

**Redes de Computadores**  
*Nivel de Aplicación:*  
*Programación con sockets 2*

Área de Ingeniería Telemática  
Dpto. Automática y Computación  
<http://www.tlm.unavarra.es/>

# En la clase de hoy

- ▶ **Sockets TCP**
  - > cliente
  - > servidor
  
- ▶ **Sockets TCP concurrentes**

# Sockets TCP

## ▶ Modo cliente

- > Toma la iniciativa de establecer la comunicación
- > Conoce la dirección y el puerto del servidor
- > Abre un socket y lo usa para conectarse a un servidor
  - + A partir de ahí intercambia datos con el servidor
  - + Normalmente el cliente pide cosas que le envía el servidor

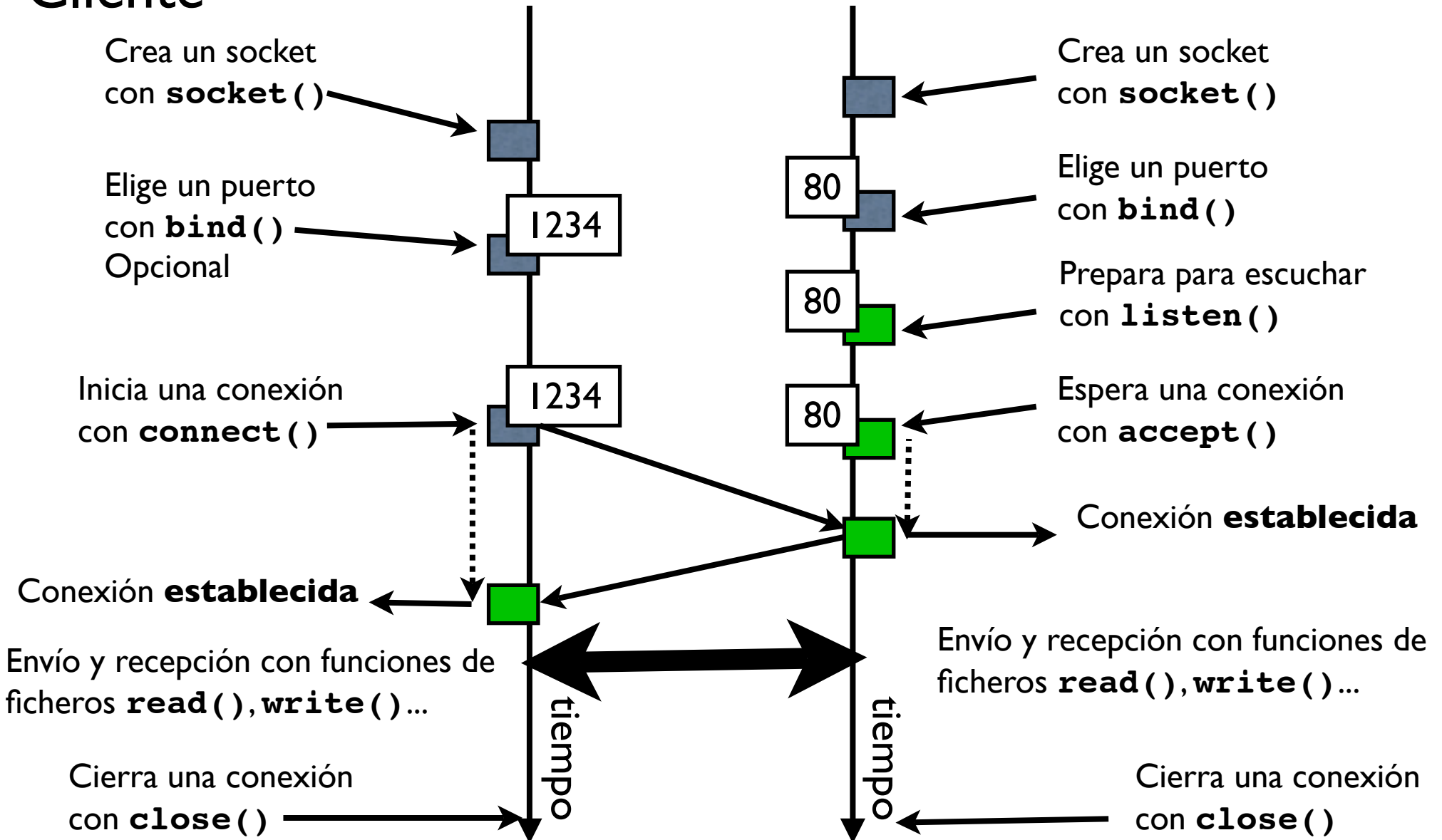
## ▶ Modo servidor

- > Espera recibir comunicaciones de clientes
- > Elige un puerto que deberá ser “conocido” por los clientes
- > Abre un socket y espera recibir conexiones
- > Obtiene un nuevo socket para hablar con cada cliente
  - + A partir de ahí intercambia datos con el cliente
  - + Normalmente el servidor recibe peticiones y las contesta

