Fundamentos de Software de Comunicaciones

Tema 3

Programación de sockets

Contenidos

- Conocimientos previos
  - Direccionamiento en Internet
  - Problemas con la representación de datos
    - Tipos de datos
    - Conversiones
    - Alineamiento de bytes
- El modelo cliente/servidor
  - Esquema general
  - Direcciones
- Sockets
  - Definición
  - Tipos
- Programación de sockets básica
  - Direcciones de sockets
  - Sockets UDP
  - Sockets TCP
  - Diseño de servidores

Motivación

- Hasta ahora hemos **comunicados** procesos que están en la **misma máquina** usando
  - Señales
  - Tuberías/FIFOs

- ¿Qué pasa si queremos comunicar procesos que están en **máquinas distintas** conectadas a través de una red?

- Consideraciones
  - ¿Dónde está el proceso? Es decir, ¿dónde está la máquina que ejecuta ese proceso?
  - ¿Qué tipo de máquina es? Linux, windows, mac, android…
  - ¿Cómo representa los datos? Por ejemplo, ¿un entero es igual en cada una de esas máquinas?

Conocimientos previos

1. Direccionamiento en Internet
2. Representación de datos

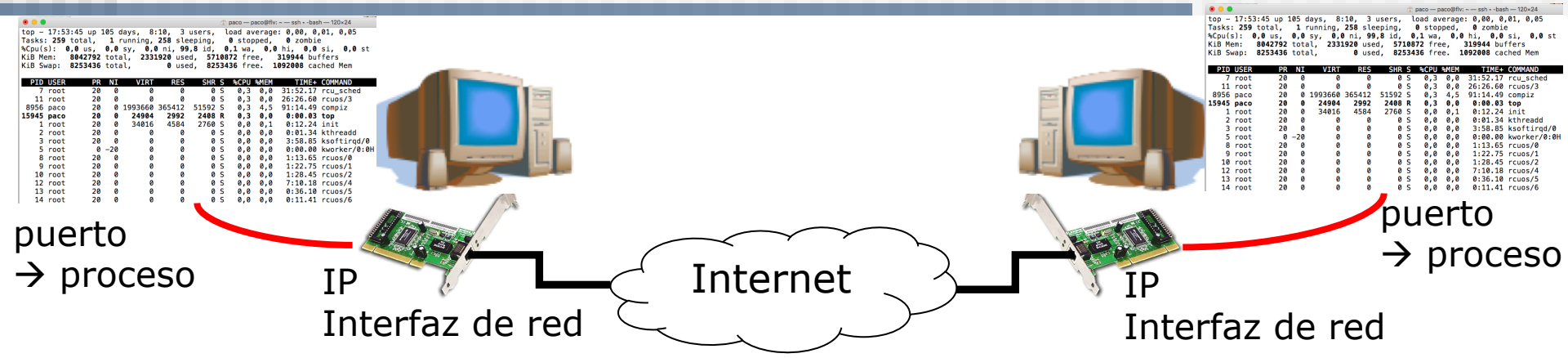
Conocimientos previos
Direccionamiento en Internet

- Si queremos comunicarnos con un proceso que se está ejecutando en una máquina remota, antes hay que saber cómo llegar a él
  - Por ejemplo: para enviar una carta a alguien, hay que saber su dirección postal

- Mecanismos de comunicación entre procesos vistos hasta ahora:

Mecanismo	"Dirección"
Señales	PID
Tuberías	Descriptor de fichero con el extremo de la misma

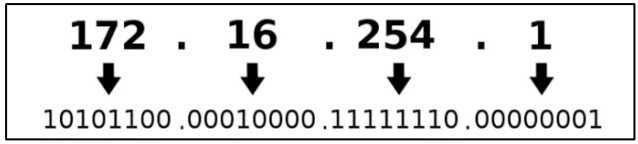
Conocimientos previos
Direccionamiento en Internet



- Una **dirección de socket** está compuesta de dos campos:
  - **La dirección IP**: que es la dirección de la máquina donde se ejecuta el proceso
  - **El puerto**: que identifica al proceso dentro de la máquina que atiende la petición
  - Siguiendo el símil anterior:
    - La dirección IP sería la dirección de un bloque de pisos
    - El puerto sería cómo identificar la casa dentro del bloque (planta, letra, etc.)

Conocimientos previos
Direccionamiento en Internet

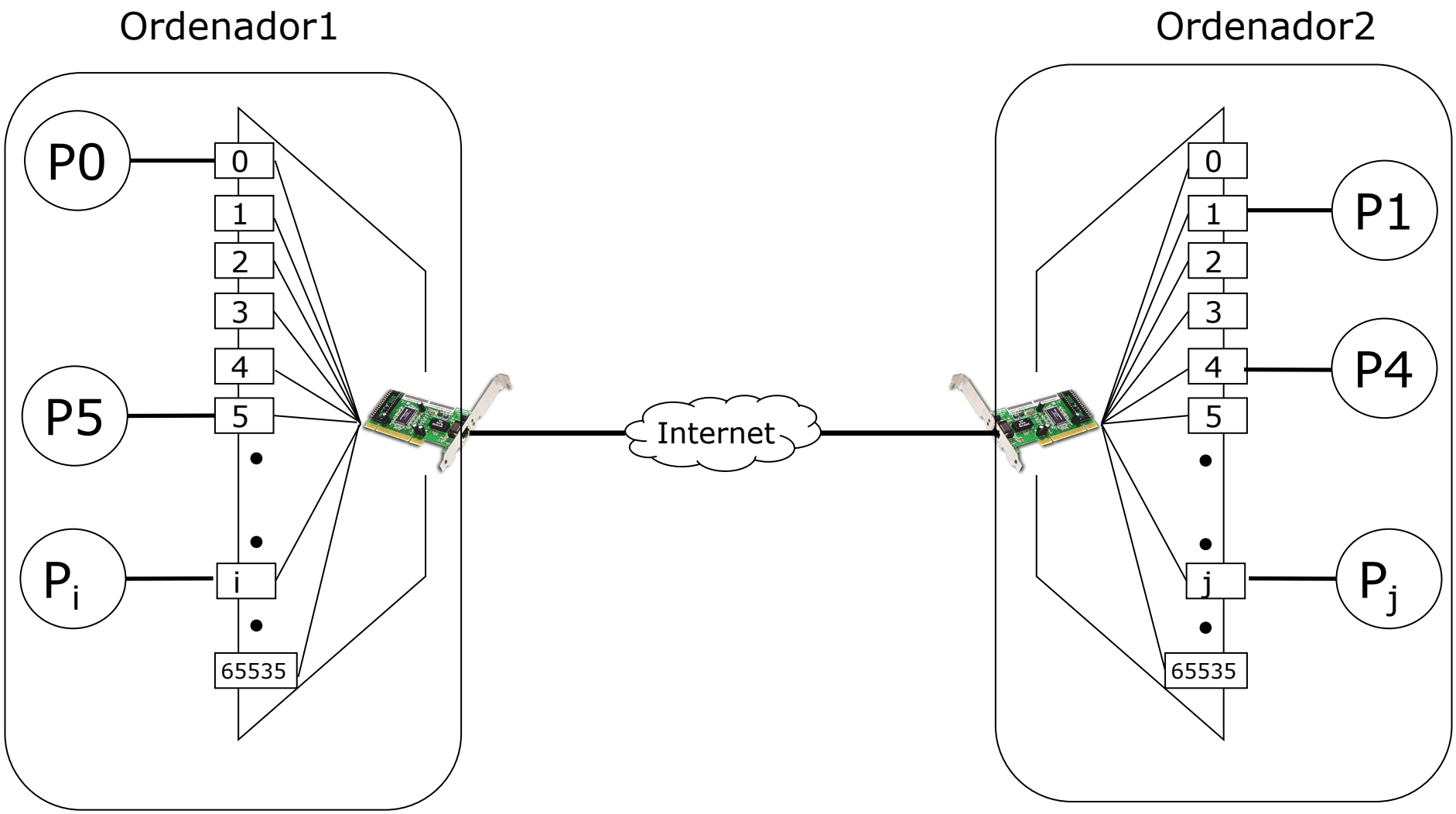
- Direcciones IP (**direcciones de red**)
  - Tienen **32 bits** (IPv4)
  - Se representan usualmente como cuatro números separados por punto:
    - Por ejemplo: **172.16.254.1**
    - Cada número tiene un valor entre **0** y **255**
    - Equivale a **8** bits → **256** valores posibles



