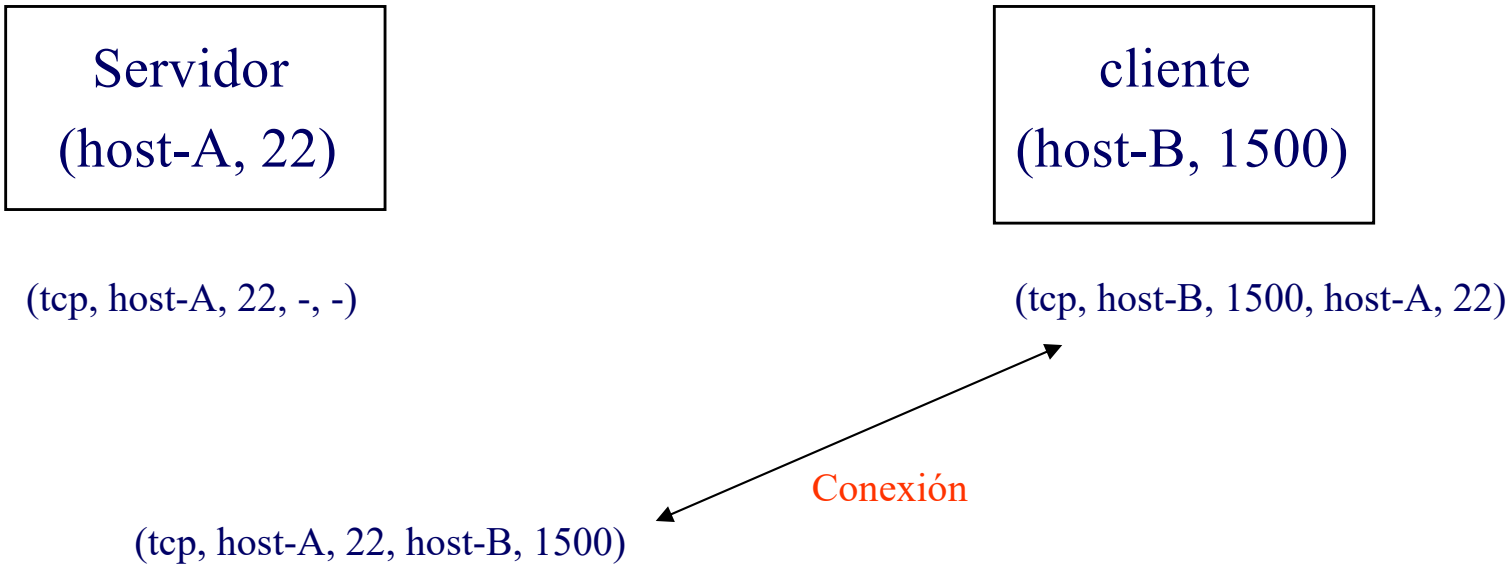
Conexión con TCP



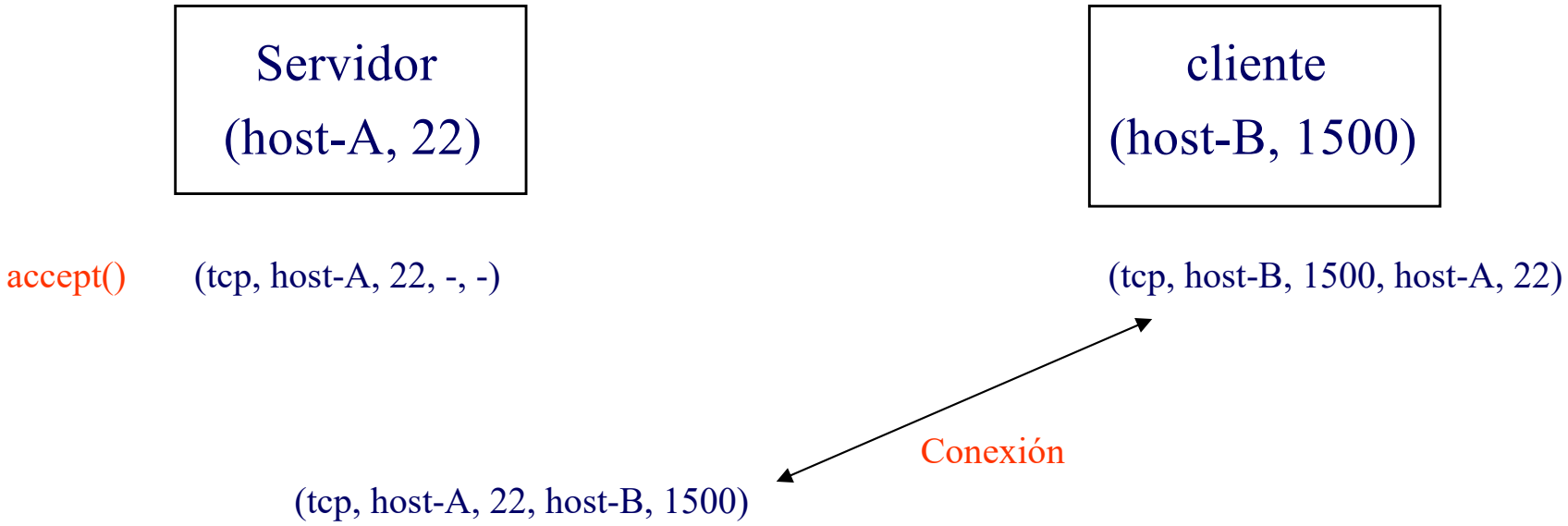
Conexión con TCP



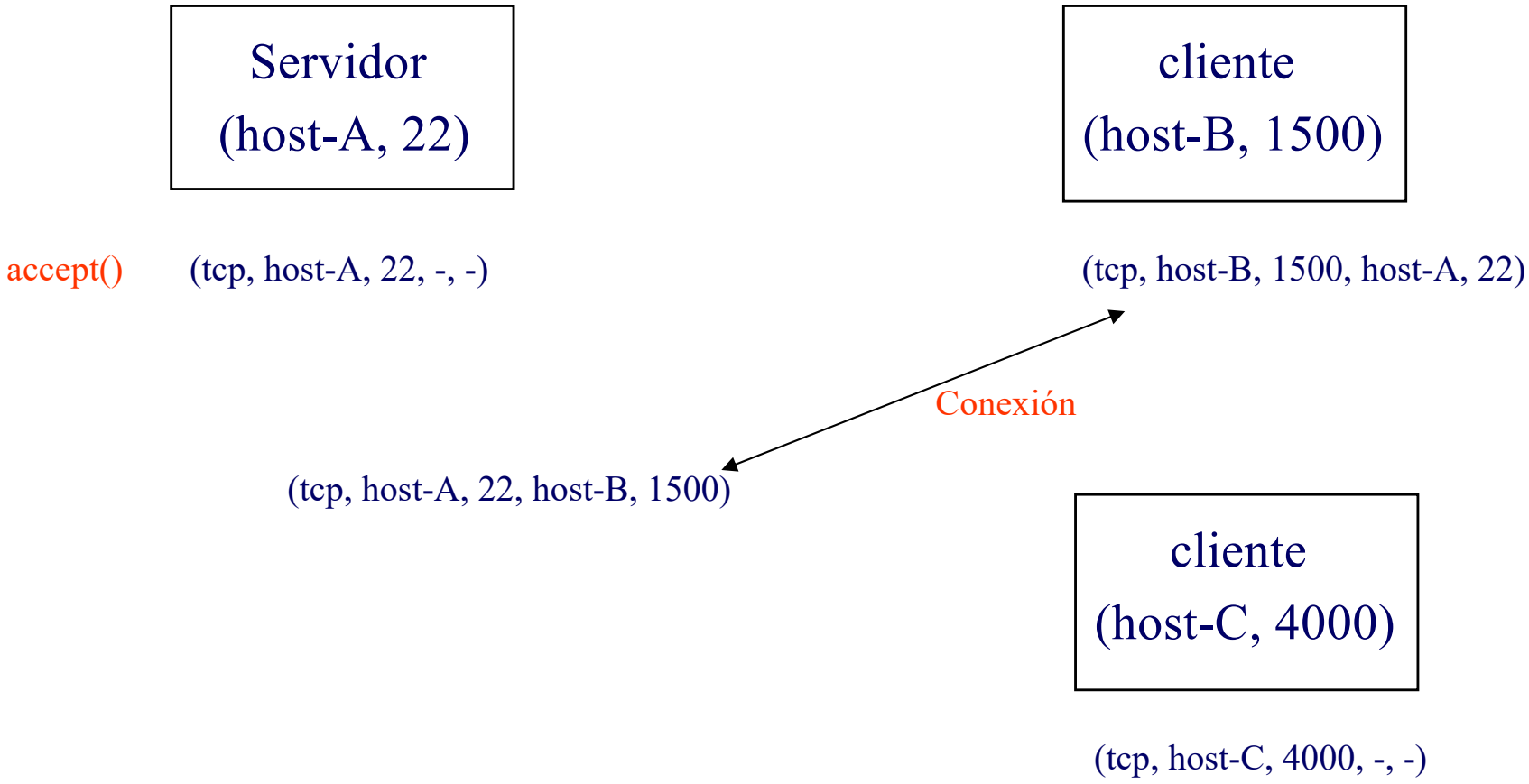
Conexión con TCP



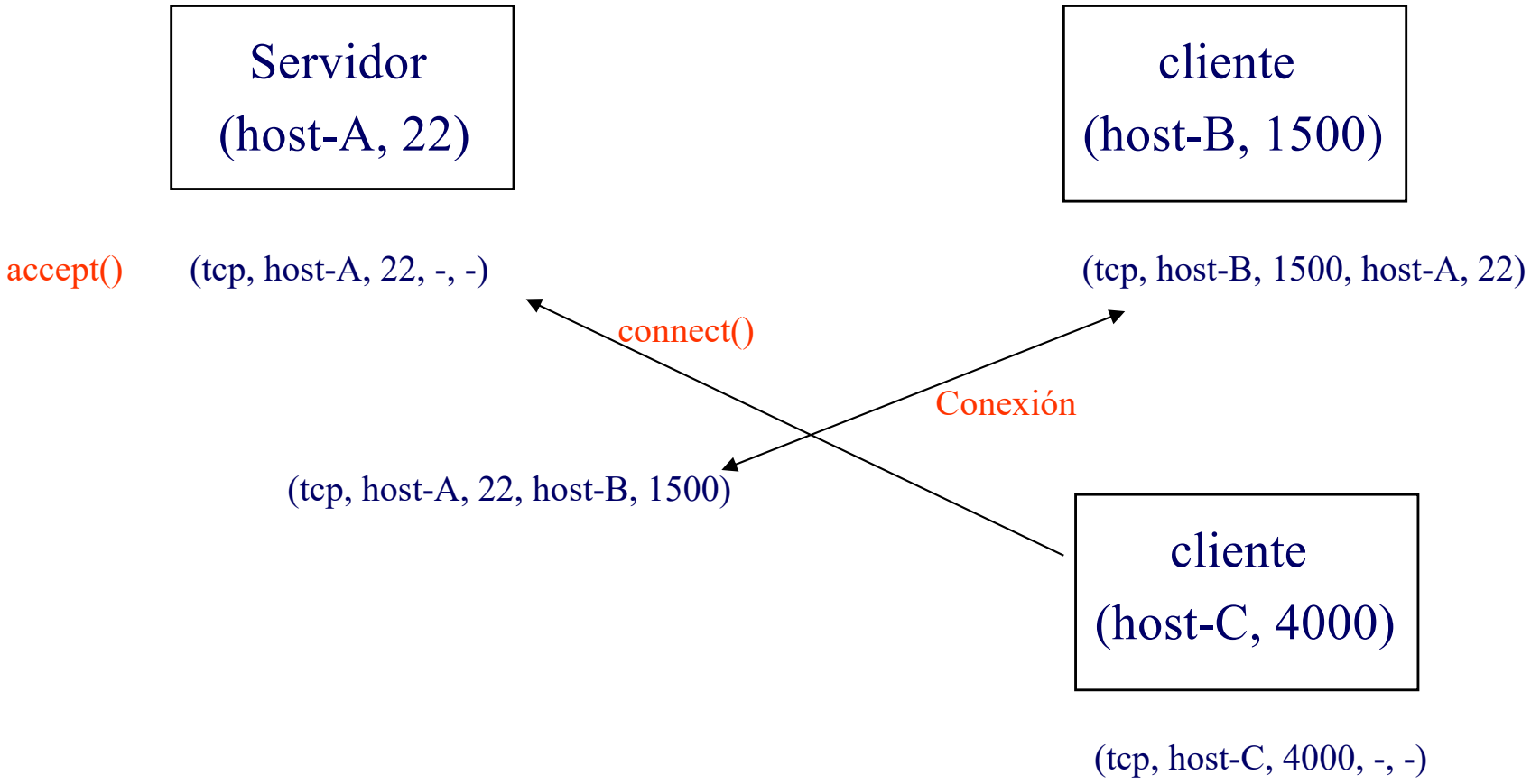
Conexión con TCP



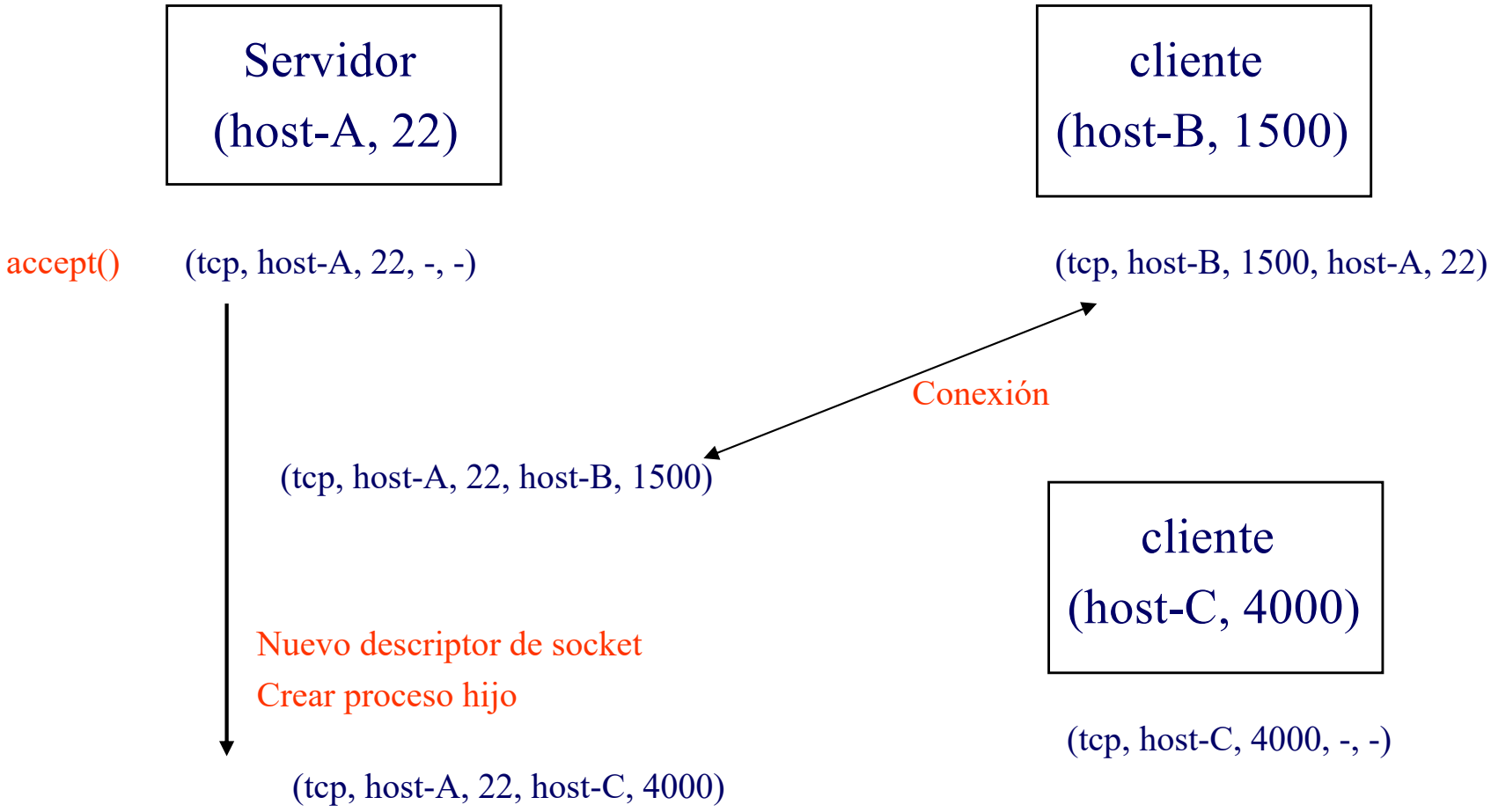
Conexión con TCP



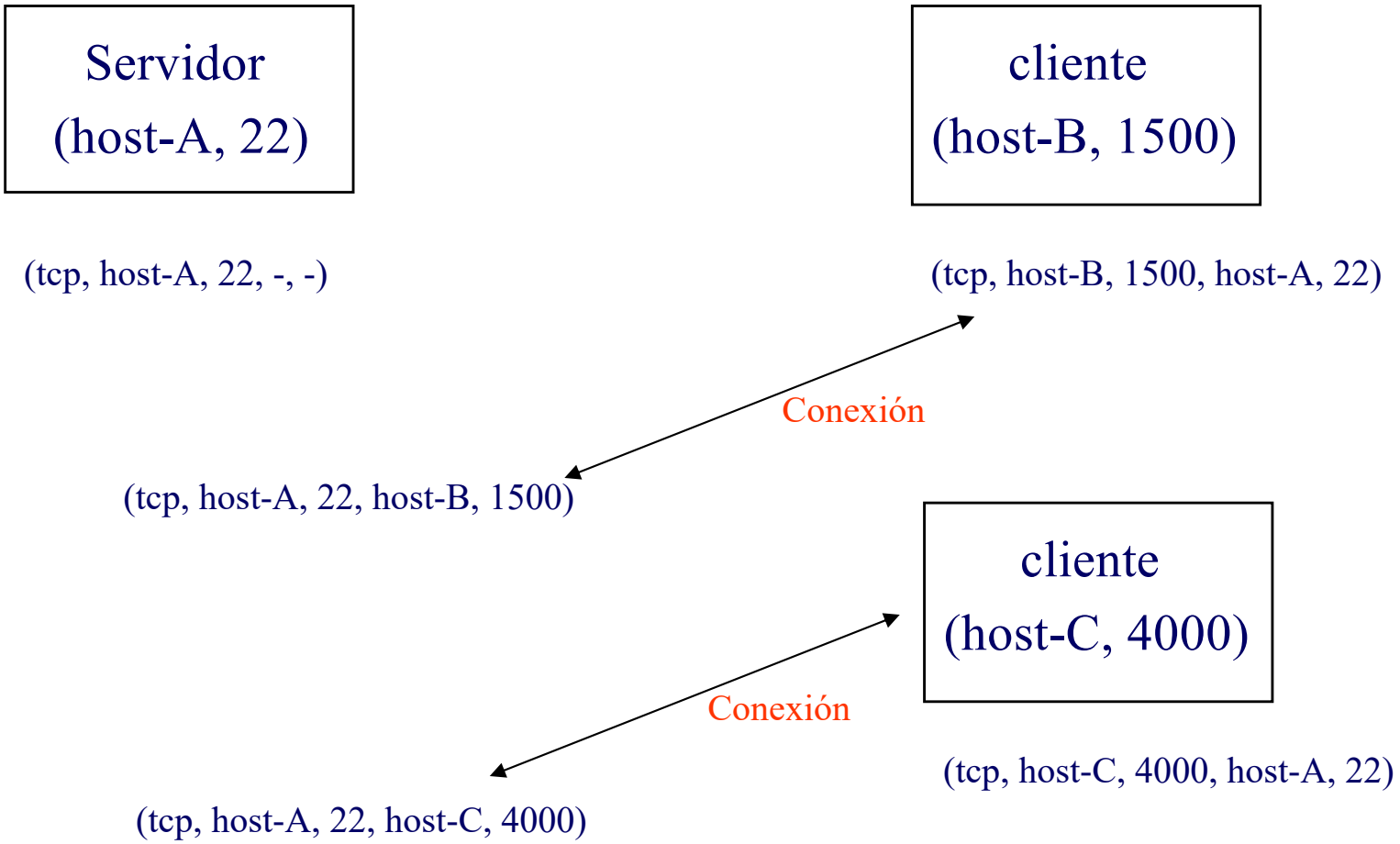
Conexión con TCP



Conexión con TCP



Conexión con TCP



Configuración de opciones

- Existen varios niveles dependiendo del protocolo afectado como parámetro
  - `SOL_SOCKET`: opciones independientes del protocolo
  - `IPPROTO_TCP`: nivel de protocolo TCP
  - `IPPTOTO_IP`: nivel de protocolo IP

- **Consultar opciones** asociadas a un socket
  - `#include <sys/types.h>`
  - `#include <sys/socket.h>`

```
int getsockopt (int sd, int nivel, int opcion,
                void *val, socklen_t *len)
```

- **Modificar las opciones** asociadas a un socket

```
int setsockopt (int sd, int nivel, int opcion,
                const void *val, socklen_t len)
```

- Ejemplo de opción (nivel `SOL_SOCKET`):
  - `SO_REUSEADDR`: permite reutilizar direcciones

¿Por qué utilizar SO_REUSEADDR?