# Sockets TCP: operaciones

## Cliente

## Servidor



# Sockets TCP: eligiendo puerto

- ▶ La función `bind()` permite configurar en el socket
  - > Puerto
  - > Dirección IP (por ejemplo para elegir sólo contestar por una conexión de red si estamos conectados a varias)

```
int bind(int s, const struct sockaddr *name, socklen_t namelen);
```

**error?**  
-1 : error  
0 : ok

**socket**  
a configurar

**direccion**  
{ IP , puerto } para el socket

Si sólo queremos poner el puerto  
debemos asignar como  
IP: INADDR\_ANY

**direccion\_len**  
tamaño de la  
estructura de  
direccion

# Sockets TCP: clientes

## ▶ Estableciendo y cerrando conexiones

```
int connect(int s, const struct sockaddr *name, socklen_t namelen);
```

### **error?**

-1 : error  
0 : ok

**socket**  
establece  
conexión en  
este socket

### **destino**

destino de la conexión  
Estructura con una dirección { IP , puerto }  
convertida a sockaddr  
(struct sockaddr \*)&servidorDir

### **destino\_len**

tamaño de la  
estructura con  
el destino

```
int close(int s);
```

### **error?**

-1 : error  
0 : ok

**socket**  
cierra el  
descriptor  
de fichero

```
int shutdown(int s, int how);
```

### **error?**

-1 : error  
0 : ok

### **socket**

**sentido** a cerrar  
SHUT\_RD lectura  
SHUT\_WR escritura  
SHUT\_RDWR los dos

# Sockets TCP: leyendo y escribiendo

## ▶ Usando funciones básicas de nivel de ficheros

> `read( socket, buffer, n )`

Lee n bytes a un bufer desde el socket

> `write( socket, buffer, n )`

Envía n bytes de este buffer por el socket

## ▶ Usando funciones de lectura de streams

> Construir un stream (`FILE *`) sobre el socket con

`fdopen(socket, modo)`    `modo={ "r", "w", "r+", "w+" ... }`

> Leer y escribir con funciones de streams `fgets()`, `fprintf()` ...

> Ejemplo:

```
socket_stream = fdopen( socket , "r+" );  
fgets( linea_leida , 200 , socket_stream );  
fprintf ( socket_stream , "GET / HTTP/1.0\r\n\r\n");
```

# Sockets TCP: detectando el cierre

- ▶ Leyendo el socket con `read()`
  - > `read()` devuelve 0 para indicar fin del fichero o de CONEXIÓN
  - > `read()` devuelve -1 para indicar error
- ▶ Leyendo el socket con `fgets()`
  - > `fgets()` devuelve `NULL` para indicar fin de fichero o CONEXIÓN
- ▶ El fin de conexión se trata como EOF (End of file) y no como un error en el fichero



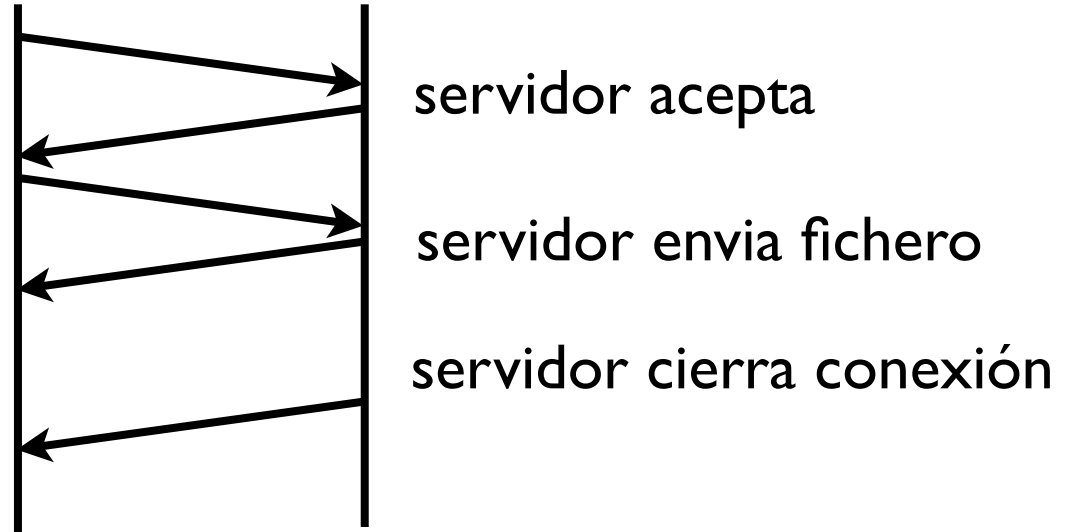
# Ejemplo: TCP cliente de web

## ▶ Protocolo de aplicación HTTP

Conexión al servidor web

Envío petición  
GET / HTTP/1.0

Recibir todo lo que  
llegue hasta que se  
cierre la conexión



## ▶ Un cliente de Web sencillo

- > Conexión al servidor indicado en el puerto 80
- > Envío de una petición de Web
- > Recepción de la respuesta
- > Imprimir en pantalla el código de la página