  - Identifican **unívocamente** a cualquier ordenador en internet
    - Más concretamente, a cualquier **interfaz de red** (tarjeta de red)

- Puertos (**direcciones de transporte**)
  - Tienen **16 bits**
  - Se representan con un número entero entre **0** y **65535**
    - No todos los valores los podremos usar (detalles más adelante)
    - Equivalen a un entero corto (short): **2^{16} = 65536** valores posibles
  - Identifican **unívocamente** a un **proceso** dentro de una máquina (multiplexación)
    - Dos procesos no pueden estar escuchando en el mismo puerto
  - Se asocian a servicios:
    - Puerto 80: Servidor web
    - Puerto 21: FTP

Conocimientos previos
Direccionamiento en Internet



Conocimientos previos
Representación de los datos

- Los procesos remotos que queremos comunicar pueden estar ejecutándose en **cualquier tipo de plataforma** (Linux, Windows, Intel, AMD, PowerPC…)
  - Eso supone que puede que la forma en la que **represente** (almacene en memoria) **un tipo de datos** concreto difiera entre ambos
  - Por ejemplo, un entero puede tener 4 bytes en el proceso A y 8 bytes en el proceso B, ¿cómo se intercambiarían ese valores de ese tipo?
- Aspectos a considerar
  - Tipos de enteros: **estándar C99**
  - Tipos de arquitectura: **big-endian** vs **little-endian**
  - **Alineamiento** de bytes

Conocimientos previos
Representación de datos: tipos de enteros (std C99)

- Para evitar problemas de compatibilidad, cuando se migra un código a distintas plataformas, se debe evitar dependencias con int, short, long, etc.

- **#include <stdint.h>**
  - Define tipos de enteros independientes de la plataforma

    uint8_t/int8_t

    uint16_t/int16_t

    uint32_t/int32_t

    uint64_t/int64_t

- **#include <inttypes.h>**
  - Para imprimir de forma correcta estos valores:

```
uint64_t v = 61489146900911925 93;
printf("valor  = %"PRIu64" \n", v);
```

Conocimientos previos
Representación de datos: tipo de arquitectura

- Aparece en tipos de datos que requieren más de **1 byte** de memoria para almacenarlos: **short**, **int**, **float**, …
  - No ocurre con el tipo **char**, que ocupa 1 byte

- Endianness de las arquitecturas
  - **Big-endian**: el byte más significativo se almacena primero
  - **Litte-endian**: el byte menos significativo se almacena primero
  - Las tablas muestran la representación en memoria del valor hexadecimal 0x0A0B0C0D en dos máquinas con distintas arquitecturas (se han utilizado las direcciones de memoria de la 0x1100 a la 0x1103, donde cada una almacena 1 byte)

Byte más significativo

0x1103	0x0D
0x1102	0x0C
0x1101	0x0B
0x1100	0x0A

big-endian

0x1103	0x0A
0x1102	0x0B
0x1101	0x0C
0x1100	0x0D

little-endian

Byte menos significativo

- Si no se especifica lo contrario, los datos de control de más de un byte se transmiten en formato **big-endian**

Conocimientos previos
Representación de datos: tipo de arquitectura

- **Envío de datos**: todos los procesos antes de enviar datos que ocupan más de 1 byte en memoria convierten de su arquitectura interna (**host byte order**) al formato **big-endian** (**network byte order**)

- **Recepción de datos**: todos los procesos que reciben datos que ocupan más de 1 byte en memoria convierten del formato big-endian (**network byte order**) a su formato interno propio (**host byte order**)

- Funciones de conversión:

'h' : host byte order 'n' : network byte order

's' : short (16bit) 'l' : long (32bit)

```
uint16_t htons(uint16_t);
uint16_t ntohs(uint16_t);

uint32_t htonl(uint32_t);
uint32_t ntohl(uint32_t);
```

```
Ejemplo:
uint16_t dato = 0xFF00;
uint16_t dato_en_big_endian = htons(dato);
```

Conocimientos previos
Representación de datos: alineamiento de bytes

- **Alineamiento de bytes**: ubicación de datos en memoria en múltiplos del tamaño de palabra

- La mayor parte de procesadores de 16, 32 y 64 bits no permiten almacenar palabras en cualquier offset de memoria
  - En procesadores de 32 bits la memoria se accede realizando ciclos de bus de 32 bits
    - En cada acceso a memoria, se pueden leer/escribir **4 bytes** (32 bits)
    - Los datos de tipo **uint32_t** se alinean sólo en direcciones de memoria divisibles entre cuatro
  - Si en en esos 4 bytes cabe más de una variable, se alinean en las posiciones pares, i.e., cada **2** bytes

- **Padding** (relleno): son bytes extra que aparecen en memoria para que la alineación sea correcta

- Cada compilador elige cómo representar una estructura en memoria
  - Si el compilador del programa emisor elige una representación distinta al del receptor se producirá una **incompatibilidad** al serializar/deserializar los datos
  - Cada compilador se adapta a la arquitectura para la que genera código

Conocimientos previos
Representación de datos: alineamiento de bytes

```
struct {
        char a;
        short b;
        int c;
        char d;
}
```

- Se asume que
  - char = 1 byte
  - short = 2 bytes
  - int = 4 bytes
- La forma más compacta de almacenarlo sería:

Cada bloque ocupa 1 byte

Cada línea ocupa 4 bytes: 1 palabra

a	b	b	c
c	c	c	d

- Sin embargo, leer/escribir el `int c` supondría a la CPU dos accesos a memoria → ineficiencia
- Alineamiento de en palabras de 4 bytes
  - **a** y **b** se caben en la misma palabra y se pueden obtener en un único acceso
  - Aparece un relleno para alinear **b** a una dirección par

a	#relleno#	b	b
c	c	c	c
d	#relleno#	#relleno#	#relleno#

Conocimientos previos
Representación de datos: alineamiento de bytes

- El compilador debe respetar las restricciones de alineamiento del procesador, por lo que tendrá que añadir bytes de relleno a las estructuras definidas por el programador para cumplir dichas restricciones

```
struct Mensaje{ /*Estructura definida por el programador*/
   uint16_t opcode;
   uint8_t subfield;
   uint32_t length;
   uint8_t version;
   uint16_t destino;
};
```

- El compilador internamente cambia a esta estructura

```
struct Mensaje{ /*Estructura que se compila en realidad*/
   uint16_t opcode;
   uint8_t subfield;
   uint8_t relleno1;    /*relleno para alinear el siguiente campo a 4 bytes*/
   uint32_t length;
   uint8_t version;
   uint8_t relleno2;    /*relleno para alinear el siguiente campo a 2 bytes*/
   uint16_t destino;
   uint8_t relleno3[4];/*relleno para alinear la estructura completa en 16 bytes*/
};
```

- Como resultado en nuestros programas

Todo struct se ha de enviar campo a campo

Ejemplo de envío de una estructura

```c
struct PDU {
    uint16_t size;
    char payload[T];
}
```

```c
/* La estructura se rellena con la información correspondiente */
data.payload = ...
data.size = ...

// Convertimos a formato de red
uint16_t data_big_endian = htons(data.size);
if (write_n(canal, &data_big_endian, sizeof(uint16_t)) != sizeof(uint16_t)) {
    perror("write PDU.size");
    close(canal);
    exit(-1);
}

if (write_n(canal, data.payload, data.size) != data.size) {
    perror("write PDU.payload");
    close(canal);
    exit(-1);
}
```

Emisor

Receptor

```c
/* Reservamos memoria para la recepción de los datos */
struct PDU data_received;
uint16_t data_big_endian;
if (read_n(canal, &data_big_endian, sizeof(uint16_t)) != sizeof(uint16_t)) {
    perror("read PDU.size");
    close(canal);
    exit(-1);
}
data.size  = ntohs(data_big_endian);

if (read_n(canal, data.payload, data.size) != data.size) {
    perror("read PDU.payload");
    close(canal);
    exit(-1);
}
```

El modelo cliente/servidor

1. Esquema general
2. Direcciones de servicios

El modelo cliente/servidor

- Es el modelo seguido en las comunicaciones en Internet

- El *cliente*
  - Accede al servidor porque requiere de sus servicios
  - Debe saber dónde está ejecutándose el servidor
    - En la máquina en la que está, identificada por su **dirección IP**
    - En el **puerto** donde está escuchando peticiones de servicio

- El *servidor*
  - Espera peticiones de servicio de clientes
  - Responde con el servicio solicitado

- Si un proceso es *cliente* o *servidor* tiene que utilizar diferentes llamadas al sistema operativo
  - que pertenecen a la librería de *sockets* (conectores, en español)

Direcciones de servicios: puertos

- Los servidores ofrecen sus servicios en **direcciones de transporte**
  - Permite que un mismo equipo proporcione múltiples servicios en diferentes puertos
  - Los puertos han de ser conocidos por el cliente

- Hay **puertos conocidos** donde escuchan los Servidores de Internet estándar (ver el fichero /etc/services en UNIX). Van desde el valor 0 al 1023
  - Los procesos que quieran vincularse a estos puertos necesitan privilegios de superusuario (root)
  - Ejemplos: **echo** (7), **SMTP** (25), **HTTP** (80), **FTP** (21), **Telnet** (23)

- Hay puertos reservados por la **IANA** a petición de una organización u empresa (del 1024 al 49151):
  - Pero pueden utilizarse por cualquier proceso si se comprueba que no están ocupados en una máquina concreta

- Por último, hay puertos dinámicos o **efímeros** (del 49152 al 65535):
  - Que son los que se utilizan, por ejemplo, para las aplicaciones cliente. Normalmente, se deja que los escoja el S.O.

