TEMA 2
Técnicas básicas de construcción de aplicaciones distribuidas

Lección 2
Modelos de aplicaciones distribuidas

Modelos de aplicaciones distribuidas

- Una aplicación distribuida sería una pieza de software que se ejecuta en máquinas distintas, concurrentemente, cada una de las partes se comunica con las otras y puede realizar un trabajo o tarea.

- Dependiendo de la forma de interacción entre las partes, tenemos distintos modelos de aplicaciones:
  - Orientado a los mensajes.
  - Redes entre iguales (p2p), computación en malla (Grid).
  - Cliente/Servidor.

Modelos de aplicaciones distribuidas

- Orientado a los mensajes.- Para aplicaciones que pueden tolerar cierto nivel de independencia frente al tiempo de las respuestas. Los mensajes se intercambian mediante *colas de mensajes*. Al no comunicarse directamente, los programas pueden estar ocupados, no disponibles o simplemente no ejecutándose cuando se produce la comunicación.
- Redes entre iguales (p2p), computación en malla (Grid).- Todos los miembros del sistema son iguales. No hay tareas predefinidas entre ellos. Comparten recursos: disco (emule, kazaa, edonkey, etc) o CPU (seti@home)
- Cliente/Servidor.- Las aplicaciones se dividen en dos partes: una de ellas (cliente) inicia la comunicación con una *petición* y la otra (servidor) *responde* a esa petición. El servidor se ha tenido que ejecutar antes que el cliente y normalmente suministra un *servicio*.

El modelo cliente/servidor

- **Un servidor es una pieza de software que espera por peticiones de clientes, las procesa y envía el resultado al cliente.**
  - **Entre cliente y servidor se utiliza el modelo de comunicación petición/respuesta.**
  - **El servidor ha sido activado siempre antes que el cliente y *espera pasivamente* por las solicitudes de servicio.**
- **Un servicio es cualquier tarea que el cliente pueda invocar de manera independiente.**
  - **Un servidor puede suministrar múltiples servicios.**

    Ejemplo: Servidor de ficheros.
    - **Crear fichero**
    - **Borrar fichero**
    - **Leer fichero**
    - **etc……**

Tipos de servicios

Dependiendo de la implementación podemos distinguir dos tipos de servicios:

Idempotente.- Cuando dos peticiones con los mismos datos de entrada generan exactamente el mismo valor de salida.

El servicio no mantiene ninguna información de "estado" sobre los clientes que contactan con él. Todas las peticiones son iguales.

No idempotente.- Cuando dos peticiones con los mismos datos de entrada pueden generar valores de salida distintos.

El servicio suele mantener información de "estado" sobre los clientes que contactan con él para mejorar la eficiencia del servicio. El servicio *reconoce* a los clientes.

El mismo servicio se puede implementar de ambas formas.
Ej: Servidor de Ficheros

Lección 3
La interfaz de *sockets*

Historia y diseño

Diseñado en 1981 en la Universidad de California, Berkeley para incorporarse al sistema operativo BSD Unix.

Idea principal: Usar las llamadas al sistema operativo siempre que fuera posible. Añadir nuevas llamadas sólo si era necesario.

