Tema 2: Interconexión de procesos

Programación con *sockets*

Jose Luis Díaz

Curso 2011-2012

Contenidos

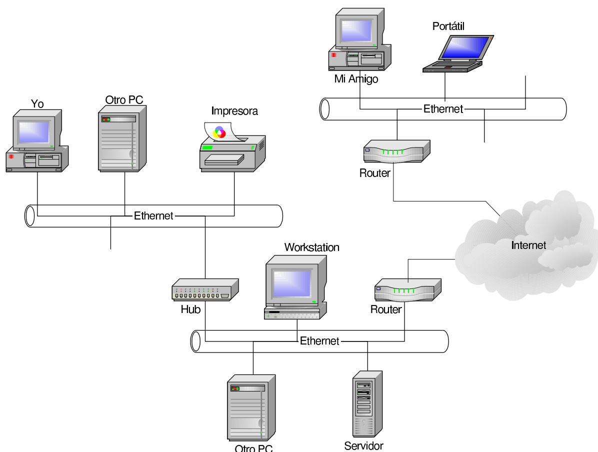
1. Introducción
   - Objetivo
   - Protocolos por capas
   - Sockets
   - Modelos de interacción
2. Sockets
   - Introducción
   - API de sockets en Unix
   - API de sockets en Windows
   - Otras funciones relacionadas con sockets
3. Programación de servidores
   - El problema de los clientes concurrentes
   - Solución creando varios procesos
   - Solución usando `select()`
4. Conclusiones
   - Ventajas
   - Inconvenientes

Esquema

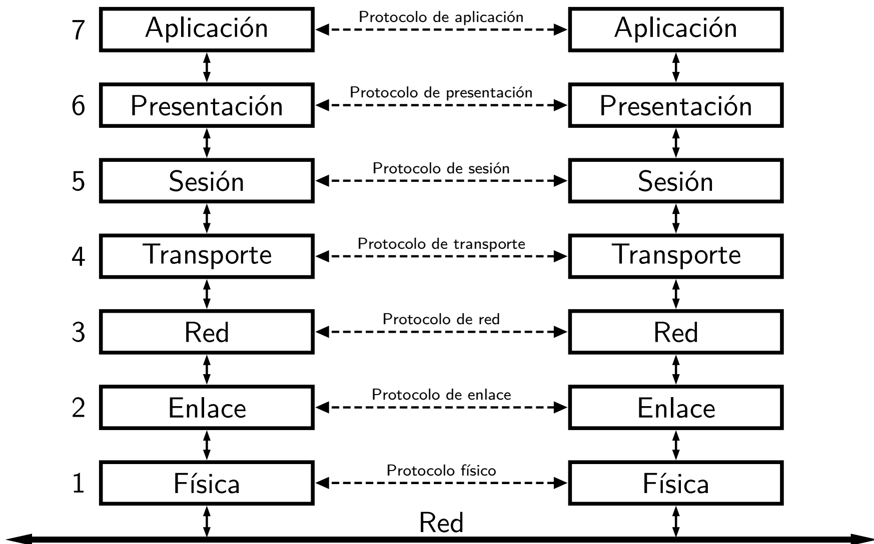
1. Introducción
   - Objetivo
   - Protocolos por capas
   - Sockets
   - Modelos de interacción
2. Sockets
3. Programación de servidores
4. Conclusiones

Objetivo

Se trata de comunicar dos procesos, a través de una red.



Modelo ISO/OSI



Capas física y de enlace

- La capa física transporta las señales, moduladas adecuadamente
- La capa de enlace
  - transmite pequeños paquetes de datos entre nodos unidos por una misma conexión física
  - identifica a los nodos por direcciones "físicas"
  - resuelve colisiones en el medio, control de errores en la transmisión
  - Ejemplo elementos: tarjetas de red Ethernet, hubs, switches, direcciones "MAC"

El modelo DoD agrupa estas dos capas: "Acceso a la red"

Capa de red

- La capa de red hace llegar paquetes de tamaño variable a nodos en diferentes segmentos.
- Identifica los nodos por direcciones lógicas.
- Se ocupa de traducir direcciones lógicas en físicas.
- Se ocupa de *enrutar*

En el modelo DoD, esta capa (“Inter-red”) implementa el *Protocolo de Internet* (IP)

Capa de transporte

- Incluye protocolos para garantizar aspectos que la capa de red no garantiza, como:
  - que los datos lleguen en el mismo orden en que fueron enviados
  - que todos los datos lleguen
  - detectar errores de transmisión
  - controlar el flujo
  - proporcionar conexión
  - identificar procesos (puertos) dentro de las máquinas

En el modelo DoD, esta capa ("Nodo-a-nodo") implementa los protocolos "TCP" y "UDP"

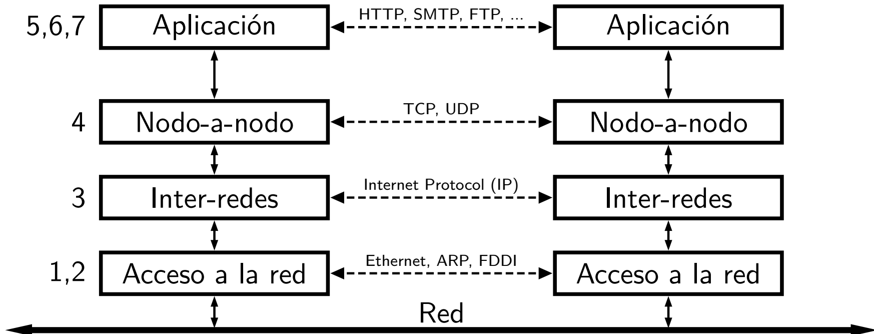
Capas superiores

- La capa de **sesión** maneja el diálogo entre procesos, y su sincronización
  - No es muy usado
  - Salvo en aplicaciones multimedia
- La capa de **presentación** maneja las diferentes codificaciones de información
- La capa de **aplicación** proporciona aplicaciones concretas (email, web, ftp...) Cada aplicación implementa sus propios protocolos

En el modelo DoD, estas capas se fusionan en una:
la "Capa de Aplicación"

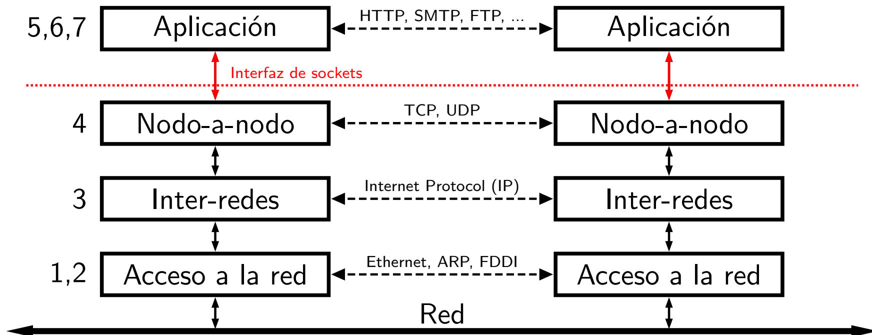
Modelo DoD (Departamento de Defensa)

También llamado TCP/IP, o "de Internet"



Sockets

El interfaz de *sockets* son un conjunto de funciones que se pueden llamar desde las aplicaciones para acceder a las capas de transporte y red.