Sockets

1. Definición
2. Tipos

Definición de un socket

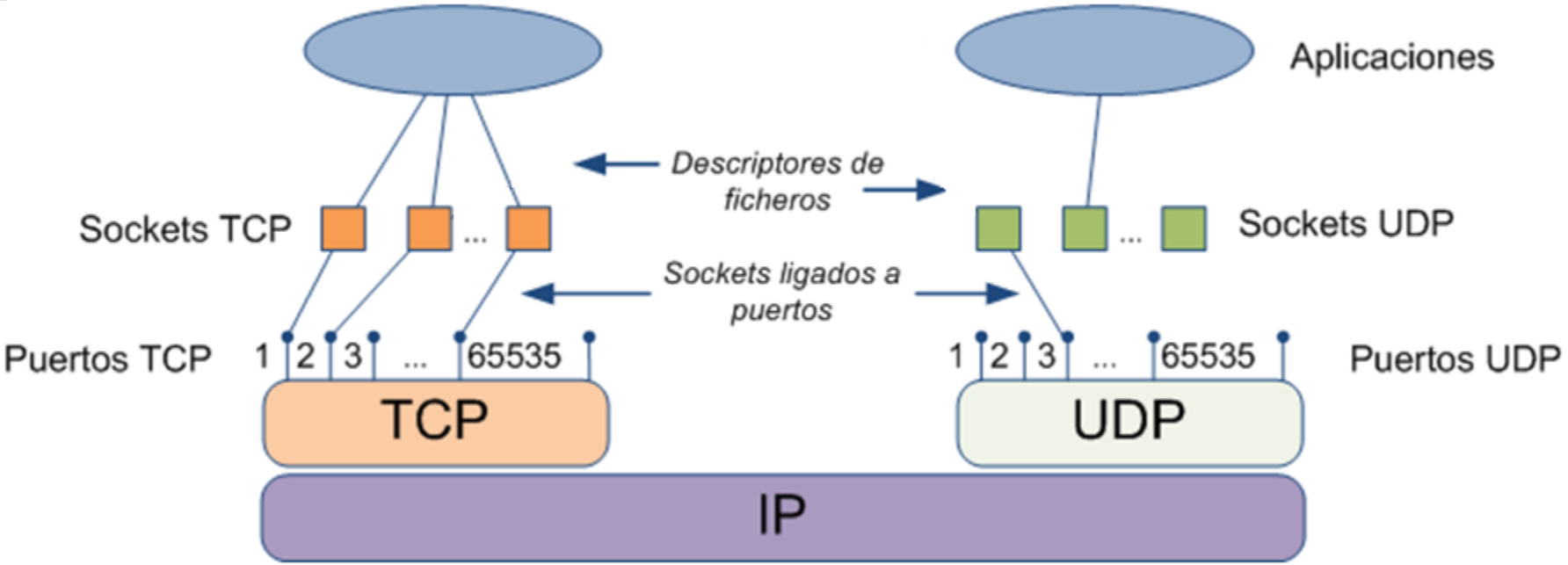
- Punto de comunicación por el que un proceso puede enviar o recibir información a/desde otro proceso que está en Internet
  - El canal de comunicación es **bidireccional** entre los dos procesos y lo proporciona el S.O.

- Los sockets son un conjunto de funciones
  - Para utilizar servicios de la capa de transporte (nivel 4) de TCP/IP (TCP, UDP...)
  - Es un estándar de facto (está presente en todos los sistemas operativos)

- Existen diferentes tipos de sockets dependiendo del protocolo a nivel de transporte que utilicen
  - TCP Stream sockets: se utilizan para intercambiar secuencias de bytes de forma fiable (no hay que preocuparse por errores introducidos por la red)
  - UDP Datagram sockets: se utilizan para intercambiar mensajes completos, pero sin garantizar fiabilidad (puede que la red pierda o desordene mensajes)

Sockets

- Las aplicaciones identifican los sockets utilizando descriptores de fichero



Sockets UDP y TCP

- Utilizar un socket **UDP** es similar a **mandar una carta**
  - No hace falta ninguna conexión previa con el destino, el mensaje se envía a su dirección utilizando los servicios de transporte (pero no hay garantía de si se recibirá o no)
  - Por cada nuevo envío, se tiene que indicar la dirección del destino
  - Por un mismo socket se puede enviar información a distintos destinatarios
  - Por un mismo socket se puede recibir información de distintos emisores (es un buzón)

- Utilizar un socket **TCP** es similar a realizar una **llamada telefónica**
  - El servidor tiene que estar esperando una solicitud de conexión
  - Solo hay que indicar la dirección del destinatario al realizar la conexión (una sola vez)
  - Un socket solo sirve como canal bidireccional entre dos procesos que están conectados (siempre los mismos mientras dure la conexión entre ellos)

Programación de sockets básica

1. Direcciones de sockets
2. Programación cliente/servidor con UDP
3. Programación cliente/servidor con TCP
4. Diseño de servidores

Programación de sockets básica

1. **Direcciones de sockets**
2. Programación cliente/servidor con UDP
3. Programación cliente/servidor con TCP
4. Diseño de servidores

Especificando direcciones

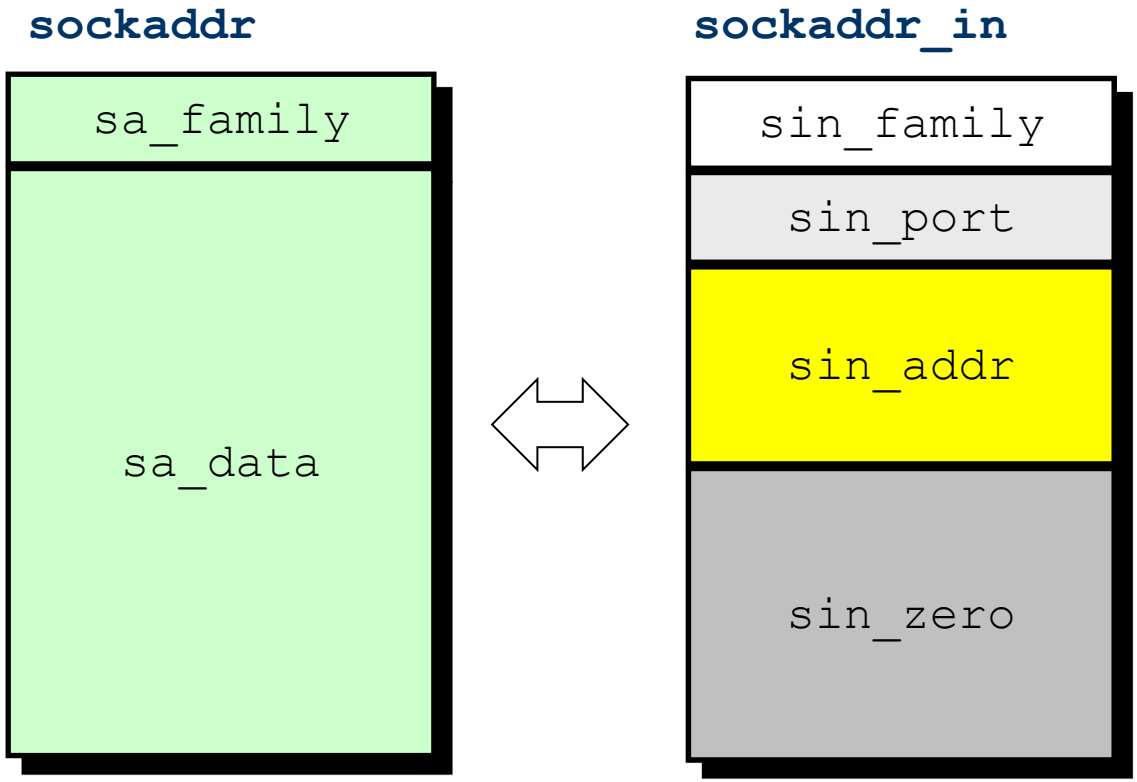
- Las aplicaciones que utilizan sockets necesitan poder especificar direcciones IP y puertos
  - Para hacer esta comunicación posible la API socket define la estructura genérica **sockaddr**

```
struct sockaddr {
  sa_family_t sa_family  ; // familia: AF_XXX
  uint8_t sa_data[14] ; // aquí está la dirección
};
```