# Ejemplo: cliente web

- ▶ Preparación de un socket TCP
- ▶ Construimos la dirección del servidor en un struct sockaddr\_in

```
int main(int argc, char *argv[]) {  
    int sock;  
    int err;  
    struct sockaddr_in servidor;  
    char buf[20000];  
    char *aux;  
    int leidos;
```

sockaddr\_in

```
/* Abrimos el socket */  
sock=socket(PF_INET,SOCK_STREAM,0);
```

Socket

```
/* Rellenamos la estructura de la direccion */  
servidor.sin_family=AF_INET;  
servidor.sin_port=htons(80);  
servidor.sin_addr.s_addr=htonl(0x82CEA0D7);
```

rellenar  
sockaddr\_in

```
/* Conexion al servidor TCP */  
err = connect(sock,(struct sockaddr *)&servidor,sizeof(servidor));  
if ( err == -1 ) {  
    printf("Error!! no consigo conectar!!!\n");  
    exit(-1);  
}
```

# Ejemplo: cliente web

- ▶ Conexión con el servidor
- ▶ Preparamos la petición en un buffer
- ▶ Enviamos el buffer con write( )

```
servidor.sin_port=htons(80);  
servidor.sin_addr.s_addr=htonl(0x82CEA0D7);
```

```
/* Conexion al servidor TCP */
```

```
err = connect(sock,(struct sockaddr *)&servidor,sizeof(servidor));
```

```
if ( err == -1 ) {
```

```
    printf("Error!! no consigo conectar!!!\n");
```

```
    exit(-1);
```

```
}
```

```
/* La conexion esta establecida */
```

```
/* Escribamos la peticion de Web */
```

```
sprintf(buf,"GET / HTTP/1.0\n\r\n\r");
```

```
/* Y la enviamos */
```

```
write(sock,buf,strlen(buf));
```

```
/* Esperamos la respuesta */
```

```
aux=buf;
```

```
while ((leidos=read(sock,aux,20000))!=0) {
```

```
    if (leidos>0) {aux=aux+leidos;};
```

```
}
```

conexión

Preparar petición  
y envío

# Ejemplo: cliente web

- ▶ Leemos con `read( )` a un buffer mientras no llegue un fin de fichero
- ▶ Imprimimos el buffer con `todo`

```
/* La conexion esta establecida */
/* Escribamos la peticion de Web */
sprintf(buf,"GET / HTTP/1.0\n\r\n\r");
/* Y la enviamos */
write(sock,buf,strlen(buf));

/* Esperamos la respuesta */
aux=buf;
while ((leidos=read(sock,aux,20000))!=0) {
    if (leidos>0) {aux=aux+leidos;};
}

/* Ya tenemos toda la pagina */
/* Completamos la cadena */
*aux=0;
/* Y la imprimimos */
printf("%s\n",buf);
}
```

Leemos mientras no  
haya fin de fichero

Imprimir lo que ha  
llegado

# Ejemplo: otro cliente web (Streams)

- ▶ Apertura igual que antes
- ▶ Con la conexión establecida construimos el stream con `fdopen()`

```
/* Abrimos el socket */
sock=socket(PF_INET,SOCK_STREAM,0);

/* Rellenamos la estructura de la direccion */
servidor.sin_family=AF_INET;
servidor.sin_port=htons(80);
servidor.sin_addr.s_addr=htonl(0x82CEA0D7);

/* Conexion al servidor TCP */
err = connect(sock,(struct sockaddr *)&servidor,sizeof(servidor));
if ( err == -1 ) {
    printf("Error!! no consigo conectar!!!\n");
    exit(-1);
}

/* La conexion esta establecida */
/* Abrimos una Stream sobre el socket */
f=fdopen(sock,"r+");
/* Escribamos la peticion de Web por la Stream */
```

## Construir Stream

# Ejemplo: otro cliente web (Streams)

- ▶ Recibimos y enviamos con funciones de Streams
- ▶ Mucho más cómodo

```
/* La conexión está establecida */  
/* Abrimos una Stream sobre el socket */  
f=fdopen(sock,"r+");  
/* Escribamos la petición de Web por la Stream */
```

```
fprintf(f,"GET / HTTP/1.0\r\n\r\n");
```

Envío con fprintf()

```
/* Vamos imprimiendo las líneas que llegan */  
while (fgets(buf,20000,f)!=NULL) {  
    printf("%s",buf);  
}  
printf("Conexión cerrada por el servidor\n");  
fclose(f);
```

Recibimos línea a línea con fgets( )

```
}
```

Cierre del Stream

# Sockets TCP: servidores

- ▶ Elegir el puerto y quizás la dirección con `bind()`
- ▶ Poner el socket en modo servidor con `listen()`