Modelo cliente/servidor

- Un proceso (servidor) espera pasivamente a que otro se "conecte"
- Otro proceso (cliente) inicia la conexión
- Una vez establecida la conexión, cualquiera de los procesos puede enviar o recibir
- Generalmente el cliente envía "comandos de petición", y el servidor envía "respuestas" a esos comandos
- Ejemplo: servidor WEB y navegador

Modelo “igual a igual”

- Todos los procesos participantes son “iguales”
- Todos están preparados para recibir conexiones
- Cualquiera de ellos puede “conectar” con otro
- Es decir: todos pueden funcionar como cliente o servidor en diferentes momentos

Esquema

1 Introducción

2 Sockets
- Introducción
- API de sockets en Unix
- API de sockets en Windows
- Otras funciones relacionadas con sockets

3 Programación de servidores

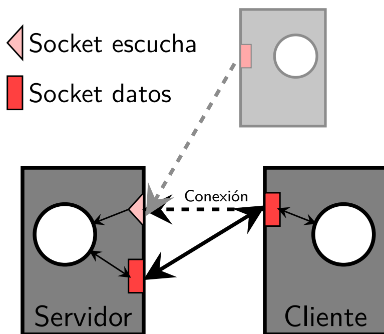
4 Conclusiones

¿Qué es un socket?

- Es una abstracción proporcionada por el software
- Representa un extremo de una conexión entre dos procesos
- Cada socket tiene una *dirección* (o nombre), que lo identifica, formada por
  - La dirección IP de la máquina
  - El número de *puerto*
- Los sockets soportan diferentes protocolos:

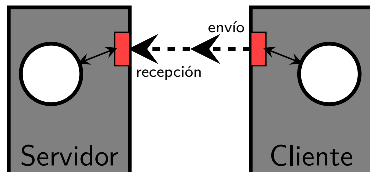
  TCP Orientado a conexión (tipo cliente/servidor)
  UDP Sin conexión (orientado a datagramas)

Sockets y el protocolo TCP



1. Dos procesos quieren comunicarse
2. El servidor crea un socket de escucha
3. El cliente crea un socket de datos
4. El cliente intenta conexión
5. El servidor acepta la conexión
    - Se crea un nuevo socket de datos en el servidor
    - Se establece un canal por el que pueden dialogar
6. El socket de escucha queda libre. Otros procesos pueden intentar conexión.
7. Cuando termina el diálogo, los sockets de datos se destruyen.

Sockets y el protocolo UDP (sin conexión)



1. Dos procesos quieren comunicarse
2. El servidor crea un socket de datos
3. El cliente crea un socket de datos
4. El cliente envía datos (no hay "canal")
5. El servidor recibe los datos
6. El servidor puede responder en la misma forma
7. Cuando termina el diálogo, los sockets de datos se destruyen.

Funciones (UNIX)

socket	Crea un socket
bind	Asigna una dirección al socket
listen	(TCP) Marca el socket como *de escucha*
connect	(TCP) Usada por el cliente para establecer conexión
accept	(TCP) Deja el servidor esperando conexiones. También crea un nuevo socket de datos en el servidor, cuando se recibe conexión
send, recv	(TCP) Comunicación
write, read	(TCP) Alternativa a **`send`** y **`recv`**
sendto, recvfrom	(UDP) Comunicación
close	Destruye el socket

Ficheros de cabecera

#include <sys/types.h> Tipos de datos básicos

#include <sys/socket.h> Prototipos de **`socket()`**, **`bind()`**, **`listen()`**, **`connect()`**, **`accept()`**, **`send()`** y **`recv()`**

Estructuras de datos para almacenar direcciones, etc.

#include <unistd.h> Prototipos de **`read()`**, **`write()`** y **`close()`**

socket()

```
int socket(int familia, int servicio, int protocolo)
```

familia
: Familia de protocolos a usar (**PF_INET** para los de Internet)

servicio
: Tipo de transporte a usar

  - **SOCK_STREAM** Para transporte TCP (orientado a conexión)
  - **SOCK_DGRAM** Para transporte UDP (orientado a datagramas)

protocolo
: Tipo de protocolo de red a usar (dar el valor 0 para usar protocolo IP)

Retorna
: Manejador del socket, o -1 (error)

bind()

```
int bind(int sock, struct sockaddr *direcc,
         socklen_t longitud)
```

sock El socket cuya dirección se quiere asignar

direcc Un puntero a una estructura que contiene la dirección

- Es una estructura genérica, válida para muchos tipos de dirección
- En el caso particular de direcciones de Internet (IPV4), la estructura se denomina **`sockaddr_in`**
- Su inicialización requiere de otras funciones (más adelante)

longitud Tamaño de la estructura que contiene la dirección

Retorna 0 (éxito) ó -1 (error)

struct\sockaddr_in

```
struct sockaddr_in {
    sa_family_t     sin_family;
    u_int16_t       sin_port;
    struct in_addr  sin_addr;
}
```

sin_family Familia de direcciones a usar
Especificar **AF_INET** para Internet (IPV4)

sin_port Número de puerto de la dirección
Debe estar en el "orden de red" (*Big-endian*)

sin_addr Es otra estructura que contiene un solo campo llamado **s_addr**, que contiene la dirección IP
4 bytes en el "orden de red"

Inicialización de la dirección

Ejemplo

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

struct sockaddr_in direccion;

direccion.sin_family=AF_INET;
direccion.sin_port=htons(80);
direccion.sin_addr.s_addr=
   inet_addr("156.35.33.99");
```

- Declaraciones de funciones **htons** e **inet_addr**
- Estructura para contener la dirección
- Familia de direcciones de Internet
- **htons** convierte un entero corto al orden de red
- **inet_addr** convierte de cadena ASCII a 4 bytes en orden de red.

`bind()` en el cliente

NOTA

No es necesario llamar a **`bind()`** en el cliente. Éste recibirá automáticamente una dirección cuando invoque **`connect()`**.
La dirección asignada automáticamente al cliente estará compuesta por la IP de su máquina más un puerto elegido aleatoriamente entre los no utilizados.

Funciones de conversión al "orden de red"

- Los datos relacionados con direcciones de red, deben estar en *Big-Endian*. En las variables del programa, en cambio, pueden estar en orden *little-endian* (depende de la máquina).
- Una serie de funciones permiten convertir del formato usado por la máquina (**h**ost), al usado por la red (**n**etwork), o viceversa
- Una letra final indica el tamaño del dato a convertir: 2 bytes (**s**hort), ó 4 bytes (**l**ong)

htons	**h**ost **to** **n**etwork **s**hort
ntohs	**n**etwork **to** **h**ost **s**hort
htonl	**h**ost **to** **n**etwork **l**ong
ntohl	**n**etwork **to** **h**ost **l**ong

Llamando a **bind()** sin especificardirección

En el campo **sin_addr.s_addr** se puede usar la constante **ADDR_ANY** en lugar de especificar una.

Ejemplo