  - Permite indicar el tipo de dirección en el campo **sa_family**
    - Si el valor de **sa_family** es **AF_INET**, entonces la estructura contendrá una dirección IP versión 4 y un puerto
  - El contenido del campo **sa_data[14]** dependerá del tipo de dirección

Direcciones de sockets para IPv4

- Las funciones de socket aceptan el tipo genérico de dirección **sockaddr**
- Pero por comodidad, el programador utiliza una versión especial, que se llama **sockaddr_in**, con los campos preparados para escribir direcciones IPv4 y puerto



Direcciones de sockets para IPv4

```
struct in_addr {
  uint32_t s_addr; // dirección IP
};

struct sockaddr_in {
  sa_family_t sin_family; // familia: AF_INET
  uint16_t  sin_port;     // número de puerto
  struct in_addr sin_addr; // dirección IP
  uint8_t   sin_zero[8];  // relleno, no usado
};
```

- Ejemplo:

```
struct sockaddr_in dir_socket;
memset(&dir_socket,0,sizeof(dir_socket));
dir_socket.sin_family = AF_INET;
dir_socket.sin_port = htons(80); // lo veremos!
uint8_t dir_IP[4] = {192, 168, 3, 1};
memcpy(&dir_socket.sin_addr, dir_IP, 4);
//La dirección genérica (que necesitan las funciones de
// sockets)se obtiene con un casting:
struct sockaddr * dir_generica =(struct sockaddr *)&dir_socket;
```

Formato de datos en direcciones

- Dada la problemática de la representación de información en diferentes arquitecturas, todos los campos de **struct sockaddr_in** han de almacenarse en formato big endian

```
dir_socket.sin_port = htons(80);
```

- Cualquier dato que ocupe más de un byte debe mandarse por un socket en formato **big-endian**
  - Esto no es aplicable a las cadenas de caracteres (cada carácter ocupa un byte)

- La dirección puede especificar IP, puerto, ambos o ninguno:
  - Sólo puerto: **`INADDR_ANY`**

Funciones auxiliares de manejo de direcciones (para IPv4)

- **inet_addr():** convierte una dirección IP con formato cadena de caracteres a un entero de 32 bits en formato **big-endian**

```
uint32_t dir = inet_addr("192.168.1.1");

struct sockaddr_in destino;
memcpy(&destino.sin_addr, &dir, 4);
```

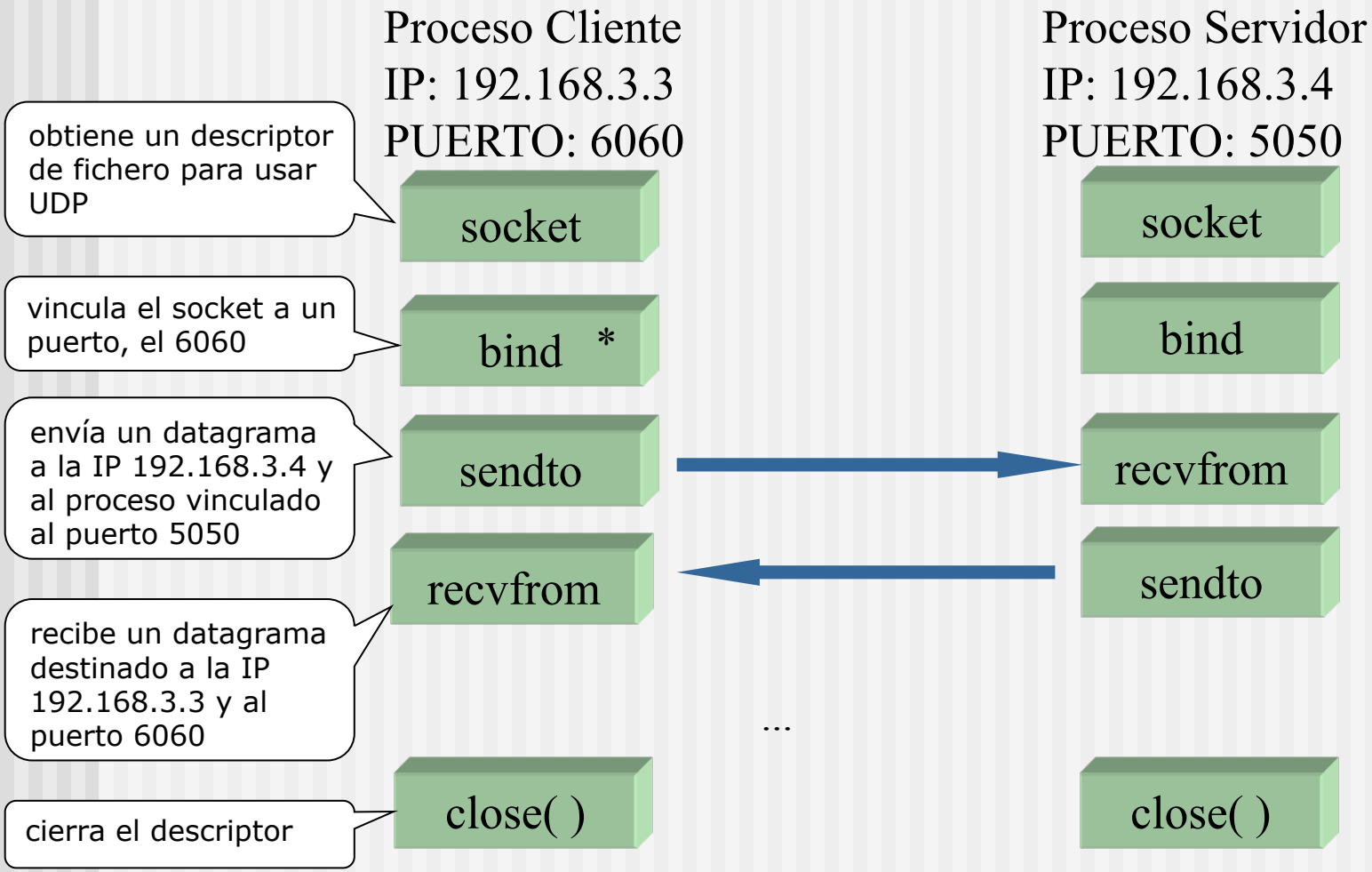
- **inet_ntoa():** convierte una dirección IP en formato entero de 32 bits (en big endian) a cadena de caracteres para imprimirse por pantalla.

```
printf("Dir IP: %s\n", inet_ntoa(destino.sin_addr))
```

Programación de sockets básica

1. Direcciones de sockets
2. **Programación cliente/servidor con UDP**
3. Programación cliente/servidor con TCP
4. Diseño de servidores

Secuencia de operaciones UDP



(*): En un cliente, bind() es opcional. Si no se usa, el sistema operativo asigna al socket un puerto efímero

Funciones: socket()

```
int socket(int familia, int tipo, int protocolo);
```

Crea un conector (socket) de un tipo específico y lo asocia a un descriptor de socket

- Familia de protocolo
  - Para TCP/IP: **PF_INET**
- Tipo
  - Orientado a la conexión: **SOCK_STREAM**
  - No orientado a la conexión: **SOCK_DGRAM**
- Protocolo
  - Si es **0**, se utiliza el protocolo TCP para SOCK_STREAM o UDP para SOCK_DGRAM
- Devuelve: el descriptor de socket abierto si hubo éxito en la creación, ó -1 si hubo error.
- Ejemplos:
  - Abrir un socket para usar UDP