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

**error?**  
-1 : error  
0 : ok

**socket**  
a poner en  
modo servidor

**conexiones pendientes**  
numero de conexiones  
pendientes de aceptación a  
almacenar

- ▶ Aceptar conexiones con `accept()`

```
int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
```

**socket**  
Acepta la siguiente  
conexión que llegue a  
este socket

**origen de la conexión**  
de qué dirección { IP , puerto} proviene la conexión?  
addr es un buffer de la longitud indicada por addrlen  
a la salida addrlen indica la longitud utilizada del buffer

# Sockets TCP: servidores (accept)

- ▶ `accept()` devuelve un nuevo socket que está conectado al del cliente
- ▶ el socket original se puede seguir usando para aceptar nuevas conexiones

```
socket_cliente = accept( socket , &direccion , &direccionlen );
```

```
cliente_stream = fdopen( socket_cliente , "r+" );  
fprintf( cliente_stream , "Bienvenido al servidor...\n");
```

```
socket_cliente2 = accept( socket , &direccion , &direccionlen );
```

- ▶ Nótese que no puede saber de quién viene una conexión sin aceptarla...



# Ejemplo: servidor web

- ▶ Servidor web muy simple (una página un cliente)
- ▶ Abrir un socket TCP
- ▶ Rellenar una estructura de sockaddr\_in para el puerto

```
/* include <sys/types.h>
```

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

```
int main(int argc,char *argv[]) {  
    int sock,c_sock;  
    int err;  
    struct sockaddr_in servidor,cliente;  
    char buf[20000];  
    FILE *f;
```

```
/* Abrimos el socket */  
sock=socket(PF_INET,SOCK_STREAM,0);
```

```
/* Rellenamos la estructura para pedir puerto */  
servidor.sin_family=AF_INET;  
servidor.sin_port=htons(80);  
servidor.sin_addr.s_addr=INADDR_ANY;  
err = bind(sock,(struct sockaddr *)&servidor,sizeof(servidor));
```

Abrir socket

Rellenar  
sockaddr\_in

# Ejemplo: servidor web

- ▶ `bind()` para elegir el puerto

Un error significa que el puerto no esta disponible

- ▶ `listen()` se usa con 5 por tradición

```
/* Abrimos el socket */
sock=socket(PF_INET,SOCK_STREAM,0);

/* Rellenamos la estructura para pedir puerto */
servidor.sin_family=AF_INET;
servidor.sin_port=htons(80);
servidor.sin_addr.s_addr=INADDR_ANY;
err = bind(sock,(struct sockaddr *)&servidor,sizeof(servidor));
if (err ==-1) {
    printf("Error!! no puedo coger el puerto!!!\n");
    exit(-1);
}
```

**bind()**

```
listen(sock,5);
```

**listen()**

```
while (1) {
    int dirlen;

    dirlen=sizeof(cliente);
    c_sock=accept(sock,(struct sockaddr *)&cliente,&dirlen);
```

# Ejemplo: servidor web

- ▶ El servidor repite indefinidamente
  - > aceptar la siguiente conexión
  - > enviar una página como respuesta (debería mirar cual le piden)

```

while (1) {
    int dirlen;

    dirlen=sizeof(cliente);
    c_sock=accept(sock,(struct sockaddr *)&cliente,&dirlen);

    printf("Sirviendo una pagina a la direccion %x\n",
           ntohl(cliente.sin_addr.s_addr));

    /* Ponemos la Stream sobre el socket */
    f=fdopen(c_sock,"w+");
    /* Deberiamos leer la peticion de Web */
    /* Para ver que pagina nos piden */
    /* con un fgets o algo asi */
    sleep(1);
    fprintf(f, "<HTML><HEAD><TITLE>funciona</TITLE></HEAD>\n");
    fprintf(f, "<BODY>\n");
    fprintf(f, "<H1 ALIGN=CENTER>\nYa tengo mi propioservidor de web\n</H1>
\n");
    fprintf(f, "</BODY></HTML>\n");
    fclose(f);