```
direccion.sin_addr.s_addr = htonl(INADDR_ANY);
```

Usando esta constante:

- No es necesario conocer la IP del servidor al programarlo
- El servidor se puede llevar a otra máquina, sin necesidad de modificar su código
- Si el servidor tiene varias direcciones de red (ej. un *router*), admitirá conexiones en todas ellas

listen()

```
int listen(int sock, int backlog);
```

sock: El socket que se quiere poner en “modo escucha”

backlog: Tamaño de la cola en la que esperan otros clientes que han intentado conectar mientras el servidor estaba ocupado

Se recomienda pasar el valor `SOMAXCONN`

Retorna: 0 (éxito) ó -1 (error)

connect()

```
int connect(int sock, struct sockaddr *direccion,
            socklen_t longitud);
```

sock	El socket (local) que se quiere usar para establecer la conexión
direccion	La dirección del socket remoto con que se quiere conectar (IP y puerto)
longitud	Tamaño de la estructura que almacena la dirección
Retorna	0 (éxito) ó -1 (error)

La estructura `direccion` se inicializa en la misma forma que ya se vio para `bind()`.

accept()

```
int accept(int sock, struct sockaddr *direccion,
           socklen_t *longitud);
```

sock
: El socket donde se esperan conexiones. Debe ser un socket de escucha (**listen()**)

direccion
: Puntero a una estructura, en la que se dejará la dirección del cliente que ha conectado

longitud
: Puntero a un entero, en el que se dejará el tamaño de la estructura anterior

Retorna
: -1 en caso de error. En otro caso retorna **un nuevo socket** de datos, que deberá usarse en adelante para la comunicación con el cliente

Observar que los parámetros `direccion` y `longitud` son valores retornados por la función. No obstante el entero `*longitud` debe estar inicializado como `sizeof(struct sockaddr_in)`

send()

```
int send(int sock, void *datos, size_t tamanio, int opciones);
```

sock El socket para comunicar

datos Puntero a los datos a enviar. Se transmitirán los bytes sin modificar su orden ni interpretarlos.

tamanio Cantidad de bytes a transmitir

opciones Opciones de la transmisión (poner cero)

Retorna -1 en caso de error. En otro caso retorna el número de bytes enviados.

Si el valor devuelto no coincide con `tamanio`, es que no se han enviado todos los datos. Sería necesario otro `send` para enviar los restantes (más adelante se verá cómo)

recv()

```
int recv(int sock, void *datos, size_t tamanio, int opciones);
```

sock El socket para comunicar

datos Puntero a una zona de memoria donde se dejarán los bytes recibidos.

tamanio Cantidad de bytes a recibir

opciones Opciones de la transmisión (poner cero)

Retorna -1 en caso de error. En otro caso retorna el número de bytes recibidos. Si retorna 0 es indicativo de que el otro extremo ha cerrado el socket.

Esta función se *bloqueará* hasta que haya datos para leer. Si el valor devuelto no coincide con `tamanio`, es que no se han leido todos los datos. Sería necesario otro `recv` para leer los restantes (más adelante se verá cómo)

write()

Es análogo a **send()**, pero sin opciones de envío

```
int write(int sock, void *datos, size_t tamanio);
```

- sock: El socket para comunicar
- datos: Puntero a los datos a enviar. Se transmitirán los bytes sin modificar su orden ni interpretarlos.
- tamanio: Cantidad de bytes a transmitir
- Retorna: -1 en caso de error. En otro caso retorna el número de bytes enviados.

Ejemplo (sin control de errores)

```
int i=4;
...
write(sock, &i, sizeof(int)); // también sizeof(i)
```

read()

Es análogo a **recv()**, pero sin opciones.

```
int read(int sock, void *datos, size_t tamanio);
```

sock	El socket para comunicar
datos	Puntero a una zona de memoria donde se dejarán los bytes recibidos.
tamanio	Cantidad de bytes a recibir
Retorna	-1 en caso de error. En otro caso retorna el número de bytes recibidos. Si retorna 0 es indicativo de que el otro extremo ha cerrado el socket.

Ejemplo (sin control de errores)

```
int i;
...
read(sock, &i, sizeof(int)); // también sizeof(i)
```

Recordatorio

Recordar que las funciones **`send()`** y **`recv()`** (o sus equivalentes **`write()`** y **`read()`** sólo pueden usarse en sockets tipo TCP, tras haber establecido la conexión.

Para sockets tipo UDP debe usarse **`sendto()`** y **`recvfrom()`** en su lugar

sendto()

```
int sendto(int sock, void *datos, size_t tamanio,
           int opciones, struct sockaddr *destino,
           socklen_t longitud);
```

- **sock** El socket para comunicar
- **datos** Puntero a los datos a enviar. Se transmitirán los bytes sin modificar su orden ni interpretarlos.
- **tamanio** Cantidad de bytes a transmitir
- **opciones** Opciones de la transmisión (poner 0)
- **destino** Dirección del socket en el otro extremo
- **longitud** Tamaño de la estructura que contiene la dirección
- **Retorna** -1 en caso de error. En otro caso retorna el número de bytes enviados.

recvfrom()

```
int recvfrom(int sock, void *datos, size_t tamanio,
             int opciones, struct sockaddr *origen,
             socklen_t *longitud);
```

sock	El socket para comunicar
datos	Puntero a una dirección donde se dejarán los datos recibidos
tamanio	Cantidad de bytes a recibir
opciones	Opciones de la transmisión (poner 0)
origen	Puntero a una estructura donde se dejará la dirección del socket que nos envió datos
longitud	Puntero a un entero que, al retorno, indicará el tamaño de la estructura anterior
Retorna	-1 en caso de error. En otro caso retorna el número de bytes recibidos.

close()

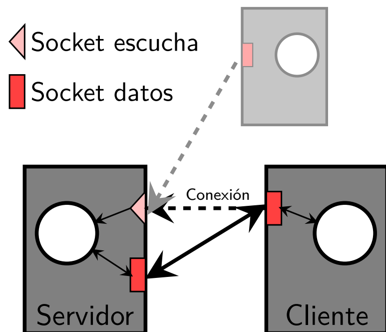
```
int close(int sock);
```

sock El socket a cerrar (destruir). Puede ser un socket de escucha o de datos.

Retorna 0 (éxito) ó -1 (error)

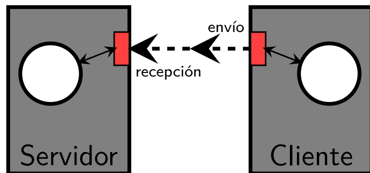
Una vez cerrado el socket, si era de datos ya no podrá enviar o recibir nada. Si era de escucha ya no podrá aceptar más conexiones.

Sockets TCP, secuencia de funciones



1. Los procesos arrancan
2. Servidor: **`socket()`**, **`bind()`**, **`listen()`**
3. Cliente: **`socket()`**
4. Cliente: **`connect()`**
5. Servidor: **`accept()`**
   Se comunican con **`write/read`** o **`send/recv`**
6. Otros clientes **`connect()`**
7. Ambos: **`close()`**

Sockets UDP, secuencia de funciones



1. Los procesos arrancan
2. Servidor: `socket()`, `bind()`
3. Cliente: `socket()`
4. Cliente: `sendto()`
5. Servidor: `recvfrom()`
6. Servidor: `sendto()`,
   Cliente: `recvfrom()`
7. Ambos: `close()`

Ejemplo TCP: servidor