    ```
    int sd = socket(PF_INET, SOCK_DGRAM, 0);
    ```
  - Abrir un socket para usar TCP

    ```
    int sd = socket(PF_INET, SOCK_STREAM, 0);
    ```

Funciones: bind()

```
int bind(int sd, sockaddr * dir, socklen_t longitud);
```

Vincula un puerto con la aplicación, así el nivel de transporte sabrá entregar los datos destinados a ese puerto a la aplicación

- **`int sd:`** descriptor abierto previamente con **socket()**
- **`sockaddr *dir:`** puntero a la estructura **sockaddr** que contiene el puerto (se rellenó en una estructura **sockaddr_in**)
- **`socklen_t longitud:`** Tamaño de la estructura de la dirección
- Devuelve:
  - **0**: en caso de éxito
  - **-1**: si hubo error
- Ejemplo:

```
sockaddr_in dir_puerto;
... //se rellena la estructura
int result = bind(sd, (struct sockaddr *)&dir_puerto,
    sizeof(dir_puerto));
```

Función de envío de datos en UDP

```
int sendto(
   int sd,              //Descriptor de socket
   char * datos,        //Puntero a los datos a enviar
   size_t longitud,     //Longitud de los datos a enviar
   int opciones,        //Opciones de envío (normalmente a 0)
   struct sockaddr * dir_destino,
                        //Puntero a la dirección del destinatario
   socklen_t longitud_dir    //Tamaño de la dirección
);
```

- Valor devuelto
  - El tamaño de los datos enviados (igual al tercer argumento)
  - -1, en caso de error

- Ejemplo:

```
char datos[512] = "Hola";
sockaddr_in dir_destino;
... //se rellena la estructura
 int bytesEnviados = sendto(sd, datos, 4, 0,
    (struct sockaddr *) &dir_destino,  sizeof(dir_destino));
```

Función de recepción de datos en UDP

```
int recvfrom(
   int sd,                  //Descriptor de socket
   char * datos,            //Dirección de memoria donde guardar los datos
   size_t longitud,         //Longitud máxima de los datos a recibir
   int opciones,            //Opciones de recepción (normalmente a cero)
   struct sockaddr *dir_origen,
                            //Puntero a la dirección del emisor
   socklen_t * longitud_dir
                            //Dirección de memoria con el tamaño de la
                            //estructura de dirección
);
```

- Devuelve:
  - el número de bytes recibidos en caso de éxito
  - -1 si hubo error

- Ejemplo:

```
char datos[512];
sockaddr_in dir_origen;
socklen_t longitud_direccion = sizeof(dir_origen);
int bytesRecibidos = recvfrom(sd, datos, 512, 0,
    (struct sockaddr *) &dir_origen, &longitud_direccion);
```

Función para cerrar el socket

```
int close(int sd);
```

- Libera los recursos asociados al socket en el nivel de transporte
- Al cerrar, ya no se pueden volver a realizar envíos ni recepciones (**sendto** y **recvfrom** devuelven -1)

Un ejemplo con UDP. Receptor

```
/*****************************************************************************/
/* FICHERO:     receptorUDP.c                                                */
/* DESCRIPCION: espera algún paquete en el puerto 4950 y muestra su contenido */
/*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define PUERTO         4950 // puerto para bind
#define MAXTAMBUFFER   100

int main(){

    int                descriptorSocket    ; /* descriptor del socket              */
    struct sockaddr_in direccionReceptor   ; /* dirección de socket del receptor   */
    struct sockaddr_in direccionEmisor     ; /* dirección de socket del emisor     */
    socklen_t          longitudDireccion   ; /* longitud de la dirección           */
    int                numeroDeBytes       ; /* número de bytes leidos             */
    char               buffer[MAXTAMBUFFER]; /* buffer de recepción de datos       */
```

Un ejemplo con UDP. Receptor

```
    /*creacion del socket UDP*/
    if ((descriptorSocket = socket(PF_INET, SOCK_DGRAM, 0)) == -1) {
          perror("socket");
          exit(1);
    }

    memset(&direccionReceptor, 0, sizeof(direccionReceptor));
    direccionReceptor.sin_family      = AF_INET       ;
    direccionReceptor.sin_port        = htons(PUERTO) ;
    direccionReceptor.sin_addr.s_addr = INADDR_ANY; /*escucha por cualquier interfaz de
    red activo del host (cualquier tarjeta que tenga una IP asignada)*/

    if (bind(descriptorSocket, (struct sockaddr *)&direccionReceptor,
             sizeof(direccionReceptor)) == -1) {
        perror("bind");
        exit(1);
    }

    longitudDireccion  = sizeof(direccionEmisor);
    if ((numeroDeBytes = recvfrom(descriptorSocket, buffer, MAXTAMBUFFER, 0,
                                  (struct sockaddr *)&direccionEmisor,
                                  (socklen_t *) &longitudDireccion)) == -1) {
        perror("recvfrom");
        exit(1);
    }
    printf("Paquete recibido de  %s\n",inet_ntoa(direccionEmisor.sin_addr));
    printf("El paquete tiene %d bytes de longitud\n",numeroDeBytes);
    buffer[numeroDeBytes] = '\0';
    printf("El paquete contiene esta cadena: %s\n",buffer);
    close(descriptorSocket);
    return 0;
} /* main */
```

Un ejemplo con UDP. Emisor

```
/*****************************************************************************/
/* FICHERO:     emisorUDP.c                                                  */
/* DESCRIPCION: envia un datagrama al puerto 4950                            */
/*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define PUERTO 4950    // puerto donde escucha del receptor

int main(int argc, char *argv[]){
   int                descriptorSocket  ; /* descriptor del socket   */
   struct sockaddr_in direccionReceptor ; /* dirección del receptor  */
   int                numeroDeBytes     ; /* número de bytes leidos  */

   if (argc != 3) {
      fprintf(stderr,"uso: emisor dir_ip mensaje\n");
      exit(1);
   }
```

Un ejemplo con UDP. Emisor

```
    /* para obtener la IP del receptor */
    uint32_t ip_receptor = inet_addr(argv[1]);
    if(ip_receptor == -1){
        perror("inet_addr");
        exit(1);
     }
    /* creacion del socket UDP */
     descriptorSocket = socket(PF_INET, SOCK_DGRAM, 0);
    if (descriptorSocket == -1) {
        perror("socket");
        exit(1);
    }
    memset(&direccionReceptor, 0, sizeof(direccionReceptor));
     direccionReceptor.sin_family = AF_INET;
    direccionReceptor.sin_port = htons(PUERTO);
    memcpy(&direccionReceptor.sin_addr,&ip_receptor, 4);

     numeroDeBytes = sendto(descriptorSocket, argv[2], strlen(argv[2]),0,
     (struct sockaddr *)&direccionReceptor, sizeof(direccionReceptor));
    if(numeroDeBytes == -1){
        perror("sendto");
        exit(1);
    }
    printf("Enviados %d bytes a %s\n", numeroDeBytes,
          inet_ntoa(direccionReceptor.sin_addr));
    close(descriptorSocket);
    return 0;
}
```

Ejemplo con UDP. Compilación y ejecución

- Compilación
  - Servidor (Linux):

    gcc -std=c99 -Wall receptorUDP.c -o receptorUDP
  - Cliente (Linux):