```

acepta la  
siguiente  
conexión

manda  
pagina  
web

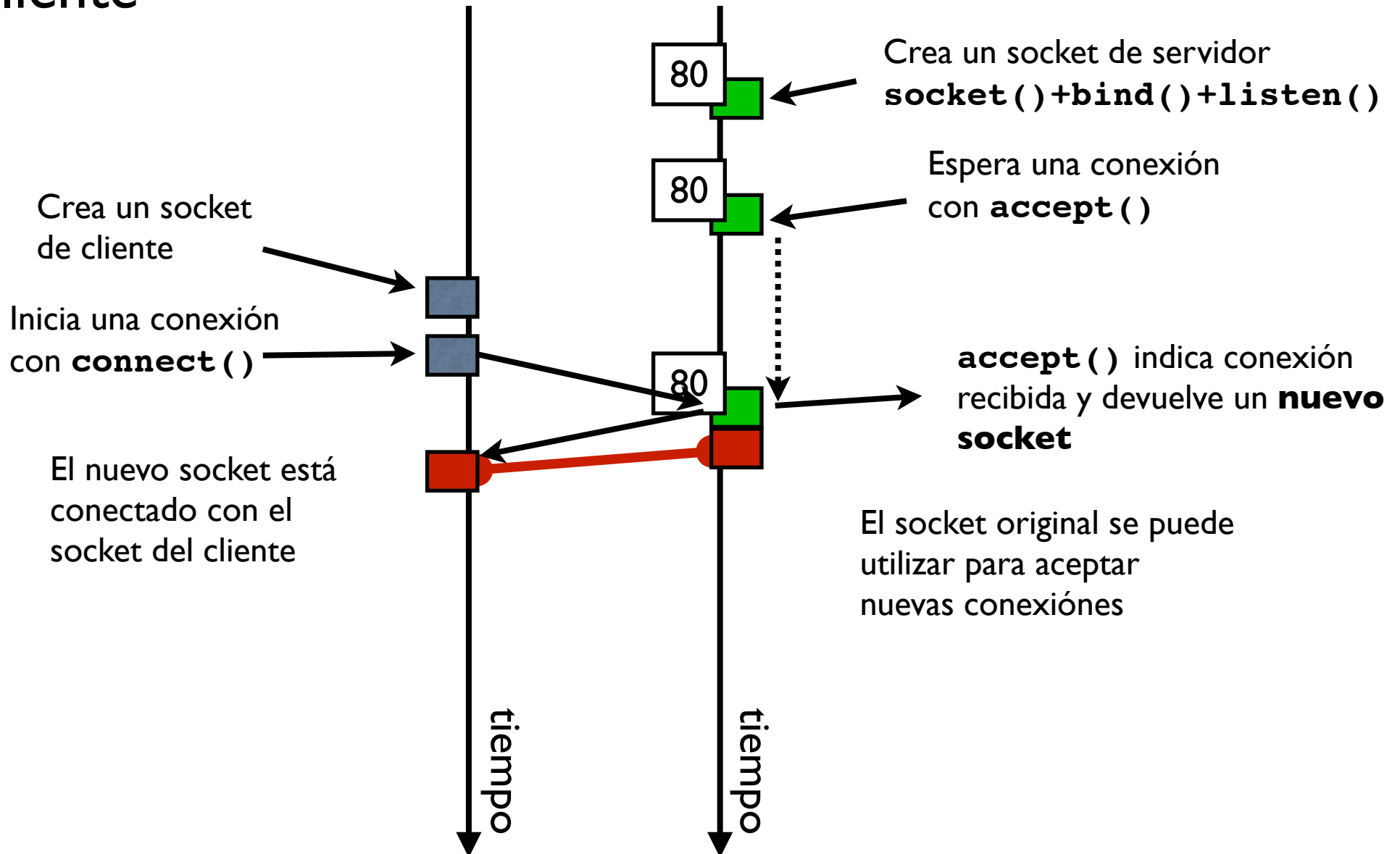
# Resumen

- ▶ Es fácil hacer clientes y servidores TCP
- ▶ El servidor que hemos construido tiene algunas limitaciones
  - > Clientes por turnos, uno no empieza hasta que acabe el anterior
  - > Esto es un problema para casi todas las aplicaciones
- ▶ Como hacer servidores TCP que atiendan a más de un cliente a la vez

# Sockets TCP: operaciones

Cliente

Servidor



# Sockets TCP: programando servidores

- ▶ En el código del servidor será:

```
socket_cliente = accept( socket , &direccion , &direccionlen );
```

- ▶ Así que ahora tenemos dos sockets uno para hablar con el cliente `socket_cliente` y otro para esperar nuevas conexiones `socket`

- ▶ En el ejemplo de la clase anterior:

Atender al cliente y después ir a esperar la siguiente petición

- ▶ Es esto razonable?
  - > Podría funcionar así un servidor de Web?
  - > Y un servidor de Telnet?

# Sockets TCP: programando servidores

- ▶ Necesitamos servidores que atiendan a varios clientes simultáneamente
- ▶ Soluciones:
  - > **Iterativos**  
Atienden a los clientes por turnos
  - > **Concurrencia** ( fork o threads )  
Atienden a varios clientes simultáneamente
  - > **E/S síncrona** ( select o poll )  
Leer solo los sockets que tienen algo que entregar
- ▶ Veamos como programar cada tipo...