```
#include <sys/socket.h>      // socket, send, recv...
#include <netinet/in.h>      // sockaddr_in, htons, htonl, etc.
#include <arpa/inet.h>       // inet_addr
#include <unistd.h>          // close
#include <stdlib.h>          // exit
#include <stdio.h>           // printf, fprintf...
#include <errno.h>           // errno
#include <string.h>          // strerror

#define PUERTO 8889
#define TAM_RECIBE 75
#define TAM_ENVIA  75

int main(void)
{
  int result, LongDirRemota, Res;
  int SocEscucha, SocDatos;
  struct sockaddr_in DirLocal, DirRemota;
  char peticion[TAM_RECIBE];
  char respuesta[TAM_ENVIA];
```

sigue ➤➤

Ejemplo TCP: servidor

➤➤ *sigue*

```
// Crear el socket para escuchar peticiones
SocEscucha = socket(PF_INET,SOCK_STREAM,0);
if (SocEscucha == -1) {
  fprintf(stderr,"Error %d en socket(): %s\n",
          errno, strerror(errno));
  exit(-1); }

// Enlazar el socket a una direccion local (cualquiera)
DirLocal.sin_family = AF_INET;
DirLocal.sin_port = htons(PUERTO);
DirLocal.sin_addr.s_addr = htonl(INADDR_ANY);
result = bind(SocEscucha,(struct sockaddr *)&DirLocal,sizeof(DirLocal));
if (result == -1) {
  fprintf(stderr,"Error %d en bind(): %s\n",
          errno, strerror(errno));
  close(SocEscucha); exit(-1); }

// Hacer el socket de escucha
result = listen(SocEscucha,SOMAXCONN);
```

sigue ➤➤

Ejemplo TCP: servidor

➤➤ *sigue*

```
if (result == -1) {
  fprintf(stderr,"Error %d en listen(): %s\n",
          errno, strerror(errno));
  close(SocEscucha); exit(-1); }

// Esperar por una conexión:
LongDirRemota = sizeof(struct sockaddr);
printf("Esperando conexion...\n");
SocDatos = accept(SocEscucha,(struct sockaddr *)&DirRemota,&LongDirRemota);
if (SocDatos == -1) {
  fprintf(stderr,"Error %d en accept(): %s\n",errno, strerror(errno));
  close(SocEscucha); exit(-1); }
printf("Conectado cliente desde %s\n", inet_ntoa(DirRemota.sin_addr));

// Recibir una peticion (cadena) del cliente
result = recv(SocDatos, peticion, sizeof(peticion), 0);
if (result == -1) {
  fprintf(stderr,"Error %d en recv(): %s\n",
          errno, strerror(errno));
  close(SocDatos); close(SocEscucha); exit(-1); }
```

sigue ➤➤

Ejemplo TCP: servidor

➤➤ *sigue*

```
printf("Recibo %d caracteres: %s\n", result, peticion);

// Enviar una respuesta (cadena) al cliente
sprintf(respuesta,"Me has enviado %d bytes\n", result);
result = send(SocDatos, respuesta, sizeof(respuesta), 0);
if (result == -1) {
  fprintf(stderr,"Error %d en send(): %s\n",
          errno, strerror(errno));
  close(SocDatos); clos(SocEscucha); exit(-1); }

// Fin
close(SocDatos);
close(SocEscucha);
return 1;
} // main
```

Ejemplo TCP: cliente

```
// Los mismos #include que en el servidor
#define PUERTO 8889
#define TAM_RECIBE 75
#define TAM_ENVIA  75
int main(int argc, char *argv[])
{
  int result, Pet;
  int SocConexion;
  struct sockaddr_in DirServidor;
  char Peticion[TAM_RECIBE], Respuesta[TAM_ENVIA];
  char *strDir;

  if (argc<2) {
    printf("Direccion remota no especificada. Usando localhost\n");
    strDir="127.0.0.1"; }
  else strDir=argv[1];

  // Creación del socket de conexión
  SocConexion = socket(PF_INET,SOCK_STREAM,0);
  if (SocConexion == -1) {
```

sigue ➤➤

Ejemplo TCP: cliente

➤➤ *sigue*

```
    fprintf(stderr,"Error %d en socket(): %s\n",errno, strerror(errno));
    close(SocConexion); exit(-1); }

  // Especificacion de la direccion del servidor
  DirServidor.sin_family = AF_INET;
  DirServidor.sin_port = htons(PUERTO);
  DirServidor.sin_addr.s_addr = inet_addr(strDir);

  // Conectar el socket
  result = connect(SocConexion,
                   (struct sockaddr *)&DirServidor,sizeof(DirServidor));
  if (result == -1){
    fprintf(stderr,"Error %d en connect(): %s\n",errno, strerror(errno));
    close(SocConexion); exit(-1); }

  // Enviar una peticion (cadena) al servidor
  sprintf(Peticion,"Respondeme...\n");
  result = send(SocConexion, Peticion, sizeof(Peticion),0);
  if (result == -1){
    fprintf (stderr,"Error %d en send(): %s\n", errno, strerror(errno));
```

sigue ➤➤

Ejemplo TCP: cliente

➤➤ *sigue*

```
    close(SocConexion); exit(-1);  }
  printf("Peticion enviada: %s\n",Peticion);

  // Recibir una respuesta (cadena) del servidor
  result = recv(SocConexion, Respuesta, sizeof(Respuesta), 0);
  if (result == -1) {
    fprintf (stderr,"Error %d en recv(): %s\n",errno,strerror(errno));
    close(SocConexion); exit(-1); }
  // Procesar la respuesta recibida
  printf("Respuesta recibida: %s\n",Respuesta);
  // Cerrar la conexion
  close(SocConexion);
  return 1;
} // main
```

Sockets en Windows

Ficheros de cabecera