    gcc -std=c99 -Wall emisorUDP.c -o emisorUDP

- Ejecución
  - Ejecutar como

    ./receptorUDP
  - Ejecutar en la misma máquina del receptor

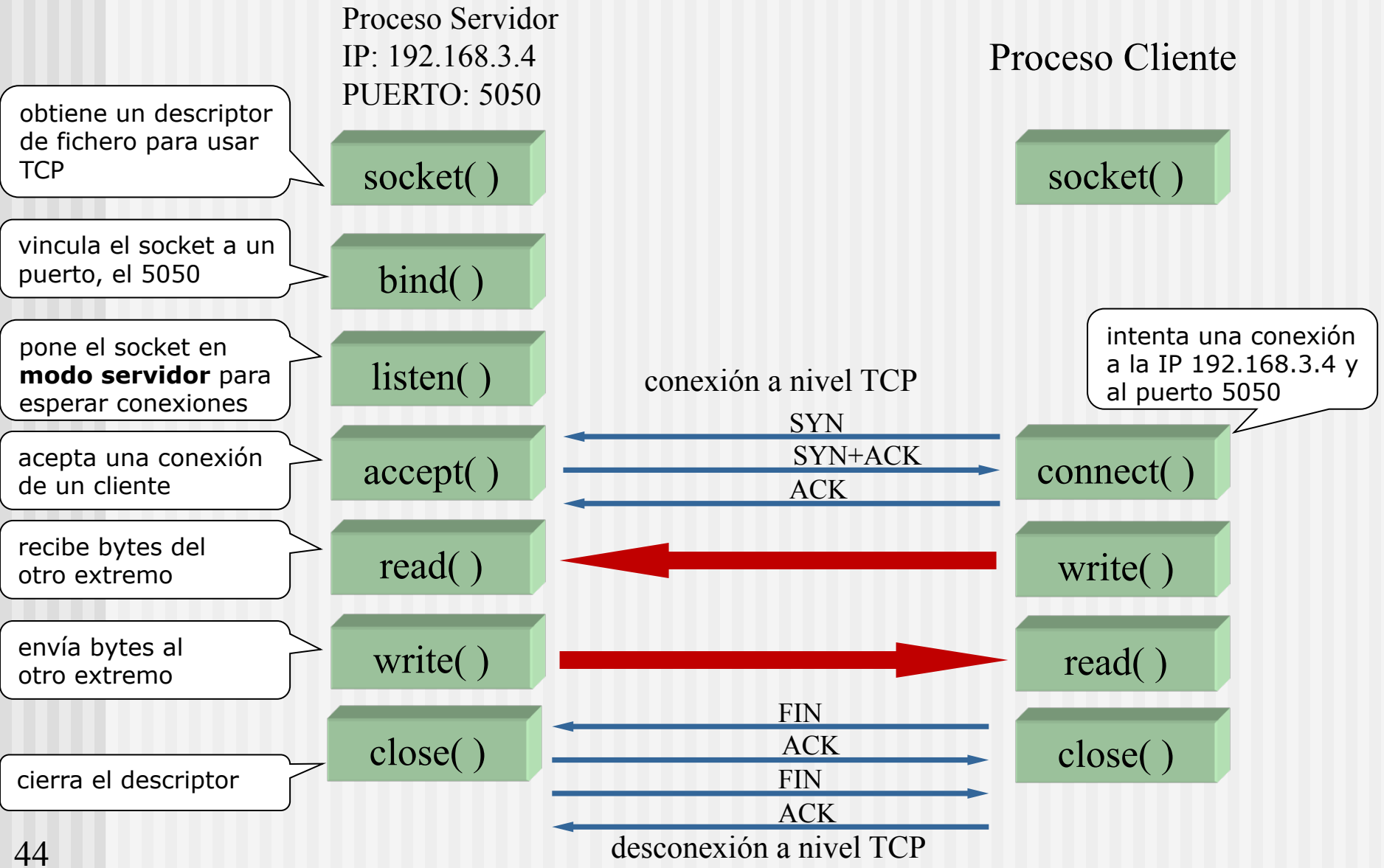
    ./emisorUDP 127.0.0.1 "mensaje al receptor"

    (NOTAS):
    - La dirección 127.0.0.1 es la dirección de loopback, que existe en la propia máquina, y se utiliza para hacer pruebas locales con clientes y servidores, sin que salgan paquetes a la red
    - Un mensaje que contenga espacios en blanco debe ir encerrado entre comillas, porque si no, el programa entenderá cada palabra como un argumento distinto

Programación de sockets básica

1. Direcciones de sockets
2. Programación cliente/servidor con UDP
3. **Programación cliente/servidor con TCP**
4. Diseño de servidores

Secuencia de operaciones TCP



Funciones: listen()

```
int listen(int sd, int longitud_cola_conexiones);
```

- Pone el socket en **modo servidor**, lo que a nivel de transporte se traduce en que el socket está preparado para **esperar conexiones** de clientes
- Primer argumento
  - Descriptor del socket (que debe estar **ya vinculado** a un puerto)
- Segundo argumento
  - Longitud de la cola de clientes que pueden estar esperando a ser atendidos (si un cliente intenta conectarse al servidor y se encuentra la cola llena, la conexión no se puede establecer y se rechaza inmediatamente)
  - El valor máximo suele ser una constante del sistema operativo como SOMAXCONN
- Valor devuelto
  - **0**, si todo es correcto
  - **-1**, si hay algún error

Funciones: accept()

```
int accept(
   int sd,                              //Descriptor de socket
   struct sockaddr *direccion, //Dirección del cliente
   socklen_t * longitud_dir
                        //Dirección de memoria con el tamaño
                        //de la dirección
);
```

- Un socket (en modo servidor) acepta una conexión de un cliente
  - El resultado devuelto es un **nuevo descriptor de socket** con las mismas propiedades que el descriptor anterior, o -1 si hubo error
    - El nuevo descriptor identifica la conexión con el cliente, y sobre él se pueden hacer envíos y recepciones de datos
    - El descriptor original NO puede utilizarse para escribir y leer datos, solo sirve para aceptar nuevas conexiones de clientes

Funciones: connect()

```
int connect(
   int sd,                  //Descriptor de socket
   struct sockaddr * dir_destino,
                            //Puntero a la dirección del servidor
   socklen_t longitud_dir
                            //Tamaño de la dirección del servidor
);
```

- Función que utiliza un cliente para establecer una conexión TCP con un servidor identificado por su IP y puerto

- Devuelve:
  - 0 en caso de éxito
  - -1 si hubo error

Funciones para envio de datos

```
int send(int sd, char *datos, size_t longitud_datos, int opciones_envio);

int write(int sd, char *datos, size_t longitud_datos);
```

- Envío de bytes a través de una conexión TCP. Se puede utilizar un **`write()`** del sistema operativo o **`send()`**, que permite opciones adicionales en el envío
- Primer argumento
  - Descriptor de un socket ya conectado a nivel de transporte
- Segundo argumento
  - Dirección de memoria donde empiezan los bytes a enviar al otro extremo
- Tercer argumento
  - Número de bytes a enviar
- Cuarto argumento (en la función send())
  - Opciones de envío (datos urgentes, etc.). Para envíos normales, se pone a cero
- Valor de retorno:
  - Se devuelve el número real de bytes escritos en el nivel de transporte TCP (del sistema operativo). NOTA: Esto no significa que se hayan entregado esos bytes al destino inmediatamente: TCP puede almacenarlos temporalmente en un buffer hasta que crea conveniente enviarlos al destino en un segmento TCP.
  - Si el valor devuelto por la función no coincide con el tercer argumento, entonces hay que reintentar enviar los bytes que faltan
  - Si devuelve -1, hubo un error (por ejemplo, cuando se intentan enviar datos y el otro extremo cerró la conexión).
- NOTA: si se intenta escribir por un socket cuya conexión ha sido cerrada previamente por alguno de los extremos, el sistema operativo puede lanzar la señal SIGPIPE que, por defecto, termina con el programa. Una solución posible es utilizar en el programa: **`signal(SIGPIPE,SIG_IGN);`**

Funciones para recepción de datos

```
int recv(int sd, char * datos, size_t longitud_datos, int opciones_recepcion);

int read(int sd, char * datos, size_t longitud_datos);
```

- Recepción de bytes a través de una conexión TCP

- Primer argumento
  - Descriptor de socket
- Segundo argumento
  - Dirección de memoria donde se van a almacenar los bytes leídos del nivel de transporte
- Tercer argumento
  - Número de bytes a leer que, como máximo, se puedan almacenar en la memoria
- Cuarto argumento (en send())
  - Opciones de recepción. Por defecto se pone a cero

- Valor de retorno:
  - Si el valor de retorno es mayor que cero: devuelve el número de bytes leidos realmente. Este valor puede no coincidir con el indicado como tercer argumento
  - **Si el valor de retorno es cero: indica que el otro extremo cerró la conexión y, por tanto, ya no se van a volver a recibir datos de él**
  - Si el valor de retorno es -1 es que ha habido un error

Funciones: cierre de conexión

```
int close(int sd);
```

- Cierra el socket
  - Libera los recursos asociados al mismo en el sistema operativo
  - Si era un socket conectado, se produce la desconexión a nivel de transporte entre los dos extremos, y ya no se pueden realizar envíos ni recepciones posteriores

Ejemplo. Cliente y Servidor de fechas

- La aplicación se compone de tres ficheros:
  - servidor.c: código del servidor
  - cliente.c: código de un cliente
  - defs.h: fichero de cabecera, con la estructura de información compartida entre el cliente y el servidor
- Compilación
  - Servidor (Linux):

    ```
    gcc -std=c99 -Wall servidor.c -o servidor
    ```
  - Cliente (Linux):