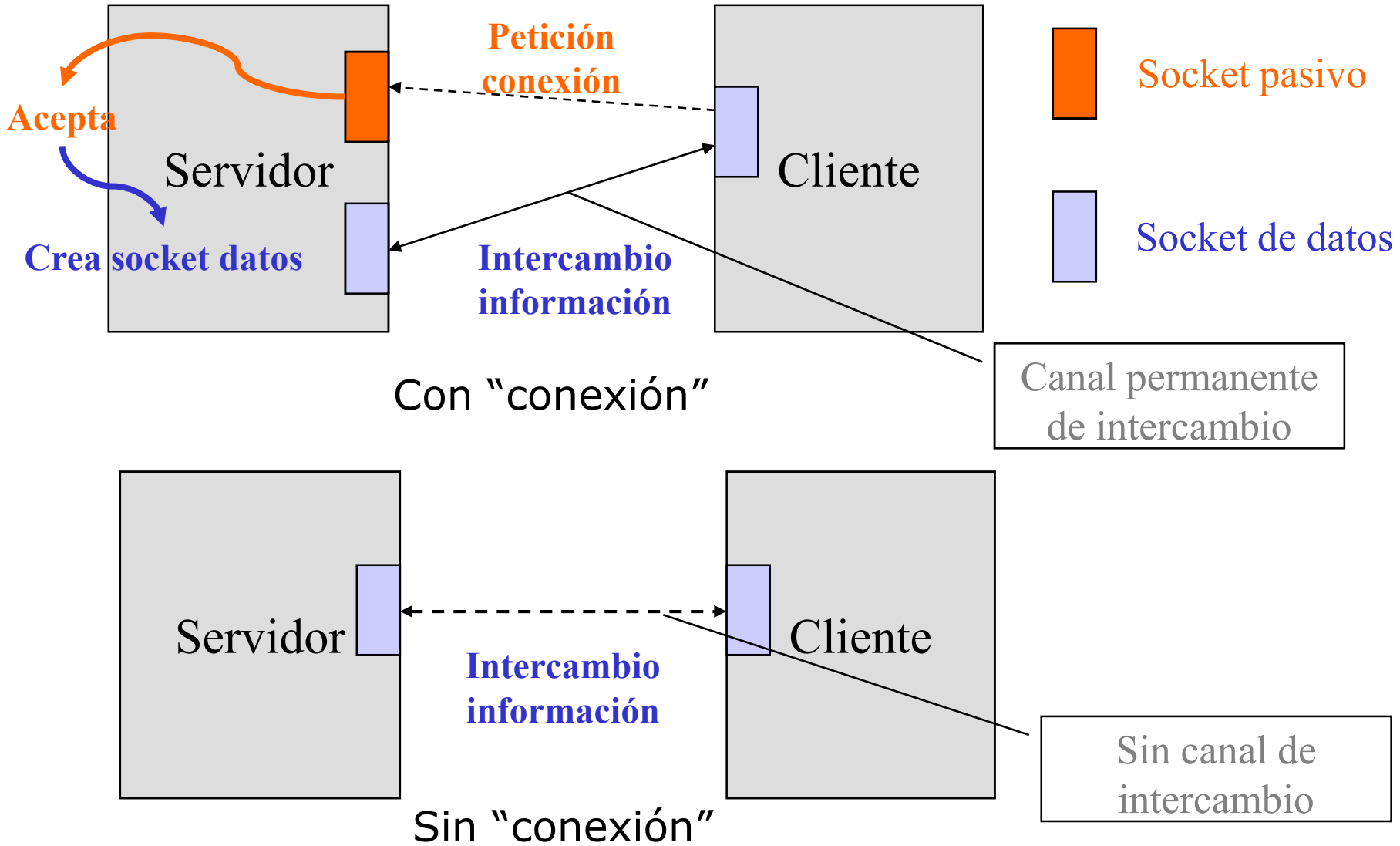
Actualmente, es un estándar de facto por lo que el diseño original se encuentra ligeramente modificado en los distintos sistemas operativos.

Un socket es una abstracción del sistema operativo que representa un extremo de una comunicación entre dos procesos.

Existen dos tipos de sockets:

- Datos. Se pueden transmitir y recibir datos a través de ellos.
- Pasivos (Escucha). Sólo se pueden recibir peticiones de conexión a través de ellos.

El modelo cliente/servidor en los sockets



Diseño

Para establecer una comunicación, para cada uno de los procesos que intervienen, necesitamos:

- Dirección de la máquina en la que se encuentra
- Localización del proceso dentro de la máquina

Ambos localizadores están muy asociados al protocolo de comunicaciones utilizado.