- Evitar el error en el servidor TCP:
  - EADDRINUSE ("Address already in use")
- TCP mantiene las conexiones bloqueadas durante un cierto tiempo (`TIME_WAIT`)
- La conexión ya ha sido cerrada y no puede utilizarse, pero todavía no se han eliminado las tablas internas asociadas por si permanecen todavía segmentos en tránsito en la red

```
int val = 1;
setsockopt(sd, SOL_SOCKET, SO_REUSEADDR,
                    (void*) &val, sizeof(val));
```

SO_RCVBUF, SO_SNDBUF

- Buffer de envío y recepción
- Fijar el tamaño del buffer de envío:

```
int size =  16384;
err = setsockopt(s, SOL_SOCKET, SO_SNDBUF,
     (char *)&size,  sizeof(size));
```

- Conocer el tamaño del buffer de envío:

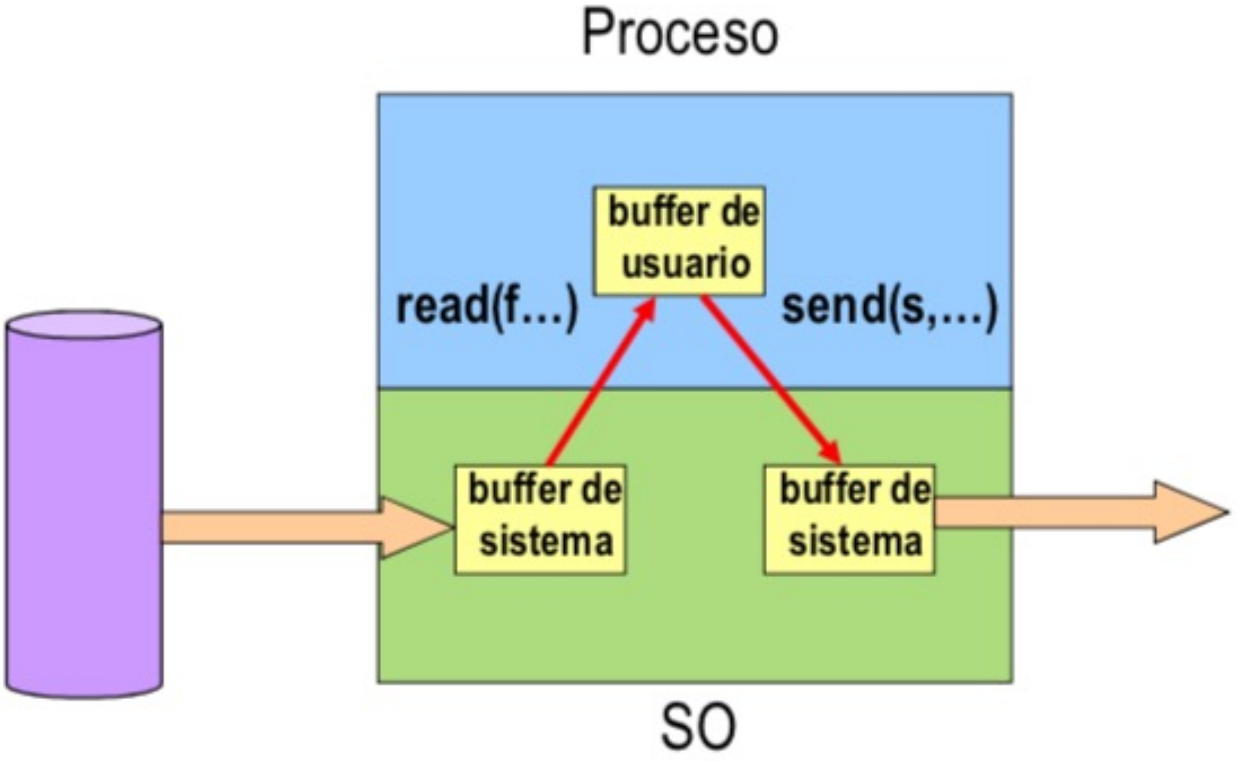
```
int size;
err = getsockopt(s, SOL_SOCKET, SO_SNDBUF,
     (char *)&size,  sizeof(size));
printf("%d\n", size)
```

TCP_NODELAY

- Envío inmediato (sin intentar agrupar mensajes relativamente juntos en el tiempo)

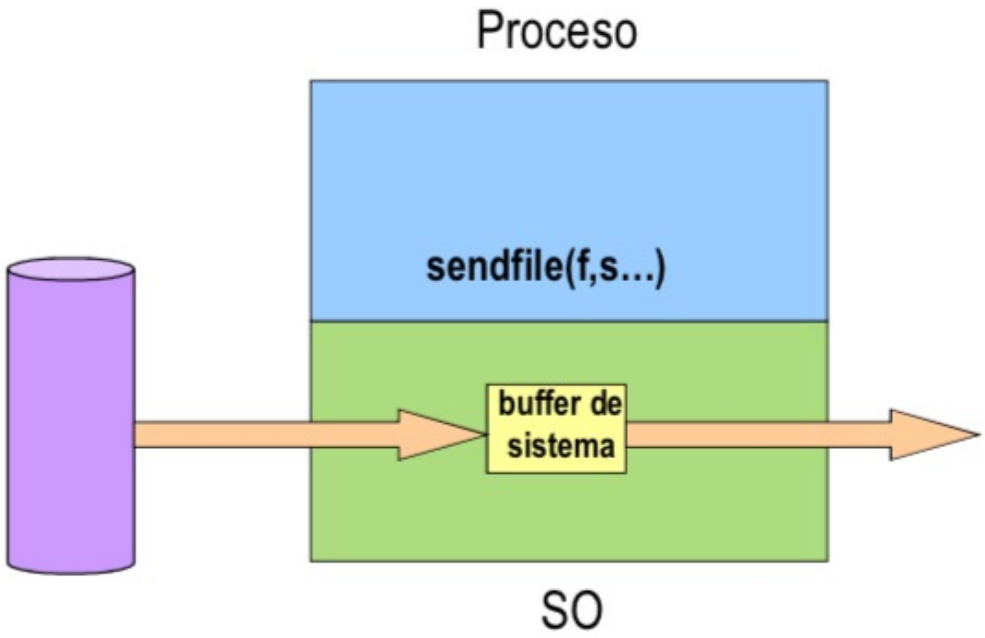
```
int option = 1;
rc = setsockopt(s,    IPPROTO_TCP,
                      TCP_NODELAY,
                      &option,
                      sizeof(option));
```

Envío de un fichero con sockets



Copia de datos entre zonas de memoria

Envío de un fichero sin copias de memoria (zero-copy)



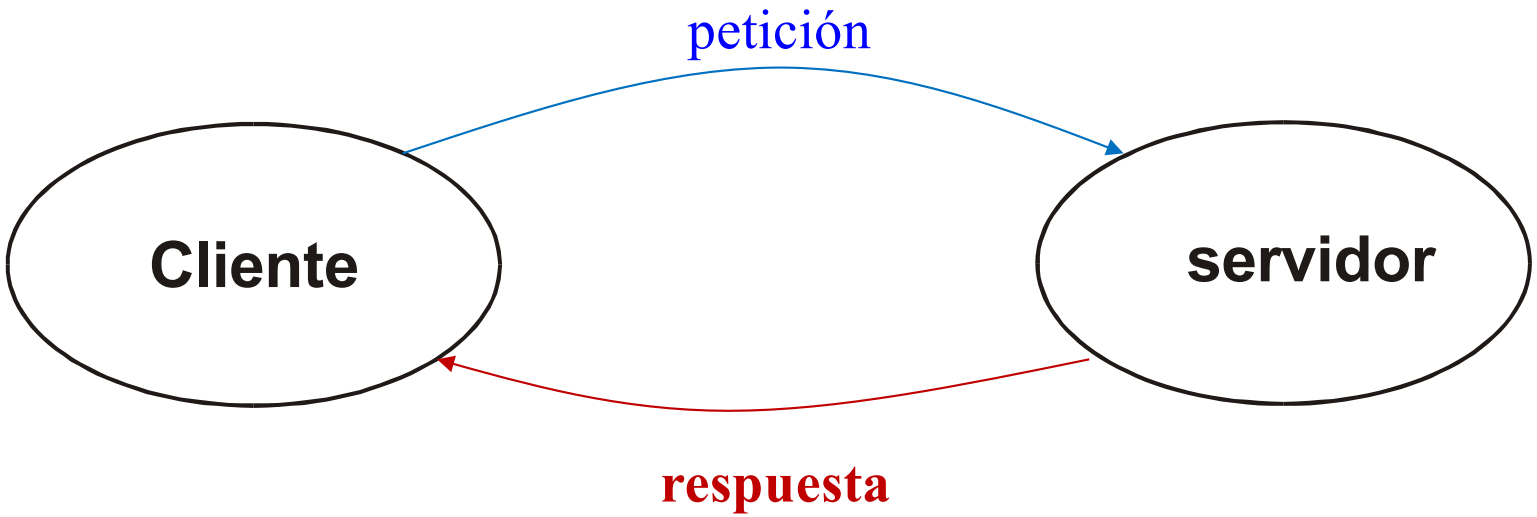
Servicio sendfile en Linux

Comparación de protocolos

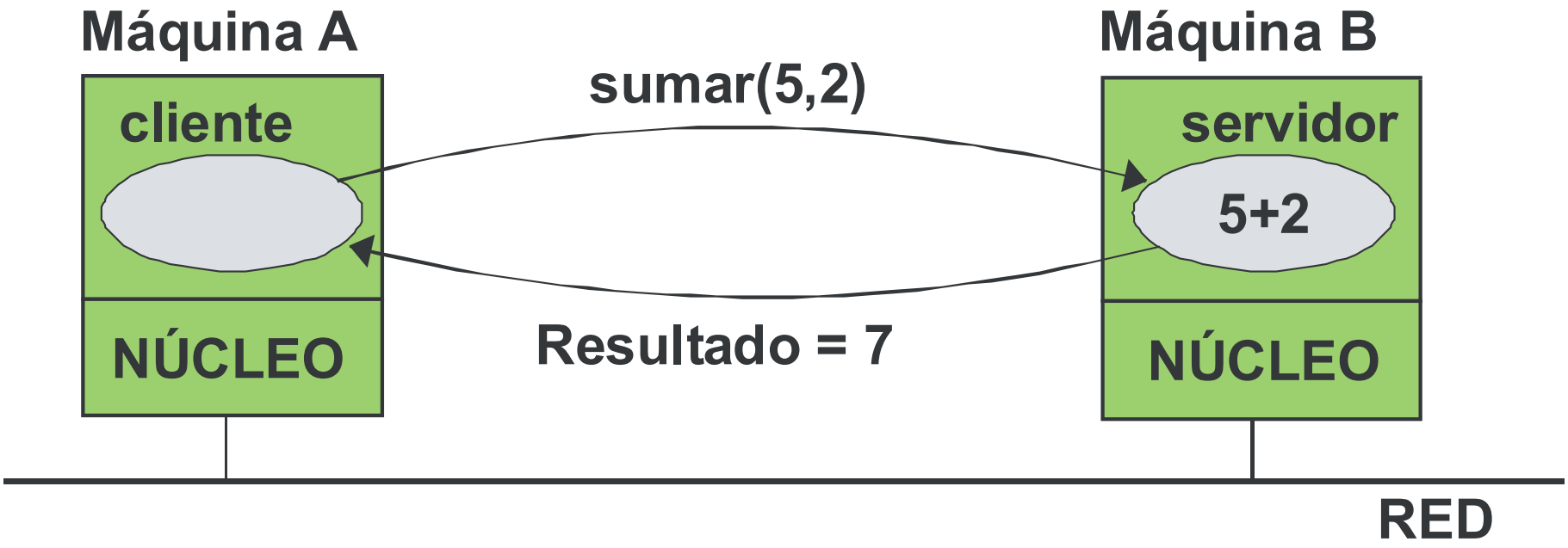
	IP	**UDP**	**TCP**
¿Orientado a conexión?	No	No	Si
¿Límite entre mensajes?	Si	Si	No
¿Ack?	No	No	Si
¿Timeout y retransmisión?	No	No	Si
¿Detección de duplicación?	No	No	Si
¿Secuenciamiento?	No	No	Si
¿Flujo de control?	No	No	Si