# Servidor Iterativo

## ▶ Atención por turno.

```
int socket, socket_cliente;
struct sockaddr_in direccion;
int direccionlen;

while (1) {
    direccionlen = sizeof(direccion);
    socket_cliente = accept( socket ,
                            (struct sockaddr *)&direccion ,
                            &direccionlen );

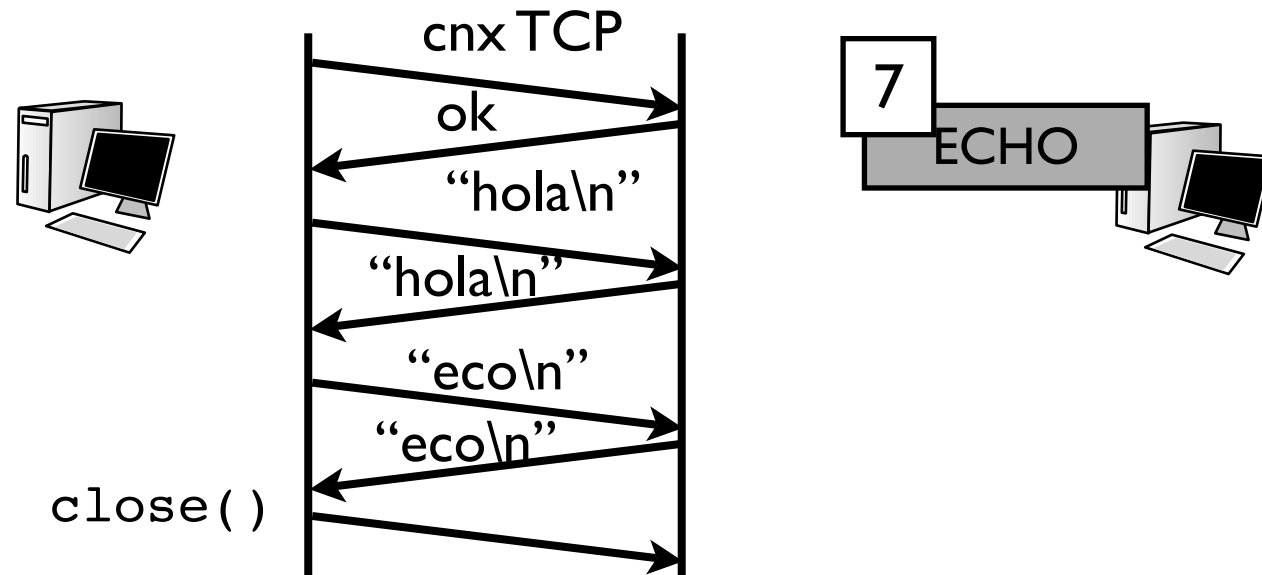
    /* aquí podríamos decidir si aceptamos al cliente según de
    donde venga comprobando su dirección de origen que
    devuelve accept */

    atender_al_cliente( socket_cliente );
    close( socket_cliente );
}
```



# Ejemplos: servidor de ECHO

- ▶ ECHO servicio clásico de UNIX para probar el funcionamiento de la red. Puerto 7 (TCP y UDP)



- ▶ El servicio de ECHO TCP acepta una conexión y devuelve todo lo que se le envíe por ella. Hasta que el cliente cierra la conexión
- ▶ El manejo de conexiones es parecido a Telnet

# Ejemplo: servidor iterativo de ECHO

- ▶ El puerto a escuchar nos lo dicen en linea de comandos
- ▶ socket de tipo TCP
- ▶ bind al puerto indicado y listen

```
if (argc<=1) {  
    puerto=1234;  
} else {  
    sscanf(argv[1], "%d", &puerto);  
}
```

Leyendo el puerto  
de los argumentos

```
/* Abrimos el socket */  
sock=socket(PF_INET, SOCK_STREAM, 0);
```

socket

```
/* Rellenamos la estructura para pedir puerto */  
servidor.sin_family=AF_INET;  
servidor.sin_port=htons(puerto);  
servidor.sin_addr.s_addr=INADDR_ANY;  
if (bind(sock, (struct sockaddr *)&servidor, sizeof(servidor))==-1) {  
    printf("Error!! no puedo coger el puerto!!!\n");  
    exit(-1);  
}
```

bind al puerto

```
listen(sock, 5);
```

```
while (1) {
```

# Ejemplo: servidor iterativo de ECHO

- ▶ repetir para siempre: accept + atender
- ▶ atender = leer y enviar lo que recibo
- ▶ podemos saber quien es el cliente

```
listen(sock,5);
```

```
while (1) {  
    int dirlen=sizeof(cliente);
```

```
    c_sock=accept(sock,(struct sockaddr *)&cliente,&dirlen);
```

accept()

```
        f = fdopen(c_sock,"w+");  
        printf("Cliente en %x\n",  
              ntohs(cliente.sin_addr.s_addr));
```

este es el cliente

```
        while ( fgets(buf,20000,f) ) {  
            printf("retransmitiendo: %s",buf);  
            fprintf(f,"%s",buf);
```

leer y enviar

```
        }  
        fclose(f);  
    }
```

```
}
```

# Servidores concurrentes

- ▶ Un proceso (o un hilo) para cada petición

```
int socket, socket_cliente;
struct sockaddr_in direccion;
int direccionlen;
int pid;

while (1) {
    direccionlen = sizeof(direccion);
    socket_cliente = accept( socket ,
                            (struct sockaddr *)&direccion ,
                            &direccionlen );

    pid = fork();
    if ( pid == 0 ) {
        atender_al_cliente( socket_cliente );
        close( socket_cliente );
        exit(0);
    }
    close( socket_cliente );
}
```