```
#include <winsock2.h>
```

Opciones de compilación

Debe enlazarse con la biblioteca **ws2_32.lib**

Observaciones

- Antes de poder usar cualquiera de las funciones de sockets, es necesario *inicializar la biblioteca de sockets*
- Cuando ya no se necesiten los sockets, se debe *cerrar la biblioteca de sockets*
- En lugar de retornar -1, las funciones Windows retornan ciertas constantes para indicar error.
- El socket no es tipo `int`, sino tipo `SOCKET`
- En Windows no se puede usar `read()` o `write()` sobre un socket. Debe usarse `recv()` y `send()` en su lugar.
- En lugar de `close()` ha de usarse `closesocket()`
- El error ocurrido se obtiene llamando a `WSAGetLastError()`, en lugar de usar `errno`.
- El resto de las funciones y conceptos son idénticos

Inicialización y cierre de la biblioteca de sockets

- Debe invocarse la función **`WSAStartup`**, solicitando la versión 2.0 de la biblioteca
- Debe comprobarse seguidamente si la versión soportada por Windows es realmente la 2.0, como se había solicitado
- Tras esto, ya se pueden usar todas las funciones de sockets normalmente.
- Para cerrar la biblioteca de sockets debe llamarse a **`WSACleanup`**

WSAStartup(), WSACleanup(), WSAGetLastError()

```
int WSAStartup(WORD version, LPWSADATA DatosWinSock);
```

version
: Es la versión de la biblioteca que solicitamos. Para crear un dato de tipo `WORD` es necesario usar la macro `MAKEWORD`

DatosWinSock
: Puntero a una estructura de tipo `WSADATA`, a través de la cual Windows devuelve información sobre qué versión de la biblioteca soporta.

Retorna
: 0 si hay éxito. Una constante de error si no.

```
int WSACleanup(void);
```

Retorna
: 0 si hay éxito. Una constante de error si no.

```
int WSAGetLastError(void);
```

Retorna
: Código del último error ocurrido.

Ejemplo: servidor

Inicialización de biblioteca

```
int EntDevuelto;
WORD Version;
struct WSAData DatWinSock;

Version = MAKEWORD(2,0);
EntDevuelto = WSAStartup(Version, &DatWinSock);
if (EntDevuelto != 0)
{ fprintf(stderr,"No hay biblioteca Winsock.\n");
  exit(-1);
}
if ((LOBYTE(DatWinSock.wVersion) != 2) ||
    (HIBYTE(DatWinSock.wVersion) != 0))
{ fprintf(stderr,"Versión incorrecta.\n");
  WSACleanup(); exit(-1);
}
```

- Inicializar **Version**
- Inicializar biblioteca
- Comprobar errores
- Comprobar versión
- Cerrar biblioteca en caso de error

Ejemplo: servidor

Creación de socket de escucha...

```
#define PUERTO 9876
 int resul;
 SOCKET SocEscucha;
 struct sockaddr_in Direc;

 // Crear socket
 SocEscucha = socket(PF_INET,SOCK_STREAM,0);
 if (SocEscucha == INVALID_SOCKET) { // Comprobación de errores
   fprintf(stderr,"Error %d en socket()\n", WSAGetLastError());
   WSACleanup(); exit(-1); }
 // Asignar dirección
 Direc.sin_family = AF_INET;
 Direc.sin_port = htons(PUERTO);
 Direc.sin_addr.s_addr = htonl(INADDR_ANY);
 resul = bind(SocEscucha, (struct sockaddr *) &Direc, sizeof(Direc));
 if (resul == SOCKET_ERROR) { // Comprobación de errores
   fprintf(stderr,"Error %d en bind()\n", WSAGetLastError());
   closesocket(SocEscucha); WSACleanup(); exit(-1); }
```

sigue

Ejemplo: servidor

...Creación de socket de escucha

➤➤ *sigue*

```
// Poner modo "escucha"
resul = listen(SocEscucha,SOMAXCONN);
if (resul == SOCKET_ERROR) { // Comprobación de errores
  fprintf(stderr,"Error %d en listen()\n", WSAGetLastError());
  closesocket(SocEscucha); WSACleanup(); exit(-1); }
```

- Observar cómo se detectan los errores.
- Observar cómo se averigua el error
- y cómo se cierra el socket y la biblioteca antes de salir.

Ejemplo: servidor

Espera por clientes y aceptación

```
SOCKET SocDatos;
struct sockaddr_in DirCliente;
int LongDirCliente;

// Inicializar la variable de longitud
LongDirCliente = sizeof(sockaddr);
printf("Esperando conexion...\n");
// accept espera hasta que llegue un cliente y retorna
// la dirección desde la que se conectó el cliente
// y un nuevo socket de datos para la comunicación
SocDatos = accept(SocEscucha, (struct sockaddr *) &DirCliente,
                  &LongDirCliente);
if (SocDatos == INVALID_SOCKET) {
   fprintf(stderr, "Error %d en accept()\n", WSAGetLastError());
   closesocket(SocEscucha); WSACleanup(); exit(-1); }

// Tenemos un nuevo cliente conectado
printf("Cliente desde: %s, puerto %d\n",
       inet_ntoa(DirCliente.sin_addr), DirCliente.sin_port);
```

Ejemplo: servidor

Recibe petición, envía respuesta

```
char Peticion[200];
char Respuesta[200];

// Leer la petición
resul = recv(SocDatos, Peticion, sizeof(Peticion), 0);
if (resul == SOCKET_ERROR) {
 fprintf(stderr, "Error %d en recv()\n", WSAGetLastError());
 closesocket(SocDatos); closesocket(SocEscucha);WSACleanup();exit(-1); }

printf("Recibidos %d caracteres: %s\n", resul, Peticion);

// Preparar una respuesta
sprintf(Respuesta,"Me has enviado %d bytes\n", resul);
// Enviarla
resul = send(SocDatos, Respuesta, sizeof(Respuesta), 0);
if (resul == SOCKET_ERROR) {
 fprintf(stderr, "Error %d en send()\n", WSAGetLastError());
 closesocket(SocDatos); closesocket(SocEscucha);WSACleanup();exit(-1); }
```

Ejemplo: cliente

- La inicialización de la biblioteca es idéntica a la del servidor
- La creación del socket es análoga, pero no requiere hacer **bind()** ni se debe hacer **listen()**

Creación del socket

```
// Inicialización de la versión 2.0 de la librería
// (no se muestra el código por ser igual al del servidor)

SOCKET SocCli;

SocCli = socket(PF_INET, SOCK_STREAM, 0);
if (SocCli == INVALID_SOCKET) {
  fprintf(stderr, "Error %d en socket()\n", WSAGetLastError());
  closesocket(SocCli); WSACleanup(); exit(-1); }
```

Ejemplo: cliente

Conexión

```
#define PUERTO 9876
struct sockaddr_in DirServidor;
int resul;