Modelo de servidor secuencial

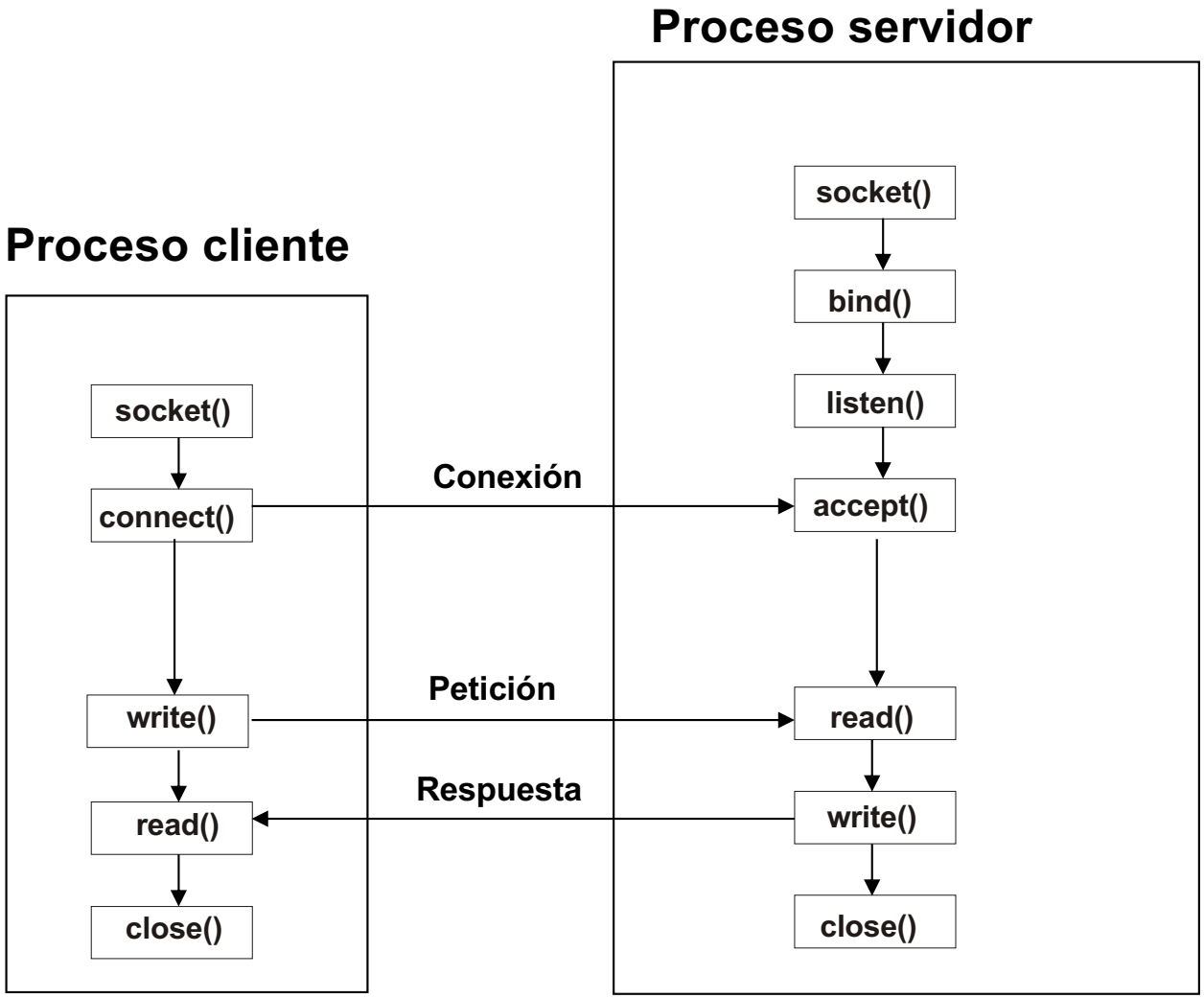
- El servidor sirve las peticiones de forma secuencial
- Mientras está atendiendo a un cliente **no** puede aceptar peticiones de más clientes



Ejemplo: servidor y cliente en TCP



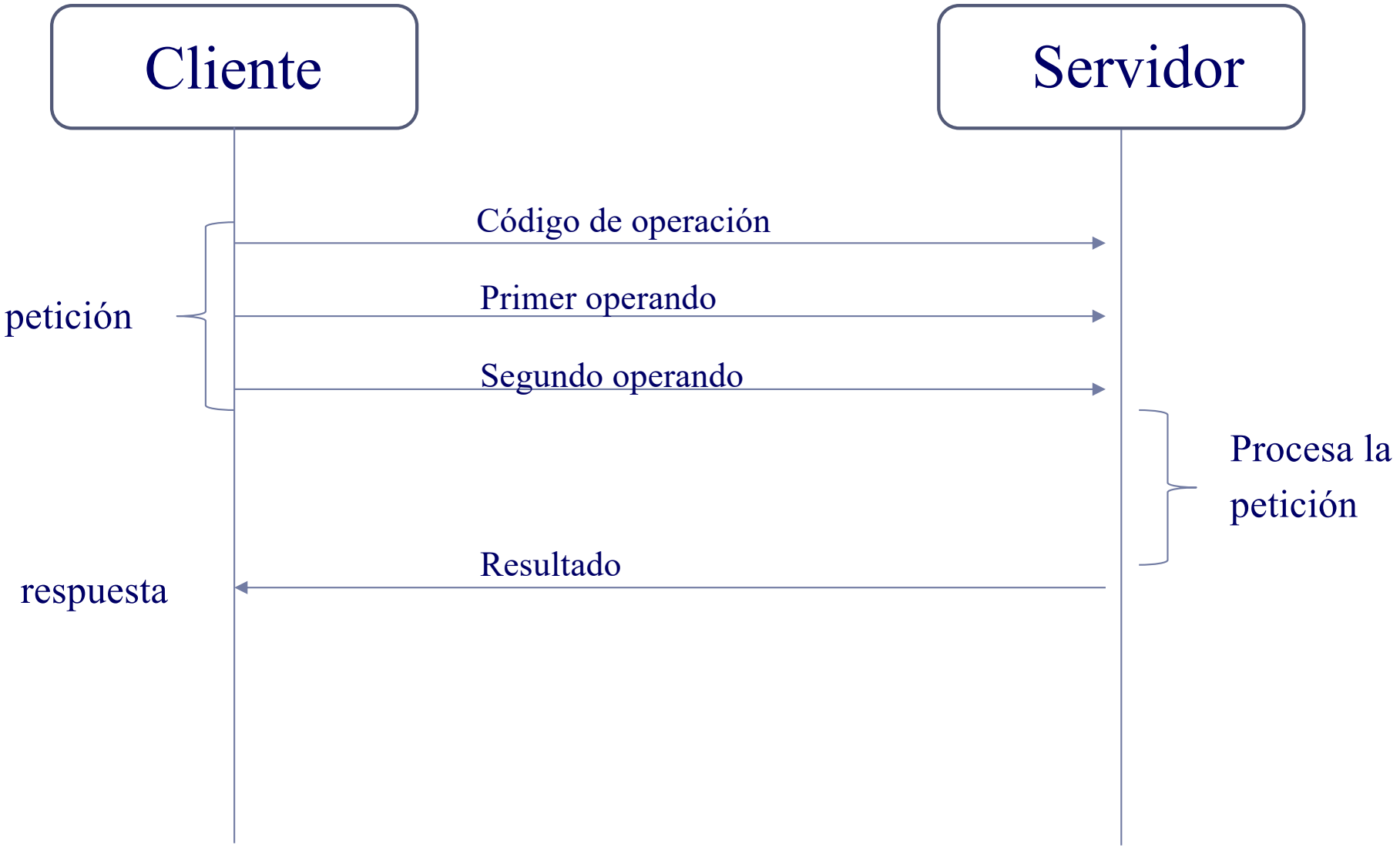
Ejemplo: Modelo de comunicación



Advertencia

- En los siguientes fragmentos de código, no se comprueban en muchos casos los errores que devuelven las llamadas al sistema. En una implementación real han de tratarse todos los errores y comprobar los valores que devuelven todas las llamadas al sistema.

Protocolo


Cliente
Servidor
Código de operación
Primer operando
Segundo operando
petición
Procesa la petición
Resultado
respuesta