# Servidores síncronos con select

- ▶ También podemos programar servidores concurrentes utilizando un solo proceso
- ▶ Función `select()` permite saber que descriptor de fichero (socket) tiene algo para leer
  - > Para ello le indicamos un conjunto de descriptores en los que estamos interesados
- ▶ Manejo de conjuntos de descriptores
  - > Tipos y macros definidos en `#include <unistd.h>`
  - > Tipo `fdset` representa un conjunto de descriptores de ficheros
  - > Macros
    - `FD_ZERO(&fdset);` borra un conjunto de descriptores
    - `FD_SET(fd, &fdset);` añade `fd` al conjunto
    - `FD_CLR(fd, &fdset);` quita `fd` del conjunto
    - `FD_ISSET(fd, &fdset);` comprueba si `fd` esta en el conjunto

# Servidores síncronos con select

- ▶ Funcion `select()`

Que descriptores tienen algo que decir...

de estos tres conjuntos de descriptores de ficheros...

**rfds** interés en  
si tienen algo para leer

**wfds** interés en  
si están listos para escribir

**efds** interés en  
si tienen errores que informar

**n** número máximo  
de descriptores en  
los conjuntos

```
int select(int n,  
           fd_set *rfds, fd_set *wfds, fd_set *efds,  
           struct timeval *timeout);
```

**n**  
número de descriptores  
con novedades  
0: no hay novedades  
-1: error

**timeout**  
tiempo a esperar por novedades  
si pasa ese tiempo y no hay  
novedades devuelve 0

# Servidores síncronos con select

- ▶ Con `select()` ya no es un problema hacer un servidor que lea de varios sockets
  - > Mantenemos una lista de sockets que tenemos abiertos
  - > preguntamos con `select()` cuales exigen contestación.
  - > tener una conexión pendiente de aceptar es una novedad de lectura
- ▶ **El servidor es así monoproceso**
  - > Mas complicado. Tiene que llevar el estado de todos los clientes a la vez
  - > Pero es más fácil la comunicación si los procesos que atienden a varios clientes deben comunicarse

# Ejemplo: servidor de ECHO con select()

- ▶ includes nuevos para usar el select
- ▶ Necesitamos variables para varios sockets

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/time.h>
```

más includes  
para el select()

```
#define N_DESCRIPTORES 3
```

```
/* El estado del servidor */
int descriptor[N_DESCRIPTORES];
int numclientes;
```

lista de sockets

```
void aceptar_nuevo_cliente(int sock);
```

```
int main (int argc, char * argv[]) {
    int sock;
    int puerto;
    struct sockaddr_in servidor;
    char buf[20000];
```



# Ejemplo: servidor de ECHO con select()

- ▶ Marcamos los sockets no usados como -1
- ▶ un socket para escuchar conexiones  
lo preparamos en el puerto del servidor

```
int main (int argc, char * argv[]) {  
    int sock;  
    int puerto;  
    struct sockaddr_in servidor;  
    char buf[20000];  
    int i;  
  
    /* Inicializar el estado del servidor */  
    numclientes=0;  
    for (i=0;i<N_DESCRIPTORES;i++) descriptores[i]=-1;  
  
    if (argc<=1) puerto=1234;  
    else sscanf(argv[1], "%d",&puerto);  
    servidor.sin_family=AF_INET;  
    servidor.sin_port=puerto;  
    servidor.sin_addr.s_addr=INADDR_ANY;  
    sock=socket(PF_INET,SOCK_STREAM,0);  
    if (bind(sock,(struct sockaddr *)&servidor,sizeof(servidor))===-1) {  
        printf("Error: no puedo coger el puerto\n");  
        exit(-1);  
    }  
}
```

socket para  
escuchar

todos los  
sockets sin usar

binding

# Ejemplo: servidor de ECHO con select()

- ▶ En cada iteración preparamos el conjunto de sockets que nos interesa escuchar

! para aceptar + los que tengamos abiertos (!=-1)

```
listen(sock,5);

while(1) {
    fd_set paraleer;
    int fd_maximo=0;

    /* Preparar el conjunto con los sockets que nos interesan */
    FD_ZERO(&paraleer);
    FD_SET(sock,&paraleer);
    fd_maximo=sock; /* Para ir calculando el maximo */
    for (i=0;i<N_DESCRIPTORES;i++)
        if (descriptores[i]!=-1) {
            FD_SET(descriptores[i],&paraleer); /* Añadimos cada
descriptor abierto */
            fd_maximo= fd_maximo>descriptores[i]?fd_maximo:descriptores[i];
            /* Y actualizamos el maximo
        */
    }
    fd_maximo++; /* sumamos uno porque select solo mira de 0 a fd_maximo-1
*/

    if (select(fd_maximo,&paraleer,NULL,NULL,NULL)>0) {
```

variable conjunto

vaciar conjunto y  
añadir sock

Añadir descriptores  
abiertos

# Ejemplo: servidor de ECHO con select()

- ▶ Preguntamos quien del conjunto tiene algo para leer
- ▶ Si sock tiene algo que leer aceptamos una nueva conexión
- ▶ Si es un socket de cliente...