// Especificacion de la direccion del servidor
DirServidor.sin_family = AF_INET;
DirServidor.sin_port = htons(PUERTO);
DirServidor.sin_addr.s_addr = inet_addr("156.35.131.166");

// Conectar el socket
resul = connect(SocCli, (struct sockaddr *) &DirServidor,
                sizeof(DirServidor));
if (resul == SOCKET_ERROR) {
  fprintf(stderr, "Error %d en connect()\n", WSAGetLastError());
  closesocket(SocCli); WSACleanup(); exit(-1); }
```

- El cliente debe especificar la dirección del servidor
- Y conectarse a ella usando `connect()`

Ejemplo: cliente

Petición/Respuesta

```
char Peticion[200];
char Respuesta[200];

// Enviar una peticion (cadena) al servidor
sprintf(Peticion,"Respondeme...\n");
resul = send(SocCli, Peticion, sizeof(Peticion),0);
// Comprobación de errores (igual que en connect)

// Recibir una respuesta (cadena) del servidor
resul = recv(SocCli, Respuesta, sizeof(Respuesta), 0);
// Comprobación de errores (igual que en connect)
printf("Respuesta recibida: %s\n", Respuesta);
```

- Cuando `connect()` retorne, tendremos una conexión
- El envío y recepción de datos es análogo al servidor
  - Usando `send()` para enviar la petición
  - Y `recv()` para leer la respuesta

Funciones adicionales

El API de sockets es extenso y tiene funciones para:

- Convertir una IP al formato interno requerido por la estructura `sockaddr_in`
- Convertir el formato interno de `sockaddr_in` a una cadena de texto que muestre la IP en la típica "notación de punto"
- Obtener la IP de una máquina a partir de su nombre
- Obtener datos de una máquina a partir de su dirección
- Obtener el número de puerto asociado a un servicio
- Obtener el nombre del servicio asociado a un número de puerto
- Modificar opciones de funcionamiento de los sockets
- etc.

Ficheros de cabecera

Son necesarios sólo en UNIX

#include <netinet/in.h> Estructuras de datos, como **`sockaddr_in`**, etc. Funciones para conversión entre el orden de máquina y orden de red.

#include <arpa/inet.h> Funciones para convertir entre texto (notación “de puntos”) y las estructuras internas: **`inet_addr()`**, **`inet_aton()`**, **`inet_ntoa()`**.

#include <netdb.h> Funciones para obtener la IP de una máquina a partir de su nombre, y viceversa: **`gethostbyname()`**, **`gethostbyaddr()`**, y definición de las estructuras necesarias para devolver los resultados (**`hostent`**).

inet_aton()

Esta función convierte un texto como `"156.35.151.2"` en un dato de 32 bits, en orden de red, listo para almacenar en el campo apropiado de la estructura **sockaddr_in**

```
int inet_aton(char *cadena, struct in_addr *resultado);
```

cadena	Puntero a un string que contiene la dirección IP en formato textual.
resultado	Puntero a una estructura donde se almacenará el resultado.
Retorna	0 si la dirección es inválida.

Ejemplo

```
struct sockaddr_in dir_servidor;

// No se incluye comprobación de errores
inet_aton("156.35.151.2", &dir_servidor.sin_addr);
```

inet_ntoa()

Esta función convierte una estructura **in_addr** en una cadena de texto que contiene la dirección IP en notación "de puntos".

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

dirección	Estructura que contiene la dirección.
Retorna	Puntero al string con el texto resultante.

Ejemplo: Mostrar la dirección del cliente que se ha conectado

```
struct sockaddr_in dir_cliente;
int sock_dat, sock_escucha;
int tam_dir_cliente;
// ...
tam_dir_cliente=sizeof(dir_cliente);
sock_dat=accept(sock_escucha, &dir_cliente, &tam_dir_cliente);
printf("Conectado un cliente desde la IP %s\n",
         inet_ntoa(dir_cliente.sin_addr);
```

gethostbyname()

Esta función recibe el nombre de una máquina y devuelve una estructura con información sobre la misma

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

maquina — Cadena de texto con el nombre de la máquina.

Retorna — Puntero a una estructura de tipo **hostent** con la información obtenida sobre esa máquina.

Estructura hostent

```
struct hostent {
 char     *h_name;         // nombre oficial del anfitrión
 char     **h_aliases;     // lista de alias
 int      h_addrtype;      // tipo dirección anfitrión
 int      h_length;        // longitud de la dirección
 char     **h_addr_list;   // lista de direcciones
}
```

Ejemplo de gethostbyname()

Para imprimir la información obtenida

```
struct hostent *info;
int i;                        // Indice de bucle
struct sockaddr_in dir; // Para convertir direcciones

info=gethostbyname("www.uniovi.es");
printf("  Nombre oficial: %s\n", info->h_name);
printf("  Lista de alias:\n");      i=0;
while (info->h_aliases[i]!=NULL) {
  printf("     %s\n", info->h_aliases[i++]);
}
printf("  Lista de IPs:\n");     i=0;
while (info->h_addr_list[i]!=NULL) {
  // Copiar la info a una estructura apropiada
  memcpy(&dir.sin_addr.s_addr, info->h_addr_list[i],
         sizeof(dir.sin_addr.s_addr));    i++;
  printf("     %s\n", inet_ntoa(dir.sin_addr));
}
```

Otro ejemplo de **gethostbyname()**

Para averiguar la IP y conectar a un servidor

```
struct hostent *info;
struct sockaddr_in dir; // Para convertir direcciones
int soc;

info=gethostbyname("www.uniovi.es");
// Inicializar estructura dir
memcpy(&dir.sin_addr.s_addr, info->h_addr_list[0],
       sizeof(dir.sin_addr.s_addr));
dir.sin_family=AF_INET;
dir.sin_port=htons(80);
// Crear el socket para conectar
soc=socket(PF_INET, SOCK_STREAM, 0);
// Intentar la conexión
if (connect(soc, (struct sockaddr *)&dir, sizeof(dir))==-1) {
  perror("connect");
  exit(-1);
}
```

gethostbyaddr()

Esta función devuelve el mismo tipo de información que **gethostbyname()**, pero a partiendo de su dirección (tal como la retorna **inet_addr()**), en lugar de su nombre.