    ```
    gcc -std=c99 -Wall cliente.c -o cliente
    ```
- Ejecución
  - Ejecutar en la máquina

    ```
    ./servidor &
    ```
  - Ejecutar en la misma máquina (las veces que se quiera)

    ```
    ./cliente
    ```
- **ADVERTENCIA: Estos dos programas contienen erratas a propósito con fines didácticos**

Servidor de fechas. Fichero de cabecera

```
/*****************************************************************/
/* FICHERO:     defs.h                                           */
/* DESCRIPCION: fichero de cabecera para hacer uso de un servidor */
/*              de fechas                                        */
/*****************************************************************/
#ifndef __DEFS_H__
#define __DEFS_H__

const char direccion_ip[] = "127.0.0.1" ; /*localhost*/
const char puerto_tcp[]   = "7010" ;

#define SOLICITUD_FECHA  1
#define RESPUESTA_FECHA  2

struct FechaYHora {
   short int dia     ;
   short int mes     ;
   short int anno    ;
   short int hora    ;
   short int minuto  ;
   short int segundo ;
} ;

#endif
```

Servidor de fechas. Servidor

```
/*************************************************************/
/* FICHERO:     servidor.c                                   */
/* DESCRIPCION: codigo de un servidor de fechas              */
/*************************************************************/
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include "defs.h"

int main() {
  struct sockaddr_in direccionServidor, direccionCliente ;
  int descSocket, nuevoDescSocket ;
  socklen_t longDirCliente;//longitud de la direccion del cliente
  int codigoOperacion ;
  /* creacion  del socket TCP */
  descSocket = socket(PF_INET, SOCK_STREAM, 0) ;
  if (descSocket < 0) {
     perror("no se ha podido abrir el socket") ;
     exit(1) ;
  }
  /* se pone a cero la direccion del servidor */
  memset(&direccionServidor, 0, sizeof(direccionServidor));
  direccionServidor.sin_family        = AF_INET;
  direccionServidor.sin_addr.s_addr   = INADDR_ANY;
  direccionServidor.sin_port          = htons(atoi(puerto_tcp));
```

Servidor de fechas. Servidor (II)

```
/*vincula el proceso con el puerto de escucha indicado en la var. direccionServidor */
if (bind(descSocket, (struct sockaddr *) &direccionServidor,
         sizeof(direccionServidor))< 0) {
   perror("error al vincular la direccion local") ;
   exit(1) ;
}
/* pone el socket en modo pasivo (ahora puede aceptar conexiones) */
listen(descSocket, 5); /*cinco clientes pueden esperar su turno en cola*/
while(1){
  printf("Servidor esperando conexion ...\n");
  /* espera una conexion del cliente, devuelve dicha conexion en nuevoDescSocket */
  longDirCliente=sizeof(direccionCliente);/* longitud de la direccion del cliente */
  nuevoDescSocket = accept(descSocket, (struct sockaddr *) &direccionCliente ,
                           &longDirCliente);
  if(nuevoDescSocket < 0) {
     fprintf(stderr, "SERVIDOR: error al aceptar nueva conexion \n") ;
     exit(1) ;
  }
  recv(nuevoDescSocket, &codigoOperacion, sizeof(codigoOperacion),0) ;
  switch(codigoOperacion) {
  case SOLICITUD_FECHA : {
    time_t fechaActual = time(0) ; /*obtiene el tiempo*/
    struct tm * fechaPtr = gmtime(&fechaActual); /*se convierte a una estructura tm*/
    struct FechaYHora  fecha  ;
    fecha.dia     = fechaPtr->tm_mday ;
    fecha.mes     = fechaPtr->tm_mon + 1 ; /* porque devuelve de 0 a 11 */
    fecha.anno    = fechaPtr->tm_year + 1900;/*devuelve el transcurrido desde 1900 */
    fecha.hora    = fechaPtr->tm_hour ;
    fecha.minuto  = fechaPtr->tm_min  ;
    fecha.segundo = fechaPtr->tm_sec  ;
```

Servidor de fechas. Servidor (y III)

```
      codigoOperacion = RESPUESTA_FECHA ;
      send(nuevoDescSocket, &codigoOperacion, sizeof(codigoOperacion),0) ;
      send(nuevoDescSocket, &fecha , sizeof(fecha),0) ;
      break ;
    }
    default: {
       fprintf(stderr, "SERVIDOR: mensaje no valido: %d\n", codigoOperacion);
       exit(1) ;
    }
    } /* fin del switch */
  } /* fin del while */
} /* fin del main */
```

Servidor de fechas. Cliente

```
/*****************************************************************/
/* FICHERO:     cliente.c                                        */
/* DESCRIPCION: cliente hace uso del servidor de fechas          */
/*****************************************************************/
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "defs.h"

int main() {
  struct sockaddr_in  direccionServidor ;
  int descSocket; /* descriptor del socket */
  int codigoOperacion ;
  struct FechaYHora   fecha;

  memset(&direccionServidor, 0, sizeof(direccionServidor)); /*la pone a cero*/
  direccionServidor.sin_family         = AF_INET;
  direccionServidor.sin_addr.s_addr    = inet_addr(direccion_ip);
  direccionServidor.sin_port           = htons(atoi(puerto_tcp));

  /*creacion del socket TCP*/
  if ((descSocket = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    perror("Cliente: no se ha podido abrir el socket") ;
    exit(1);
  }
```

Servidor de fechas. Cliente

```
  /*conexion al servidor*/
  if (connect(descSocket,(struct sockaddr *) &direccionServidor,
                                              sizeof(direccionServidor)) < 0) {
    perror("no se ha podido conectar con servidor") ;
    exit(1) ;
  }
  printf("Cliente: conexion establecida\n") ;

  codigoOperacion = SOLICITUD_FECHA ;
  /*solicita la fecha al servidor*/
  send(descSocket, &codigoOperacion, sizeof(codigoOperacion),0) ;
  /*espera respuesta*/
  recv(descSocket, &codigoOperacion, sizeof(codigoOperacion),0) ;

  if (codigoOperacion == RESPUESTA_FECHA) {
    /*recibe la fecha*/
    recv(descSocket, &fecha, sizeof(fecha),0) ;
    printf("Fecha: %d-%d-%d ", fecha.dia , fecha.mes   , fecha.anno   ) ;
    printf("Hora:%d-%d-%d\n",  fecha.hora, fecha.minuto, fecha.segundo) ;
  }
  else {
    fprintf(stderr, "CLIENTE: Error al recibir->") ;
    fprintf(stderr, "mensaje desconocido: %d\n", codigoOperacion) ;
    exit(1) ;
  }

  return 0;
} /* main */
```