Aspectos a tener en cuenta en el diseño

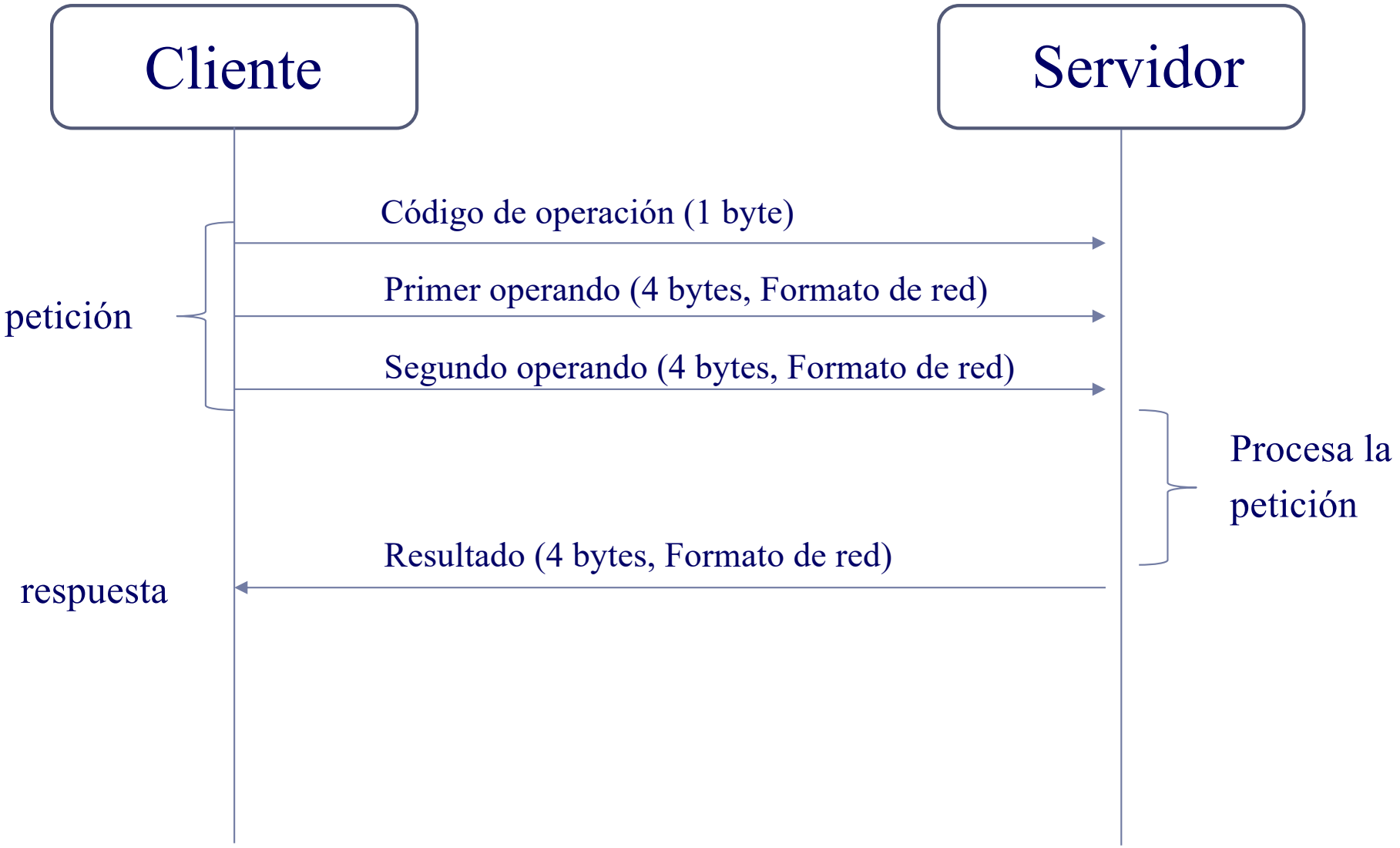
- La solución planteada tiene que tener en cuenta la ejecución en arquitecturas distintas: big-endian o little-endian
- Protocolo definido:
  - El Código de operación es un byte
  - Los enteros son de 32 bits y se envían en el formato de red
  - Antes de enviar un entero se convierte a formato de red:

```
int32_t n = 8;
n = htonl(n);
write(sc, &n, sizeof(int32_t));
```

  - Después de recibir un entero se convierte a formato de host para operar después con él:

```
int32_t n = 8;
read(sc, &n, sizeof(int32_t));
n = ntohl(n);
```

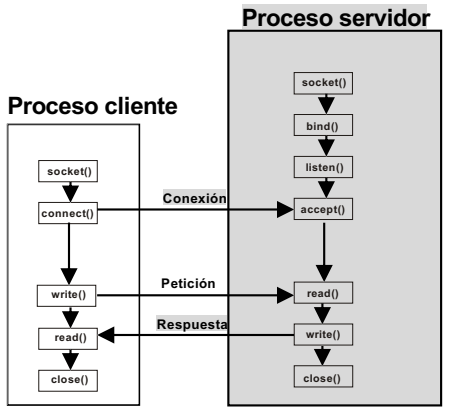
Protocolo detallado



Ejemplo: Servidor TCP

servidor-calc.c

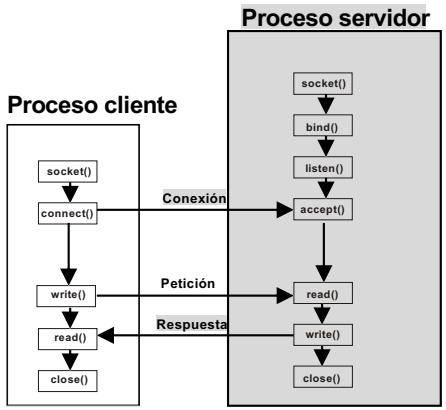
```
#include <stdio.h>
#include <unistd.h>
#include <strings.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
int main(int argc, char *argv[]){
        struct sockaddr_in server_addr,  client_addr;
        socklen_t size;
        int sd, sc;
        int val;
        char op;   int32_t a, b,  res;
        if ((sd =  socket(AF_INET, SOCK_STREAM, 0))<0){
                printf ("SERVER: Error en el socket");
                return (0);
        }
        val = 1;
        setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, (char *) &val, sizeof(int));
```



Ejemplo: Servidor TCP (cont)

servidor-calc.c

```
bzero((char *)&server_addr, sizeof(server_addr));
server_addr.sin_family      = AF_INET;
server_addr.sin_addr.s_addr = INADDR_ANY;
server_addr.sin_port        = htons(4200);

bind(sd, (const struct sockaddr *)&server_addr,
          sizeof(server_addr));
listen(sd, SOMAXCONN);
size = sizeof(client_addr);
```



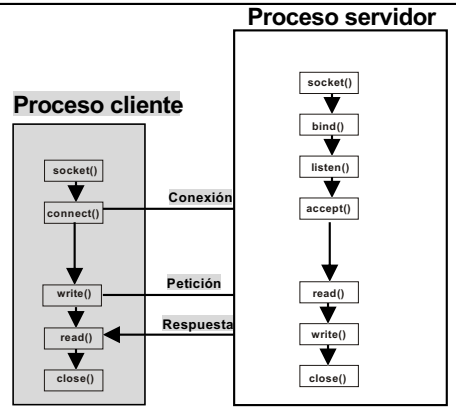
Ejemplo: Servidor TCP (cont)

servidor-calc.c

```
      while (1){
         printf("esperando conexion\n");
         sc = accept(sd, (struct sockaddr *)&client_addr,
                         (socklen_t *)&size);

         read ( sc, (char *) &op, sizeof(char));   // recibe la operación
         read ( sc, (char *) &a, sizeof(int32_t)); // recibe a
         read ( sc, (char *) &b, sizeof(int32_t)); // recibe b

         if (op == 0) // procesa la petición
            res = ntohl(a) + ntohl(b);
         else
            res = ntohl(a) - ntohl(b);
         res = htonl(res);
         write(sc, (char *)&res, sizeof(int32_t));  // envía el resultado
         close(sc);                          // cierra la conexión (sc)
      }
      close (sd);
      return(0);
} /*fin main */
```

Ejemplo: Cliente TCP

cliente-calc.c




```
#include <stdio.h>
#include <netdb.h>
#include <strings.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
int  main(int argc, char **argv)  // en argv[1] == servidor
{
        int sd;
        struct sockaddr_in server_addr;
        struct hostent *hp;
        int32_t a, b , res;          char op;
        if (argc != 2){
                printf("Uso: cliente <direccion_servidor> \n");
                return(0);
        }
        sd = socket(AF_INET, SOCK_STREAM, 0);
```

Ejemplo: Cliente TCP (cont)

cliente-calc.c

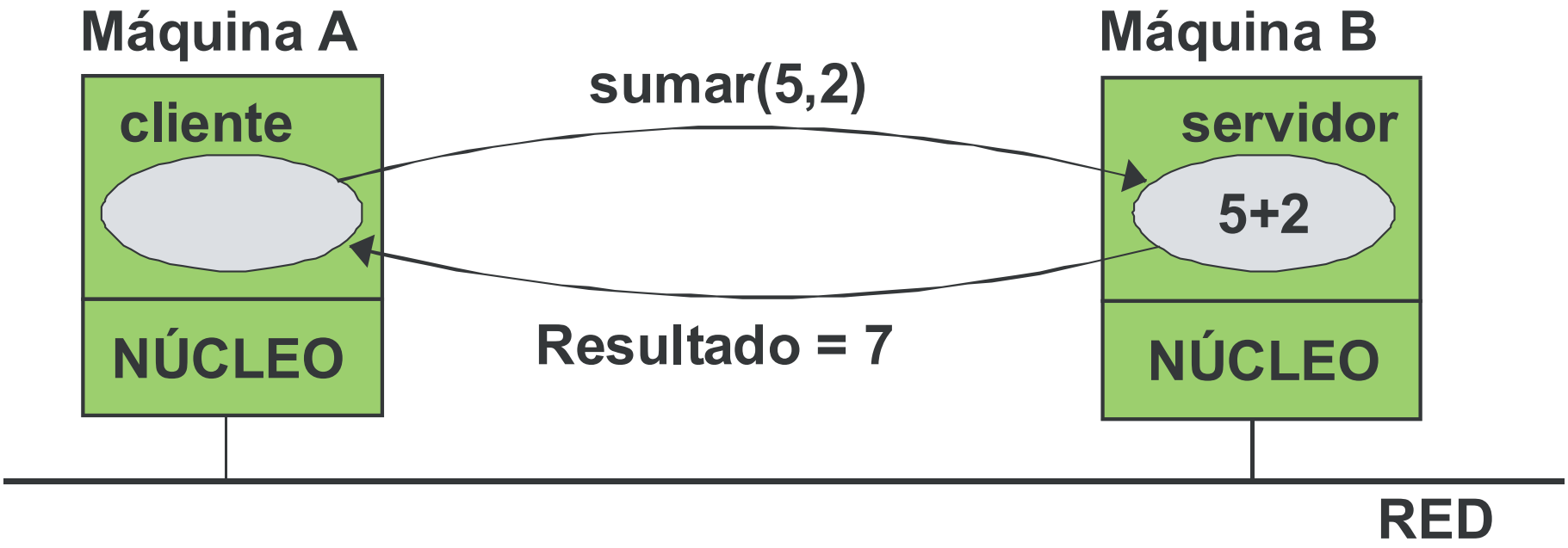
```
bzero((char *)&server_addr, sizeof(server_addr));
hp = gethostbyname (argv[1]);

memcpy (&(server_addr.sin_addr), hp->h_addr, hp->h_length);
server_addr.sin_family  = AF_INET;
server_addr.sin_port    = htons(4200);

// se establece la conexión
connect(sd, (struct sockaddr *) &server_addr,
            sizeof(server_addr));
```