```
struct hostent *gethostbyaddr(char *direccion, int longitud, int tipo);
```

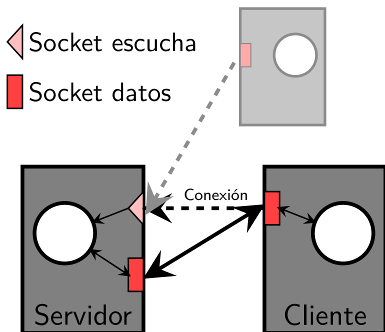
- **direccion**: Puntero a un dato de 32 bits que contiene la IP de la máquina (en orden de red). Por ejemplo, un puntero al campo **s_addr** de una estructura **in_addr**.
- **longitud**: Tamaño de la dirección (4 bytes, pero es más seguro usar **sizeof()**)
- **tipo**: Tipo de dirección: **AF_INET**
- **Retorna**: Puntero a una estructura de tipo **hostent** con la información obtenida sobre esa máquina.

No veremos ejemplos. Remitirse a **gethostbyname()**

Esquema

1. Introducción
2. Sockets
3. Programación de servidores
   - El problema de los clientes concurrentes
   - Solución creando varios procesos
   - Solución usando `select()`
4. Conclusiones

Concurrencia de servicios



El problema:

- Mientras el servidor está ocupado con un cliente, otros intentan conectar
- Si el servidor no `accepta` pronto, el cliente puede abandonar
- ¿Cómo aceptar clientes nuevos sin dejar de dar servicio a los ya aceptados?

Soluciones

- **Creación de procesos extra** para dar el servicio. Hay dos estrategias:
  - Un proceso atiende el socket de escucha. Cada vez que se conecta un cliente, se crea otro proceso.
  - Se crean de antemano un conjunto de procesos. Todos ellos escuchan en el mismo socket.
- **Evitar que el servidor quede bloqueado** en las funciones `accept()`, `read()` o `recv()`. De nuevo hay dos estrategias:
  - Modificar las opciones de los sockets para que sean *no-bloqueantes* (no veremos este método)
  - Hacer uso de la función `select()`

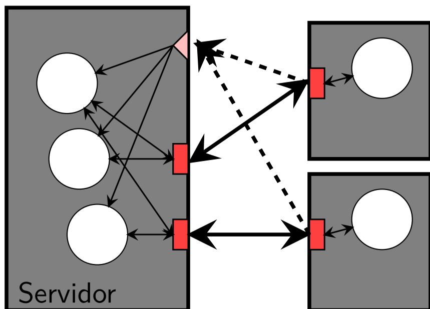
Creación de procesos cuando se conectan los clientes

1. El proceso inicial crea un socket de escucha.
2. El proceso inicial hace **`accept()`**
3. Cuando **`accept()`** retorne, el padre obtiene un nuevo socket de datos.
4. Entonces crea un nuevo proceso (hijo)
   - Los procesos padre e hijo reciben los mismos descriptores de socket.
   - El proceso padre, cierra el socket de datos, y vuelve al paso 2.
   - El proceso hijo, en cambio, cierra el socket de escucha, y proporciona el servicio solicitado a través del socket de datos.

Implementación

Ejemplo mínimo

```
// Includes omitidos
main()
{
  int SockEscucha, SockDatos; // etc. .
  SockEscucha=socket(PF_INET, SOCK_STREAM, 0);
  // Asignarle direccion con bind()
  // Convertirlo en escucha con listen()
  while(1) { // Bucle infinito
    SockDatos=accept(SockEscucha, &DirCliente, &lDirCliente);
    if (fork()==0) { // Es el hijo
      close(SockEscucha);  // Cierra su conexión con el socket de escucha
      // Comunicarse a traves de SockDatos
      // read( ... )    write( ... )     close(SockDatos)
      exit(0);         // El hijo muere, una vez termina el servicio
    } else { // Es el padre
      close(SockDatos);   // Cierra su conexión con el socket de datos
                          // Tras lo cual volverá al while
    }
  } // fin del while
}
```



1. Servidor espera
2. Cliente conecta
3. Servidor acepta
4. Servidor hace `fork()`
5. Padre cierra datos. Queda a la escucha. Hijo cierra escucha, atiende al cliente.
6. Otro cliente conecta
7. Padre acepta, crea nuevo hijo
8. Como el paso 5
9. Cliente termina. Hijo muere.
10. Cliente termina. Hijo muere.

Problema: Procesos *zombies*

- Cuando un hijo muere, envía una señal **SIGCHLD** al proceso padre
- El proceso padre debe estar programado para recoger esta señal y obrar en consecuencia
- Si no lo hace, el proceso hijo no muere del todo, quedando en estado *zombie*

La forma más sencilla de evitar este problema es programar el padre para que ignore la señal **SIGCHLD**. Esto se logra ejecutando en el proceso padre la siguiente línea:

```
signal(SIGCHLD, SIG_IGN);
```

Requiere `#include <signal.h>`

Creación previa de procesos

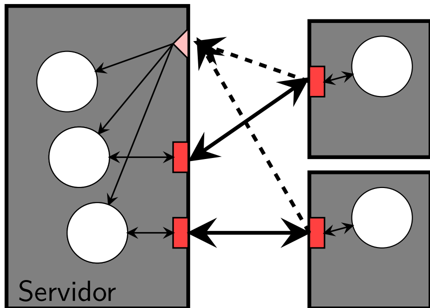
1. El proceso inicial crea el socket de escucha
2. A continuación crea un conjunto de procesos hijos
3. Todos los procesos "heredan" el mismo descriptor de socket y todos ejecutarán seguidamente el mismo código:
    - Llamar a `accept()` sobre el socket de escucha
    - Cuando un cliente se conecta, uno de los procesos despertará
    - El socket de datos creado por `accept()` se usará para comunicarse con el cliente.
    - Una vez terminado el servicio, el proceso cerrará el socket de datos, y volverá a bloquearse en `accept()`

Implementación

Ejemplo mínimo

```
// Includes omitidos
main()
{
  int SockEscucha, SockDatos; // etc...
  // Crear socket
  SockEscucha=socket(PF_INET, SOCK_STREAM, 0);
  // Darle una dirección con bind() y hacerlo de escucha con listen()
  // Creacion previa de procesos
  for (i=0; i<5; i++) if (fork()==0) break; // Creamos 5 hijos
  // Todos los hijos y el padre ejecutan lo que sigue:
  while (1) { // Bucle infinito
    // Obtener cerrojo si es necesario (no se muestra cómo)
    SockDatos=accept(SockEscucha, &DirCliente, &lDirCliente);
    // Liberar cerrojo si es necesario (no se muestra cómo)