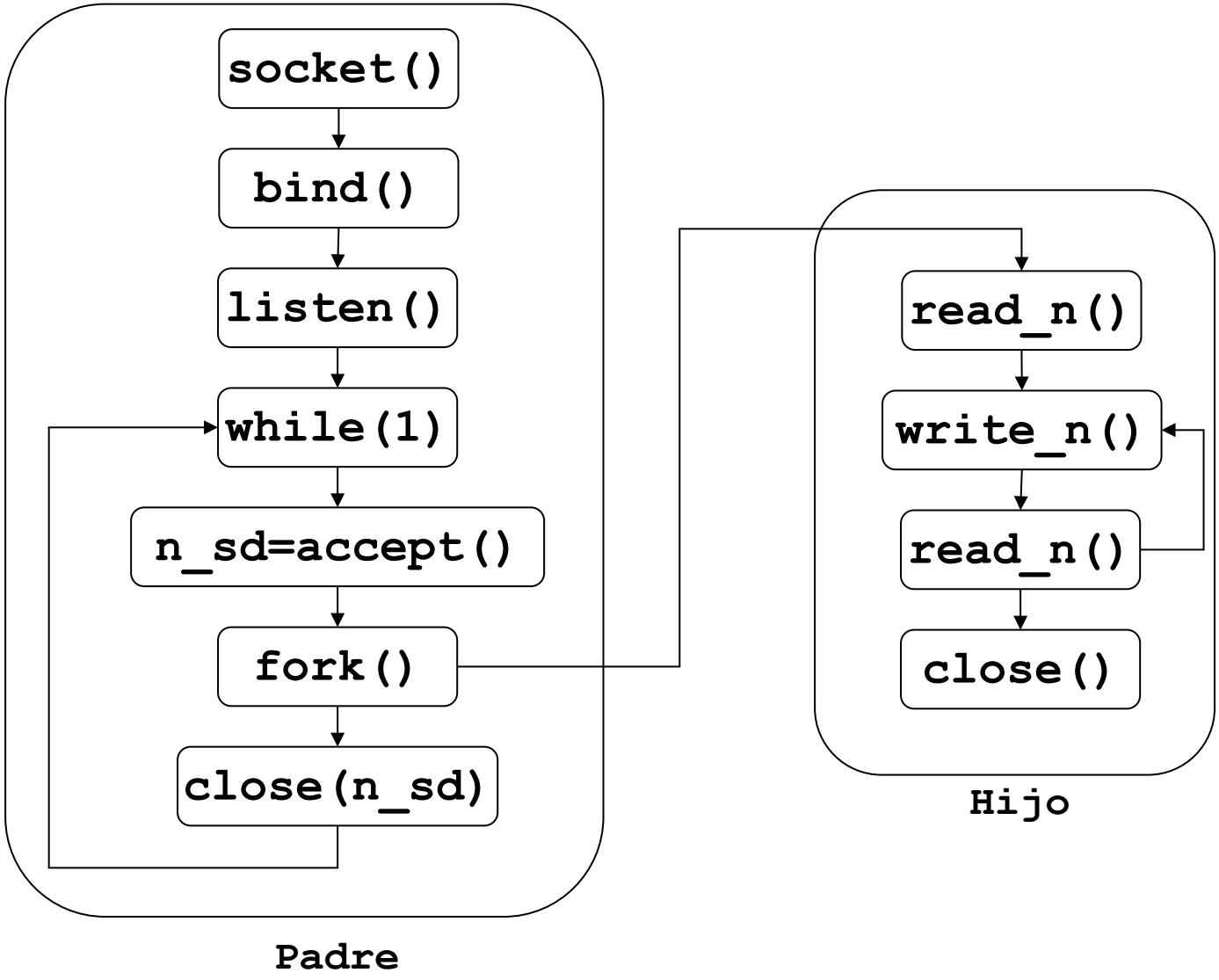
Programación de sockets básica

1. Direcciones de sockets
2. Programación cliente/servidor con UDP
3. Programación cliente/servidor con TCP
4. **Diseño de servidores**

Diseño de servidores

- Hasta ahora, un servidor solo podía atender a un cliente a la vez
  - Estos servidores se conocen como iterativos
- No es la mejor manera de aprovechar las características multiusuario/multiproceso del S.O.
  - Los clientes se pueden impacientar esperando a ser servidos
- Para atender a más de un cliente de forma simultánea se proponen estas dos técnicas:
  - servidor multiproceso (fork para que un hijo atienda a cada cliente)
    - Más pesado
    - Tiempo de atención a los clientes largo
  - servidor con multiplexación de E/S: el mismo proceso atiende a todos los clientes
    - Más ligero
    - Tiempo de atención a los clientes corto

Servidores multiproceso con fork()



Terminación ordenada de los hijos

- Este diseño es un caso típico donde se evitan zombies con uso de un manejador de señal para **SIGCHLD** que llame a **wait(0)**
- Problemas cuando se compila con **std=c99**
  - Si el padre está bloqueado en el **accept()**, cuando llega **SIGCHLD**, la función se interrumpe (al igual que **read()**) y **errno = EINTR**
  - Solución: **iterar**

```
errno=0;
n_sd = accept(sd, (struct sockaddr *) &cli_addr, &addr_len );
while (n_sd < 0) {
    if (errno != EINTR) {
        perror("accept");
        close(sd);
        exit(1);
    }
    n_sd = accept(sd, (struct sockaddr *) &cli_addr, &addr_len );
}
```

  - Se llama a **accept()** siempre que devuelva **-1** y **errno** sea igual a **EINTR** (interrumpido por la llegada de una señal)
  - Cualquier otro error provoca que la salida del servidor

Servidores multiproceso con fork

```c
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <signal.h>

#define PUERTO 3490

int main(){
    int sockfd; /* descriptor de socket de conexiones*/
    int n_sd; /* descriptor de socket para E/S con clientes */
    struct sockaddr_in server_addr;
    struct sockaddr_in client_addr;
    int sin_size = sizeof(client_addr);/* tamaño de la escructura sockaddr_in */

    signal(SIGCHLD, SIG_IGN); /*evita zombies */
    if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1){
        perror("socket");
        exit(1);
    }
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PUERTO);
    server_addr.sin_addr.s_addr = INADDR_ANY;
```

Servidores multiproceso con fork (II)

```
29
30      if (bind(sockfd, (struct sockaddr *)&server_addr, sizeof(server_addr)) == -1){
31          perror("bind");
32          exit(1);
33      }
34      if (listen(sockfd, 10) == -1){
35          perror("listen");
36          exit(1);
37      }
38      while(1){
39          if ((n_sd = accept(sockfd, (struct sockaddr *)&client_addr,  &sin_size)) == -1){
40             perror("accept");
41             break; /*sale del bucle while  */
42          }
43          printf("conexion desde: %s\n", inet_ntoa(client_addr.sin_addr));
44          pid_t p = fork();
45          if(p == 0){ /*codigo del hijo*/
46             write(n_sd, "hola cliente!\n", 15);
47             close(n_sd);
48             exit(0);
49          }else if(p == -1){
50             perror("error en fork");
51             break; /*sale del bucle while  */
52          }
53      }
54      /*fin del servidor*/
55      close(sockfd);
56      return 0;
57  }
```

Servidores con multiplexación de E/S

- Para hacer un servidor en un solo proceso que atienda simultáneamente a más de un cliente (por distintos sockets) existe un problema:
  - las llamadas a **`recv()`** y **`accept()`** son bloqueantes. ¿Cómo se puede atender a un cliente o a otro cuando lleguen datos o nuevas solicitudes de conexión?
  - Solución: utilizar la función **`select()`**
    - Sea **`sd`** el descriptor sobre el que ha hecho **`bind`** y **`listen`**
    - Se añade **`sd`** al conjunto de descriptores de lectura con **`FD_SET`**
    - Comienza un bucle infinito
    - Se llama a **`select()`** hasta que aparezca una nueva conexión de un cliente, momento en el que se desbloquea el proceso
    - Si **`FD_ISSET`** confirma que **`sd`** está listo para lectura, entonces se hace **`accept()`**. Supóngase que devuelve un nuevo descriptor conectado con el cliente, de **`nombre n_sd`**
    - Se añade **`n_sd`** al conjunto de descriptores y se vuelve a llamar a **`select()`**
    - Si **`FD_ISSET`** confirma que **`n_sd`** está listo para lectura, entonces se hace **`read()`**. Si **`read()`** devuelve cero, entonces se quita **`n_sd`** del conjunto de descriptores con **`FD_CLR`** y se cierra el socket **`n_sd`**

Ejemplo de servidor con select()

```
/* ... */
/* Parte de un main de un proceso con varios sockets simultáneos de los que se
   pueden leer datos
*/

fd_set rfd, active_rfd;

/* supongamos que inicialmente tenemos dos sockets ya conectados: s1 y s2 */
FD_ZERO(&rfd);
FD_SET(s1, &rfd);
FD_SET(s2, &rfd);

/* se calcula el mayor valor de descriptor en el conjunto */
int max_fd = get_max_fd(s1,s2);

while(1){
   /*cada vez que se llama a select se crea una copia del conjunto original
     gracias a una funcion hecha a medida (ver tema 2, parte 3)*/
   copia_fdset(&active_rfd, &rfd, max_fd + 1);

   /* select comprueba si los descriptores están disponibles para lectura.
   Como no se ha establecido un timeout, se bloquea hasta que alguno esté
   disponible (tenga datos listos para ser leídos)*/
   rc = select(max_fd+1, &active_rfd, NULL, NULL, NULL);
```

Ejemplo de servidor con select()

```
    /* chequeo de error si rc < 0 ... */
    if(rc < 0){
         perror("select");
         close(s1); close(s2);
         exit(1);
    }
    /* si el socket s1 tiene datos disponibles para leer ... */
    if (FD_ISSET(s1, &active_rfd)) {
         read(s1, buffer, MAXBUFFSIZE);
         /* ... código asociado para dar servicio a s1 ...*/
    }
    /* si el socket s2 tiene datos disponibles para leer ... */
    if (FD_ISSET(s2, &active_rfd)) {
         read(f2, buffer, MAXBUFFSIZE);
         /* ... código asociado para dar servicio a s2...*/
    }
} /* while */
```