Ejemplo: Cliente TCP - cont

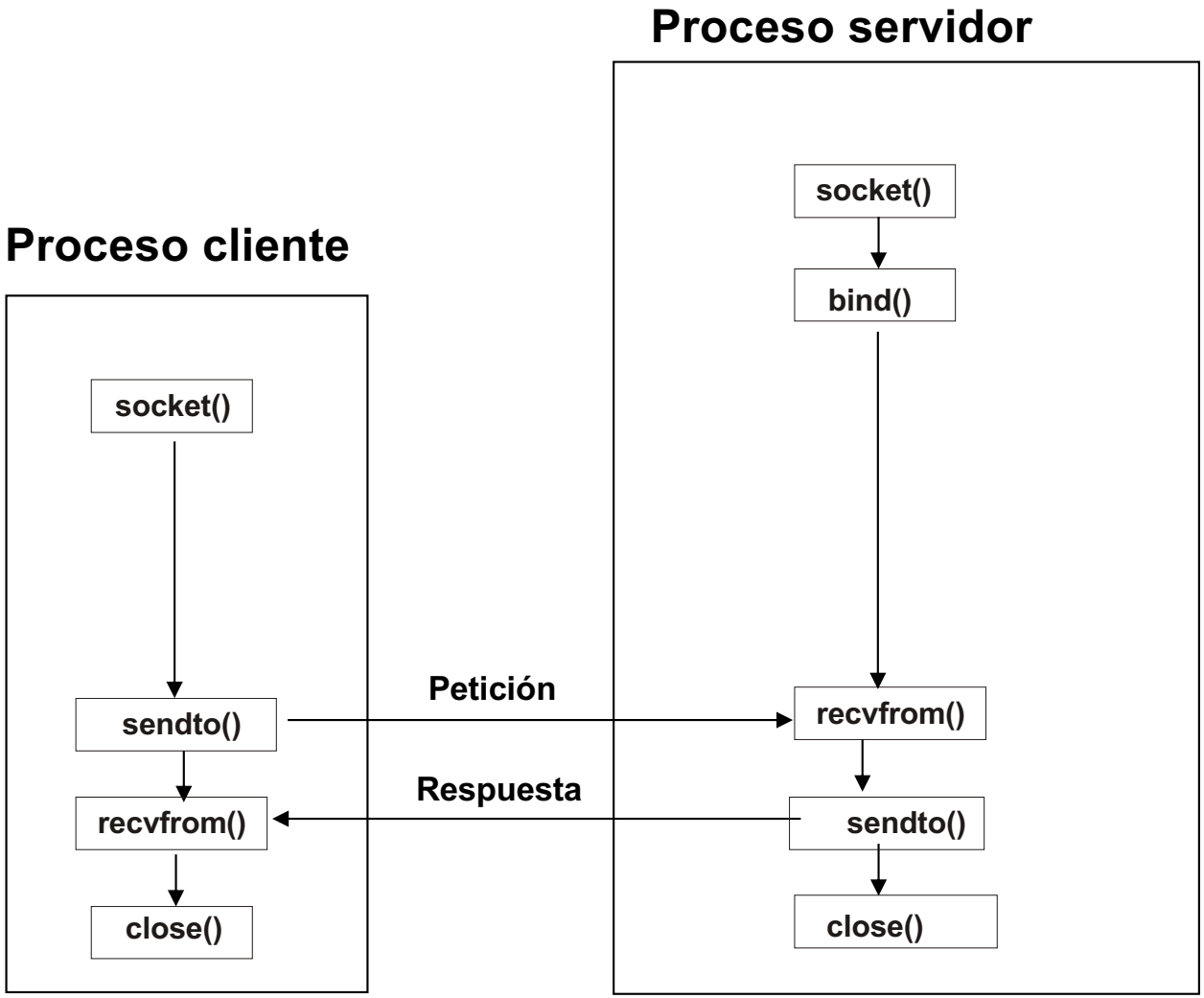
cliente-calc.c

```
    a = htonl(5);
    b = htonl(2);
    op = 0; // suma

    write(sd, (char *) &op, sizeof(char));  // envía la operacion
    write(sd, (char *) &a, sizeof(int32_t)); // envía a
    write(sd, (char *) &b, sizeof(int32_t)); // envía b

    read(sd, (char *) &res, sizeof(int32_t));  // recibe la respuesta

    printf("Resultado es %d \n", ntohl(res));

   close (sd);
   return(0);
} /* fin main */
```

Ejemplo: Servidor y cliente en UDP



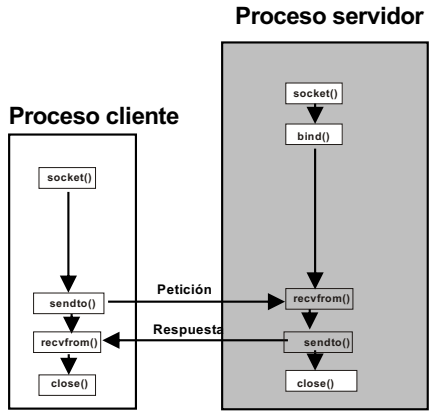
Ejemplo: Modelo de comunicación



Ejemplo: Servidor UDP

servidor-calc.c

```
#include <stdio.h>
#include <strings.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
int main(void)  {
        int32_t peticion[3];
        int s, res, clilen;
        struct sockaddr_in server_addr, client_addr;

        s =  socket(AF_INET, SOCK_DGRAM, 0);

        bzero((char *)&server_addr, sizeof(server_addr));
        server_addr.sin_family          = AF_INET;
        server_addr.sin_addr.s_addr     = INADDR_ANY;
        server_addr.sin_port            = htons(7200);
```



Ejemplo: Servidor UDP (cont)

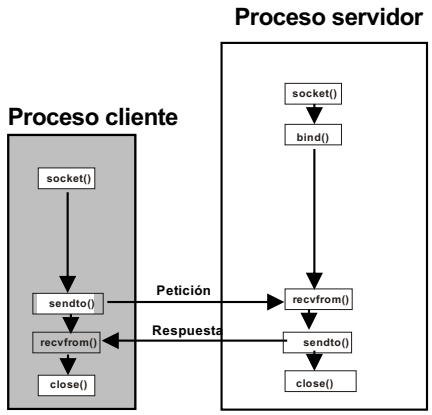
servidor-calc.c

```
  bind(s, (struct sockaddr *)&server_addr, sizeof(server_addr));
        clilen = sizeof(client_addr);
   while (1){
     recvfrom(s, (char *) peticion, 3* sizeof(int32_t), 0,
                    (struct sockaddr *)&client_addr, &clilen);
     peticion[0] = ntohl(peticion[0]);  // operacion
     peticion[1] = ntohl(peticion[1]);
     peticion[2] = ntohl(peticion[2]);
     if (peticion[0]== 0)
          res = peticion[1] + peticion[2];
     else
          res = peticion[1] - peticion[2];

     res = htonl(res);
     sendto(s, (char *)&res, sizeof(int32_t), 0,
                  (struct sockaddr *)&client_addr, clilen);
  }
} /* fin main */
```

Ejemplo: Cliente UDP

cliente-calc.c

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <netdb.h>
#include <strings.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
```

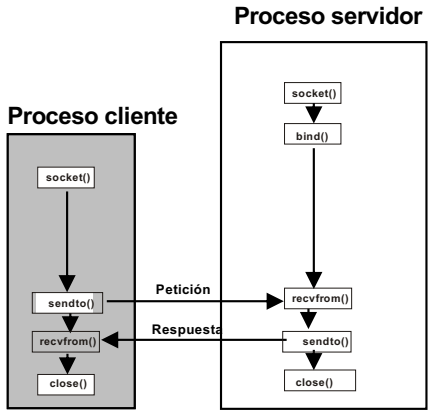



```
int main(int argc, char *argv[]) {
        struct sockaddr_in server_addr, client_addr;
        struct hostent *hp;
        int s;
        int32_t peticion[3], res;
        if (argc != 2){
                printf("Uso: cliente <direccion_servidor> \n");
                return(0);
        }
```

Ejemplo: Cliente UDP (cont)

cliente-calc.c

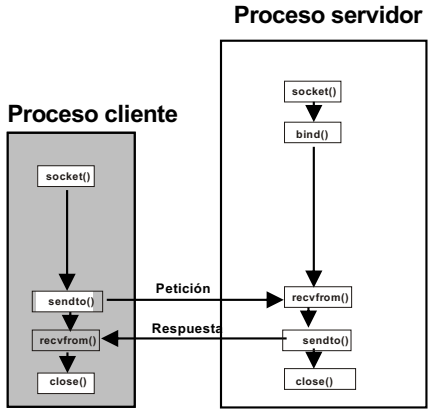
```
s =  socket(AF_INET, SOCK_DGRAM, 0);
hp = gethostbyname (argv[1]);

bzero((char *)&server_addr, sizeof(server_addr));
memcpy (&(server_addr.sin_addr), hp->h_addr, hp->h_length);
server_addr.sin_family = AF_INET;
server_addr.sin_port   = htons(7200);

bzero((char *)&client_addr, sizeof(client_addr));
client_addr.sin_family      = AF_INET;
client_addr.sin_addr.s_addr = INADDR_ANY;
client_addr.sin_port        = htons(0);
```



Ejemplo: Cliente UDP (cont)

cliente-calc.c

```
    bind (s, (struct sockaddr *)&client_addr,
              sizeof(client_addr));

    peticion[0] = htonl(0);          // sumar
    peticion[1] = htonl(5);
    peticion[2] = htonl(2);

    sendto(s, (char *) peticion, 3 * sizeof(int32_t), 0,
           (struct sockaddr *) &server_addr, sizeof(server_addr));

    recvfrom(s, (char *)&res, sizeof(int32_t), 0, NULL, NULL);

    res = ntohl(res);
    printf("2 + 5 = %d\n", res);
    close(s);
}
```



Consideraciones a tener en cuenta

- Comprobación de errores. **Muy importante**
- Aspectos a tener en cuenta en la transferencia de los datos multibyte:
  - Ordenamiento de los bytes
    - ¿Qué ocurre si el cliente es *little-endian* y el servidor *big-endian*?
    - Hay que resolver el problema de la representación de los datos. Una posibilidad es utilizar las funciones:

```
u_long  htonl(u_long  hostlong)
u_short htons(u_short hostshort)
u_long  ntohl(u_long  netlong)
u_short ntohs(u_short netshort)
```

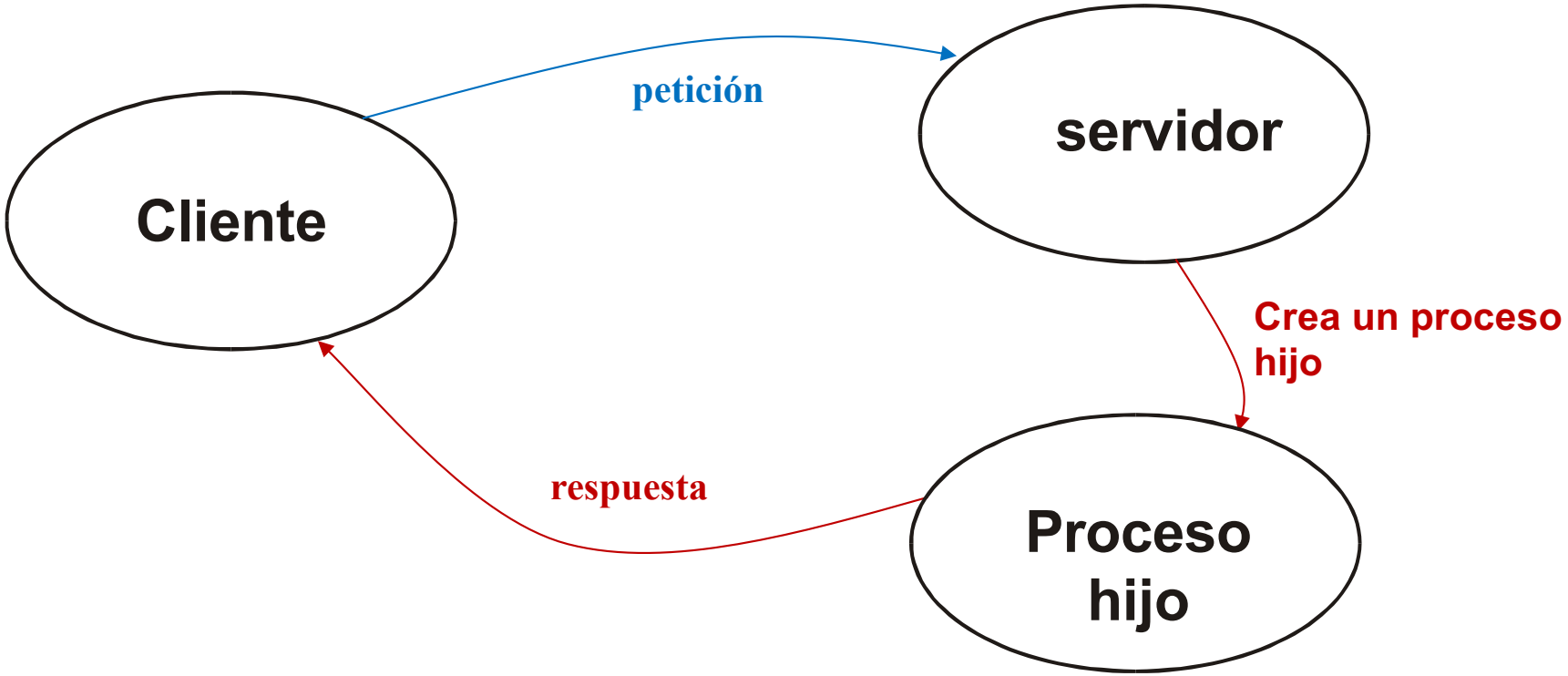
```
num[0] = 0;
num[1] = 5;
num[1] = 2;
```

error

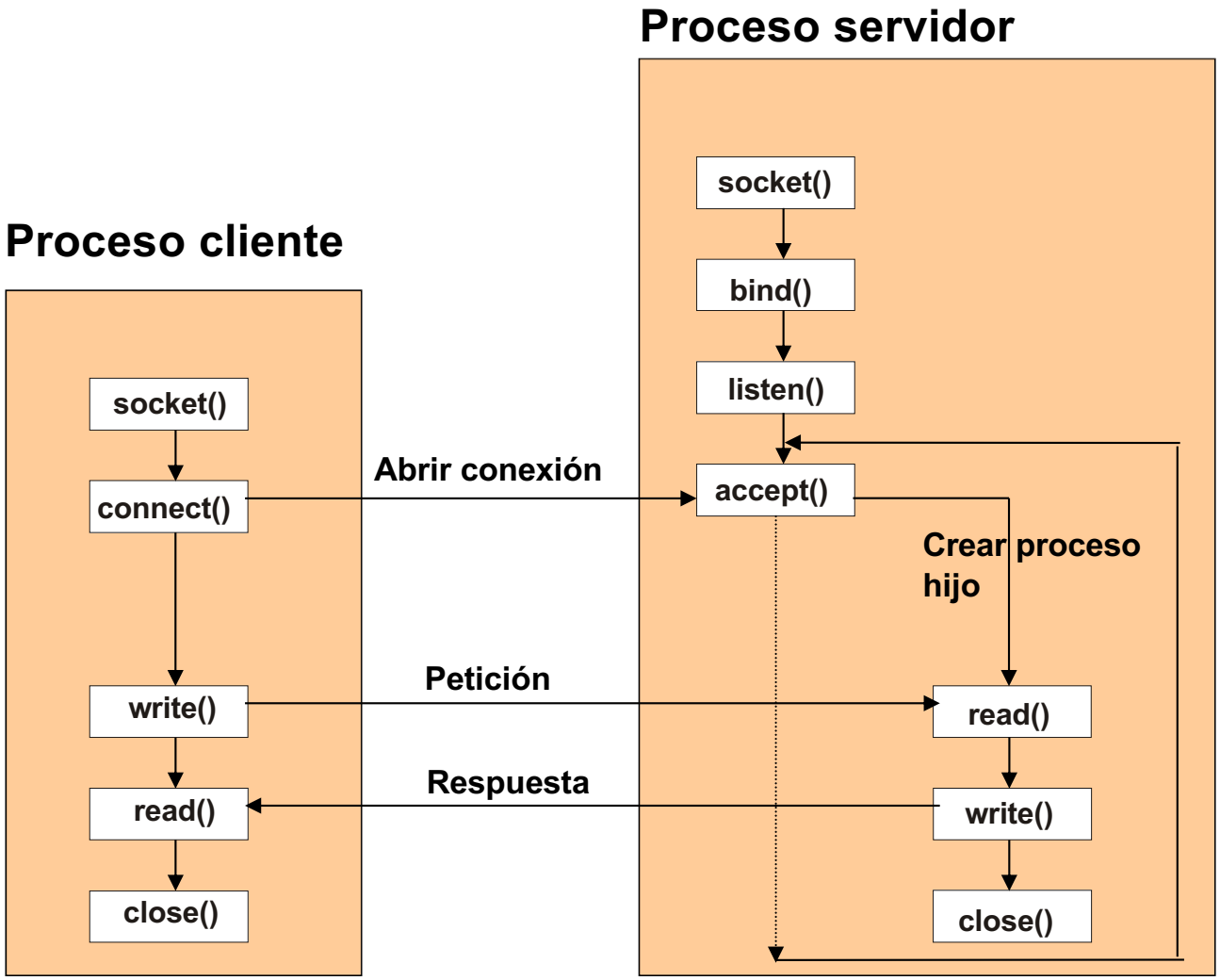
  - Definir el formato de representación e intercambio de datos. Ejemplo:
    - Los datos que se envian a la red deben estar en Network Byte Order
    - Los datos que se reciben de la red deben estar en Host Byte Order

Modelo de servidor concurrente

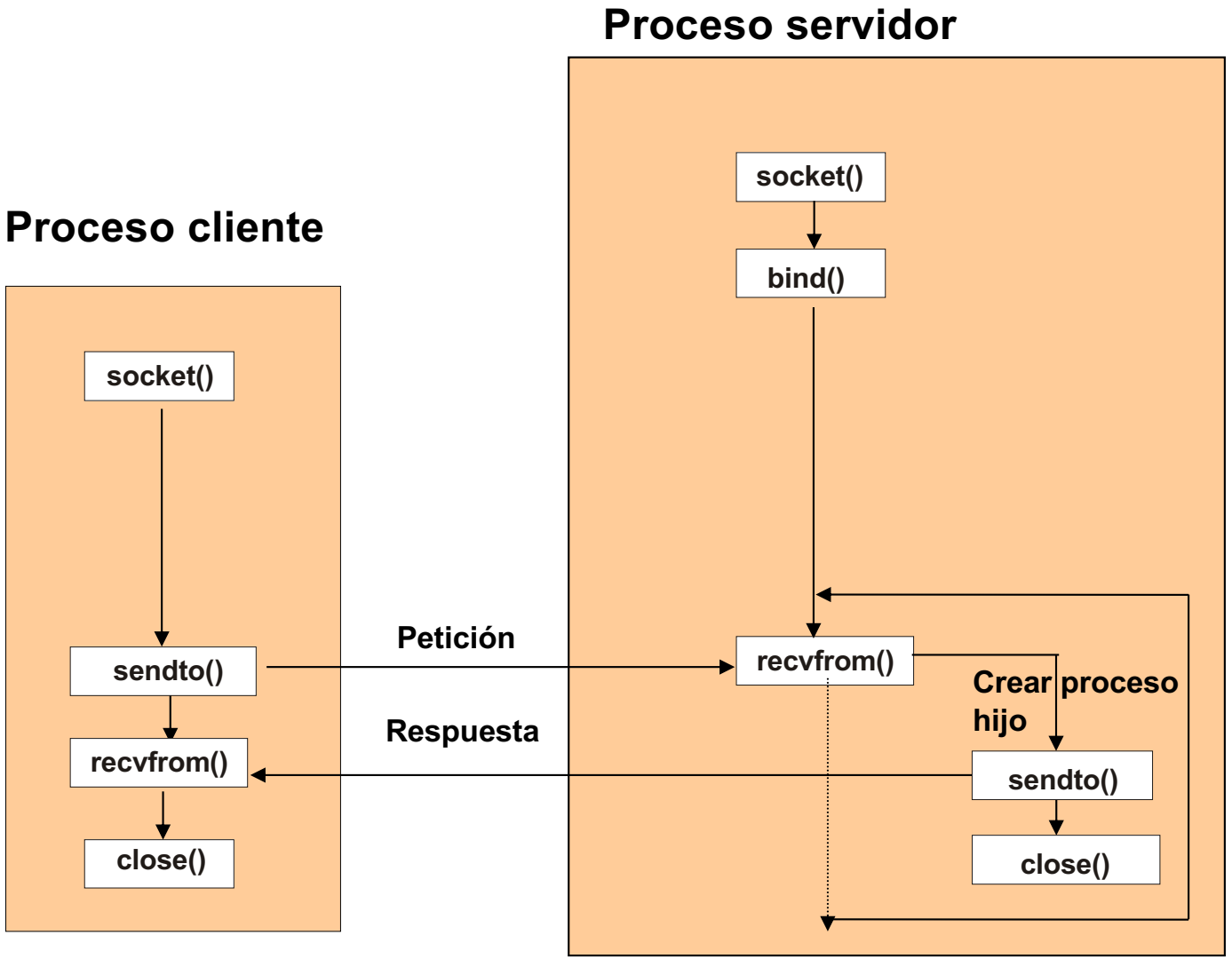
- El servidor crea un hijo que atiende la petición y envía la respuesta al cliente
- Se pueden atender múltiples peticiones de forma concurrente



Servidores concurrentes con sockets *stream*



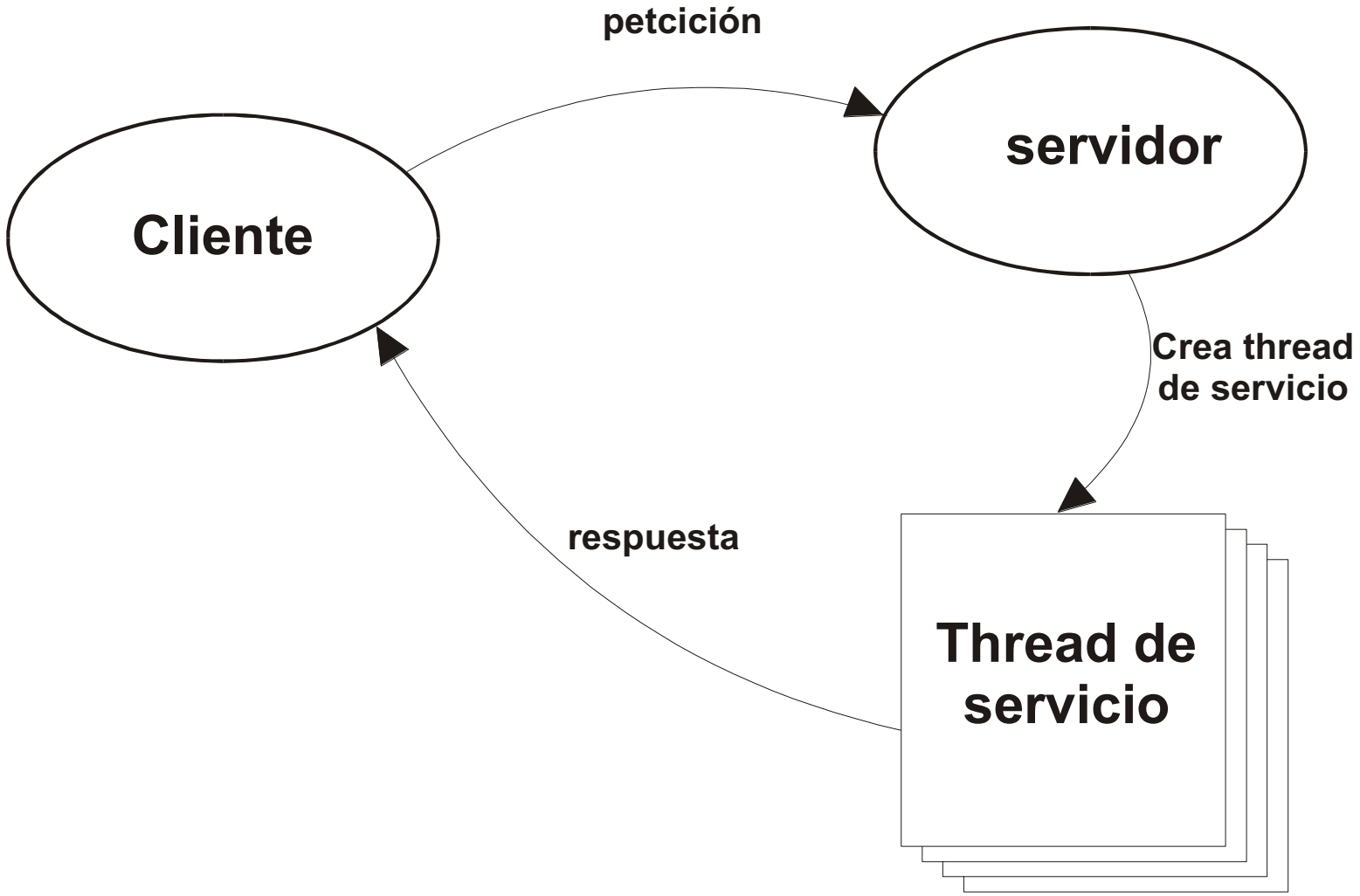
Modelo de comunicación con sockets datagrama



Servidores concurrentes con threads

- Procesos ligeros (*threads*)
  - Creación bajo demanda
  - Pool de threads

Servidores concurrentes con P. ligeros



Procesos concurrentes con *threads (creación bajo demanda)*

- El servidor crea un socket `s` y le asocia una dirección
- El cuerpo principal del servidor es:

```
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

 for(;;) {
      sd = accept(s, (struct sockaddr *)&  &cliente, &len);
      pthread_create(&thid, &attr, tratar_peticion, (void *)&sd);
   }
```

- La función que ejecuta el proceso ligero es:

```
void tratar_peticion(void * s) {
   int s_local;

   s_local = (* (int *) s);
   /* tratar la petición utilizando el descriptor de s_local */

   close(s_local);
   pthread_exit(NULL);
}
```

- **¿Es correcta esta solución?**

Necesario sincronización

- La solución anterior es incorrecta ya que los procesos padre e hijo compiten por el acceso al descriptor (sd) devuelto por *accept*
  - El proceso ligero hijo creado podría usar sd en tratar_petición mientras que un nuevo sd es asignado en *accept*
  - **Condiciones de carrera**
- Necesario sincronizar las acciones con mutex y variables condicionales
  - En el proceso servidor principal
  - En los procesos ligeros hijos

Necesario sincronización

- El proceso ligero principal debe ejecutar:

```
for(;;) {
        sd = accept(s, (struct sockaddr *)  &cliente, &len);
        pthread_create(&thid, &attr, tratar_peticion, (int *)&sd);
        /* esperar a que el hijo copie el descriptor */

        pthread_mutex_lock(&m);
        while(busy == TRUE)
                pthread_cond_wait(&c, &m);
        busy = TRUE;
        pthread_mutex_unlock(&m);

}
```

Necesario sincronización

- El proceso ligero hijo debe ejecutar:

```
void tratar_peticion(int * s) {
  int s_local;

  pthread_mutex_lock(&m);
  s_local = (* (int *)s);
  busy = FALSE;
  pthread_cond_signal(&c);
  pthread_mutex_unlock(&m);

  /* tratar la petición utilizando el descriptor s_local */
  pthread_exit(NULL);
}
```

Heterogeneidad

- En general cuando se desarrolla una aplicación con sockets hay que plantear una solución que sea independiente de:
  - Arquitectura (Little-endian, big-endian)
  - Lenguaje de programación
- Emplear soluciones que definan el tamaño de los enteros (por ejemplo: 32 bits) puede ser un problema
- Solución: desarrollar aplicaciones que codifiquen los datos en cadenas de caracteres y envíen cadenas de caracteres
  - Protocolos basados en texto (Ejemplo: HTTP, SMTP)

Lectura de cadenas de caracteres con sockets stream

- Cuando una cadena de caracteres finaliza con el código ASCII '\0' no se sabe a priori su longitud y no se puede usar la función recvMessage para leerla
- En este caso hay que leer byte a byte hasta leer el el código ASCII '\0':

lines.c

```
ssize_t readLine(int fd, void *buffer, size_t n)
{
        ssize_t numRead;  /* num of bytes fetched by last read() */
        size_t totRead;       /* total bytes read so far */
        char *buf;
        char ch;

        if (n <= 0 || buffer == NULL) {
                  errno = EINVAL;
                  return -1;
        }

        buf = buffer;
        totRead = 0;
```

Lectura de cadenas de caracteres con sockets stream

lines.c

```
for (;;) {
        numRead = read(fd, &ch, 1);   /* read a byte */
        if (numRead == -1) {
                if (errno == EINTR) /* interrupted -> restart read() */
                continue;
        else
                return -1;                      /* some other error */
        } else if (numRead == 0) {    /* EOF */
                if (totRead == 0)   /* no byres read; return 0 */
                return 0;
                else
                break;
        } else {  /* numRead must be 1 if we get here*/
                if (ch == '\n')
                        break;
                if (ch == '\0')
                        break;
                if (totRead < n - 1) {                  /* discard > (n-1) bytes */
                        totRead++;
                                *buf++ = ch;
                        }
                }
        }
        *buf = '\0';
        return totRead;
}
```

Envío de una cadena de caracteres con socktes stream

- Para enviar una cadena:

```
char buffer[256];
strcpy(buffer, "Cadena a enviar");
sendMessage(socket, buffer, strlen(buffer)+1);
```

Envía la cadena más el código ASCII '\0' que indica el fin de la cadena.

Servidor y cliente de suma enviando cadenas de caracteres

- Para enviar un número como cadena:

```
int n = 1234;
char buffer[256];
sprintf(buffer, "%d", n);
sendMessage(socket, buffer, strlen(buffer)+1);
```

  Envía la cadena más el código ASCII '\0' que indica el fin de la cadena.

- De esta forma se independiza del formato concreto en el que se almacenen los enteros

Servidor y cliente de suma enviando cadenas de caracteres

- Para recibir un número como cadena:

```
int n ;
char buffer[256];
readLine(socket, buffer, 256);
n = atoi(buffer);
```

Uso de strtol para detectar errores

- La función `atoi` no comprueba que el buffer a convertir sea realmente un numero.

- La función `strtol` sí que lo hace:

Uso de strtol para detectar errores

```
int n ;
char buffer[256], *endptr;
recvMessage(socket, buffer, 256);
// convert buffer in mumber
errno = 0;

/* To distinguish success/failure after call */
n = strtol(str, &endptr, 10);  // base 10
/* Check for various possible errors */
if ((errno == ERANGE && (n == LONG_MAX || n == LONG_MIN)) ||
     (errno != 0 && n == 0)) {
        perror("strtol"); exit(EXIT_FAILURE);
} if (endptr == str) {
        fprintf(stderr, "No digits were found\n");
        exit(EXIT_FAILURE);
}
/* If we got here, strtol() successfully parsed a number */
printf("strtol() returned %ld\n", n);
```

Protocolo del cliente de suma usando cadenas

```
char op = 0; // suma
int a = 5 ;
int b = 6;
int res;
char buffer[256];

// PETICIÓN
sendMessage(socket, &op, 1) ;   // envío operación
sprintf(buffer, "%d", a);  // envío a
sendMessage(socket, buffer, strlen(buffer)+1);
sprintf(buffer, "%d", b);   // envío b
sendMessage(socket, buffer, strlen(buffer)+1);

//RESPUESTA
readLine(socket, buffer, 256);
res = atoi(buffer);       // obtiene res
```

Protocolo de servidor de suma usando cadenas de caracteres

```
char op = 0; // suma
int a;
int b;
int res;
char buffer[256];

recvMessage(socket, &op, 1);  // obtiene op
// recibe argumentos
readLine(socket, buffer, 256);
a = atoi(buffer);          // obtiene a
readLine(socket, buffer, 256);
b = atoi(buffer);          // obtiene b

if (op == 0) {
        res = a + b;
        sprintf(buffer, "%d", res);
        sendMessage(socket, buffer, strlen(buffer)+1);
}
```

Guía de desarrollo de aplicaciones cliente-servidor con paso de mensajes

1. Identificar el cliente y el servidor
   - Cliente: elemento activo, varios
   - Servidor: elemento pasivo
2. Protocolo del servicio
   - Identificar los tipos mensajes y la secuencia de intercambios de mensajes (peticiones y respuestas)
3. Elegir el tipo de servidor
   - UDP sin conexión
   - TCP:
     - Una conexión por sesión
     - Una conexión por petición
4. Identificar el formato de los mensajes (representación de los datos)
   - Independencia (lenguaje, arquitectura, implementación, …)

Ejercicio

- Desarrollar un servidor que permita obtener la hora, la fecha y el día de la semana en la que cae un dia determinado. Diseñar y desarrollar el cliente y el servidor.