    // Proporcionar el servicio a través de SockDatos
    // read( ... )     write( ... )     close(SockDatos)
    // Tras lo que se vuelve al while
  }
}
```



1. Padre crea socket escucha
2. Padre crea hijos
3. Cliente conecta
4. Un hijo cualquiera acepta y da servicio
5. Otro cliente conecta
6. Otro hijo acepta y da servicio
7. Cliente termina. Hijo vuelve a accept.
8. Cliente termina. Hijo vuelve a accept.

Concurrencia aparente

Existe otra forma de proporcionar concurrencia (aparente) en los servicios:

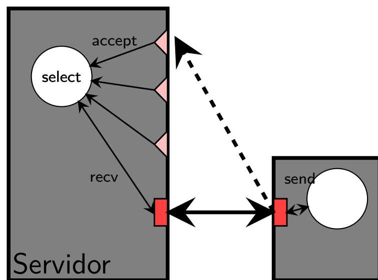
- Un solo proceso atiende los socket de escucha y también los de datos
- El problema es que este proceso no debería quedar bloqueado, pues esto afecta a los restantes clientes.
- Evitar el bloqueo implica:
  - No llamar nunca a **`accept()`**, a menos que se sepa de antemano que hay un cliente esperando
  - No llamar nunca a **`read()`** (o **`recv()`**), a menos que se sepa de antemano que hay datos para leer
  - No llamar nunca a **`write()`** (o **`send()`**), a menos que se sepa de antemano que el otro extremo va a leer los datos

Usando `select()`

La función `select()`

- Recibe un conjunto de *descriptores* a observar
  - El descriptor representa un socket
  - También puede representar un fichero
  - O incluso la entrada estándar
- Mientras no haya actividad en ninguno de los *descriptores*, `select()` no retorna.
- Cuando se detecta actividad en uno de los *descriptores*, `select()` retorna información acerca del descriptor activo.
- También se puede especificar un *tiempo máximo de espera*

Uso de `select()`



1. El proceso crea sockets de escucha.
2. Llama a `select()`
3. Cuando un cliente conecta, `select()` retorna
4. Se puede hacer entonces `accept()`.
5. Se vuelve al `select()`
6. Cuando el cliente envíe algo, `select()` retorna
7. Se puede hacer entonces `read()`
8. etc.

Prototipo de `select()`

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
int select(int max, fd_set *fdlectura, fd_set *fdescritura,
           fd_set *fdexcepc, struct timeval *timeout);
```

max
: Máximo más 1 de los valores de los descriptores a observar.

fdlectura, fdescritura, fdexcepc
: Estructuras en las que se especifican los descriptores en los que se desea detectar “se puede leer”, “se puede escribir” o “hay excepción”, respectivamente. Al retorno estarán modificadas.

timeout
: Estructura en que se especifica el tiempo máximo que **select()** debe esperar por actividad en los descriptores.

Retorna
: Cuántos descriptores han tenido actividad, 0 en caso de *timeout*, -1 en caso de error.

Manejo de las estructuras **fd_set**

Ejemplo de declaración

```
fd_set fd;
```

Una estructura **fd_set** almacena un *campo de bits*, en el que cada bit representa un *descriptor*. Si el bit está a 1, el correspondiente descriptor será observado, si está a 0, no lo será.

Los bits individuales se manejan con las siguientes funciones

FD_ZERO(&fd)	Pone todos los bits de **fd** a cero
FD_SET(n,&fd)	Pone a 1 el bit *n* de **fd**
FD_CLR(n,&fd)	Pone a 0 el bit *n* de **fd**
FD_ISSET(n,&fd)	Devuelve el valor del bit *n* de **fd**, para usar en condiciones.

Ejemplo de uso de `select()`

```
// Este ejemplo muestra cómo llamar a select() para que detecte
// cuándo es seguro leer de cualquiera de los sockets s1 o s2.
// La creación e inicialización de s1 y s2 no se muestra. Sólo las
// estructuras fd_set y variables necesarias para llamar a select.
fd_set fdlectura;   // Solo vigilaremos para lectura
int max;

FD_ZERO(&fdlectura);          // Inicializar todos los bits a cero
FD_SET(s1, &fdlectura);       // Poner a 1 los bits de los sockets
FD_SET(s2, &fdlectura);       // que queremos observar
max=(s1>s2)?s1:s2;            // Calcular el maximo de s1 y s2
max++;                        // y sumarle 1
ret=select(max, &fdlectura, NULL, NULL, NULL);
if (ret==-1) { perror("En el select"); exit(-1); }
if (FD_ISSET(s1, &fdlectura)) printf("Actividad en socket s1\n");
if (FD_ISSET(s2, &fdlectura)) printf("Actividad en socket s2\n");
```

Especificando un *timeout*

Tipo de datos **timeval**:

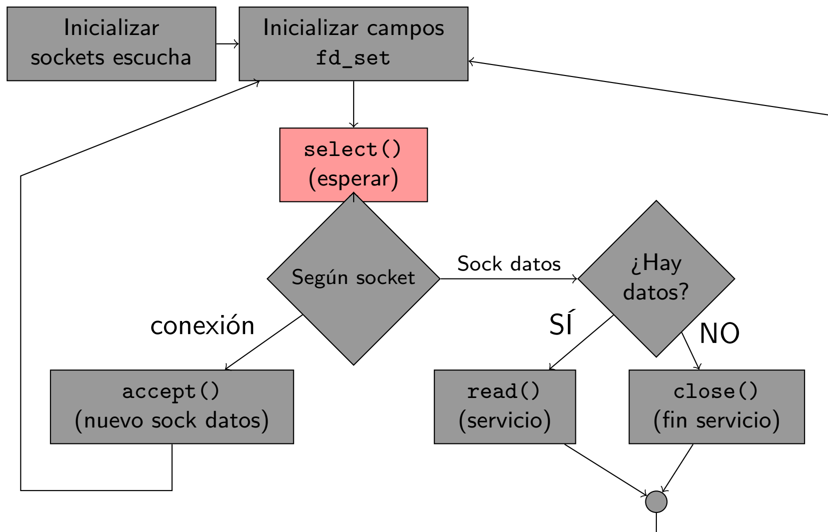
```
struct timeval {
  long tv_sec;  /* segundos */
  long tv_usec; /* microsegundos */
};
```

- Si se inicializan ambos campos con cero, **select()** retornará inmediatamente
- Si se quiere que **select()** espere indefinidamente, se le debe pasar un puntero **NULL** como último parámetro

Ejemplo: especificar un tiempo de 3,5 seg

```
struct timeval tiempo;
tiempo.tv_sec=3;  tiempo.tv_usec=500000;
ret=select(max, &fdlectura, NULL, NULL, &tiempo);
```

Diagrama de flujo simplificado del servidor



Esquema

1 Introducción

2 Sockets

3 Programación de servidores

4 Conclusiones
- Ventajas
- Inconvenientes

Ventajas de la API de sockets

- Abstracción de la capa de red
- Soporte para protocolos TCP y UDP
- Acceso de bajo nivel a los servicios de red del operativo
- Programación del modelo cliente/servidor relativamente sencilla

Inconvenientes

- Falta de transparencia.
- Programación tediosa.
- Problemas al intercomunicar diferentes arquitecturas.
- Problema del envío o recepción incompleto.
- Gestión de errores compleja
  - Basada en la variable **`errno`** o el uso de **`WSAGetLastError()`**.
  - No hay diferente tipo para socket de escucha o de datos, ni para socket TCP o UDP.
  - En UNIX el socket es un **`int`**.
- Portabilidad.
- Seguridad.

Conclusión

Sería interesante una capa de software *por encima* de los sockets, que mantenga sus ventajas pero oculte sus inconvenientes.

Esta capa se denomina *middleware* y será objeto de estudio en las próximas lecciones.