```
*/
```

```
if (select(fd_maximo,&paraleer,NULL,NULL,NULL)>0) {  
    /* Hay algo que leer en alguno */  
    if (FD_ISSET(sock,&paraleer)) {  
        /* Conexion para aceptar */  
        aceptar_nuevo_cliente(sock);  
    }  
    for (i=0;i<N_DESCRIPTORES;i++) {  
        if (descriptores[i]!=-1) {  
            if (FD_ISSET(descriptores[i],&paraleer)) {  
                int leidos;  
                /* hay algo para leer en el i */  
                leidos=read(descriptores[i],buf,20000);  
                if (leidos>0) {  
                    write(descriptores[i],buf,leidos);  
                } else {  
                    close(descriptores[i]);  
                    descriptores[i]=-1;  
                    numclientes--;  
                    printf("Ahora hay %d clientes\n",numclientes);  
                }  
            }  
        }  
    }  
}
```

quien?

Es el socket de  
servidor

es uno de los otros?

# Ejemplo: servidor de ECHO con select()

- ▶ El cierre de conexión es un caso de lectura !!  
Recordemos que era lo mismo que fin de fichero
- ▶ Si leemos algo lo enviamos si no cerrar conexión

\*/

```
if (select(fd_maximo,&paraleer,NULL,NULL,NULL)>0) {
    /* Hay algo que leer en alguno */
    if (FD_ISSET(sock,&paraleer)) {
        /* Conexion para aceptar */
        aceptar_nuevo_cliente(sock);
    }
    for (i=0;i<N_DESCRIPTORES;i++) {
        if (descriptores[i]!=-1) {
            if (FD_ISSET(descriptores[i],&paraleer)) {
                int leidos;
                /* hay algo para leer en el i */
                leidos=read(descriptores[i],buf,20000);
                if (leidos>0) {
                    write(descriptores[i],buf,leidos);
                } else {
                    close(descriptores[i]);
                    descriptores[i]=-1;
                    numclientes--;
                    printf("Ahora hay %d clientes\n",numclientes);
                }
            }
        }
    }
}
```

leer

echo

cierre y

actualiza estado

```
} else {
```

# Ejemplo: servidor de ECHO con select()

- ▶ Aceptar un cliente es actualizar el estado
- ▶ Como usamos un array podemos no tener sitio para un nuevo cliente

```
void aceptar_nuevo_cliente(int sock) {  
    int c_sock;  
    struct sockaddr_in cliente;  
    int dirlen=sizeof(cliente);  
    int i;
```

```
    c_sock=accept(sock,(struct sockaddr*)&cliente,&dirlen);
```

**accept()**

```
    if (numclientes<N_DESCRIPTORES) {
```

```
        /* Todavía hay sitio */
```

```
        numclientes++;
```

```
        printf("Ahora hay %d clientes\n",numclientes);
```

```
        for (i=0;i<N_DESCRIPTORES;i++) {
```

```
            if (descriptores[i]==-1) {
```

```
                descriptores[i]=c_sock;
```

```
                break;
```

```
            }
```

```
        }
```

```
    } else {
```

```
        char cadena[100];
```

```
        /* Ya no queda sitio */
```

```
        sprintf(cadena, "Lo siento, el servidor está lleno\n");
```

**busca un hueco libre  
y apunta el socket**

# Servidores síncronos con poll

- ▶ También podemos programar servidores concurrentes utilizando un solo proceso
- ▶ Función `poll()` permite saber que descriptor de fichero (socket) tiene algo para leer
  - > Para ello le indicamos un conjunto de descriptors en los que estamos interesados
- ▶ Manejo de conjuntos de descriptors
  - > Tipos y macros definidos en `#include <poll.h>`
  - > Tipo struct `pollfd` almacena un descriptor de ficheros y sus eventos asociados
  - > Un array de structs `pollfd` con todos los descriptors que nos interesan y `poll` nos dice en cuales ha pasado algo interesante

# Servidores síncronos con poll

- ▶ Funcion poll()

Que descriptores tienen algo que decir...

**fds** array de struct pollfd

En la entrada dicen en que descriptores y que eventos estamos interesados

En la salida indica los eventos que han ocurrido

**nfds** numero de descriptores en el array

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

**n**

numero de descriptores con novedades

0: no hay novedades

-1: error

**timeout**

tiempo a esperar por novedades (milisegundos)

si pasa ese tiempo y no hay novedades devuelve 0

0 = no esperes

-1 = si no hay novedades espera indefinidamente

# Conclusiones

- ▶ Los servidores que atienden varios clientes simultáneos implican uso de técnicas concurrentes o de entrada/salida síncrona
- ▶ Ya hemos visto el uso básico de sockets TCP

## Próxima clase...

- ▶ Y que pasa con UDP
- ▶ Cuestiones avanzadas de sockets