Protocolo de internet IP (transporta paquetes entre máquinas):

- Localización de la máquina:
  entero de 32 bits (156.35.151.2)
- Localización del proceso:
  número de 16 bits (puerto)

Diseño

Resumiendo, para cada extremo de una comunicación, usando IP necesitamos:

- Dirección IP de la máquina
- Identificación protocolo de transporte utilizado
- Número de puerto

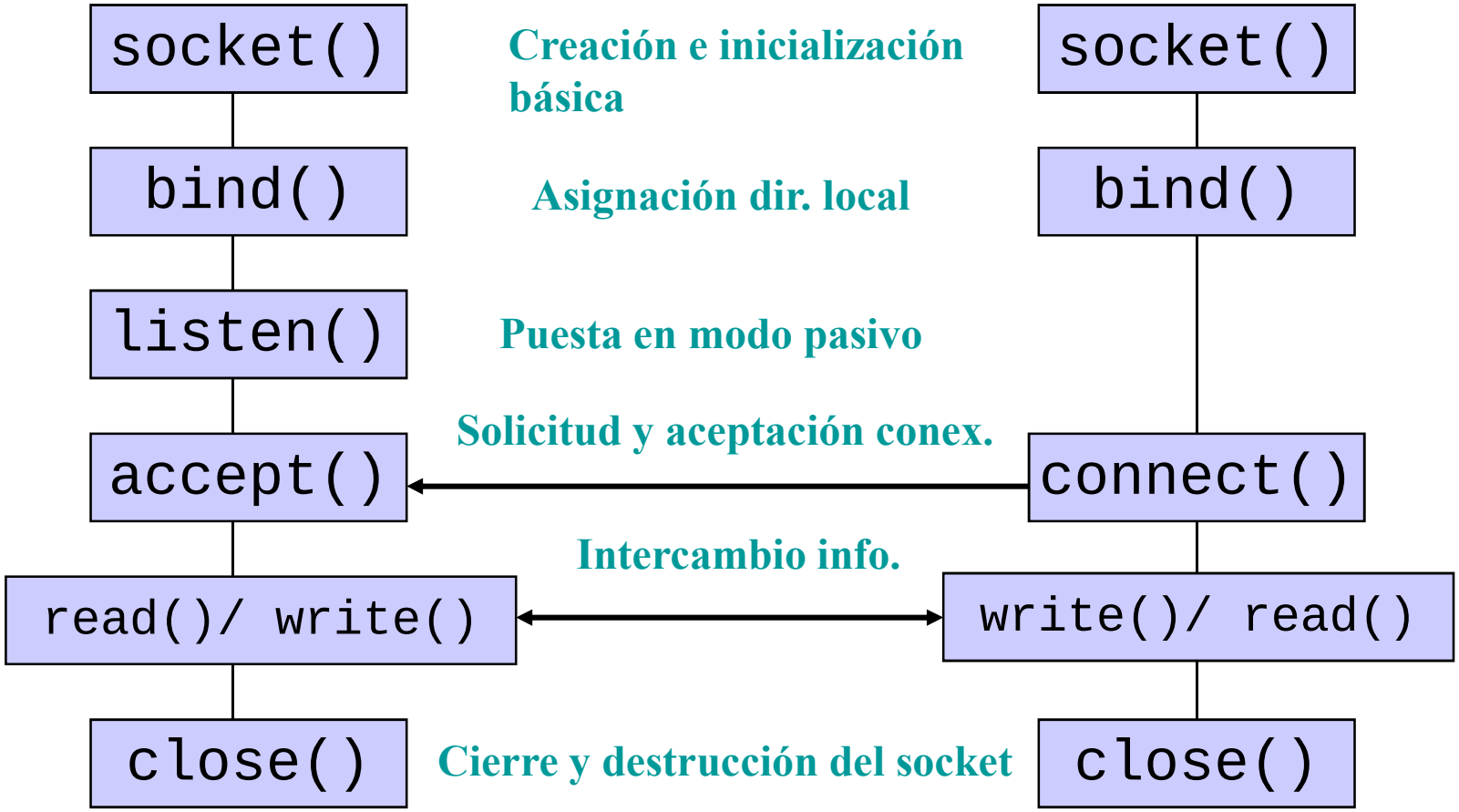
Una comunicación está establecida cuando se conocen los siguientes cinco datos:

<protocolo, IP máq. local, puerto local, IP máq. remota, puerto remoto>

La API de sockets básicamente se dedica a asignar esos valores a un socket para luego enviar o recibir datos a través de él.

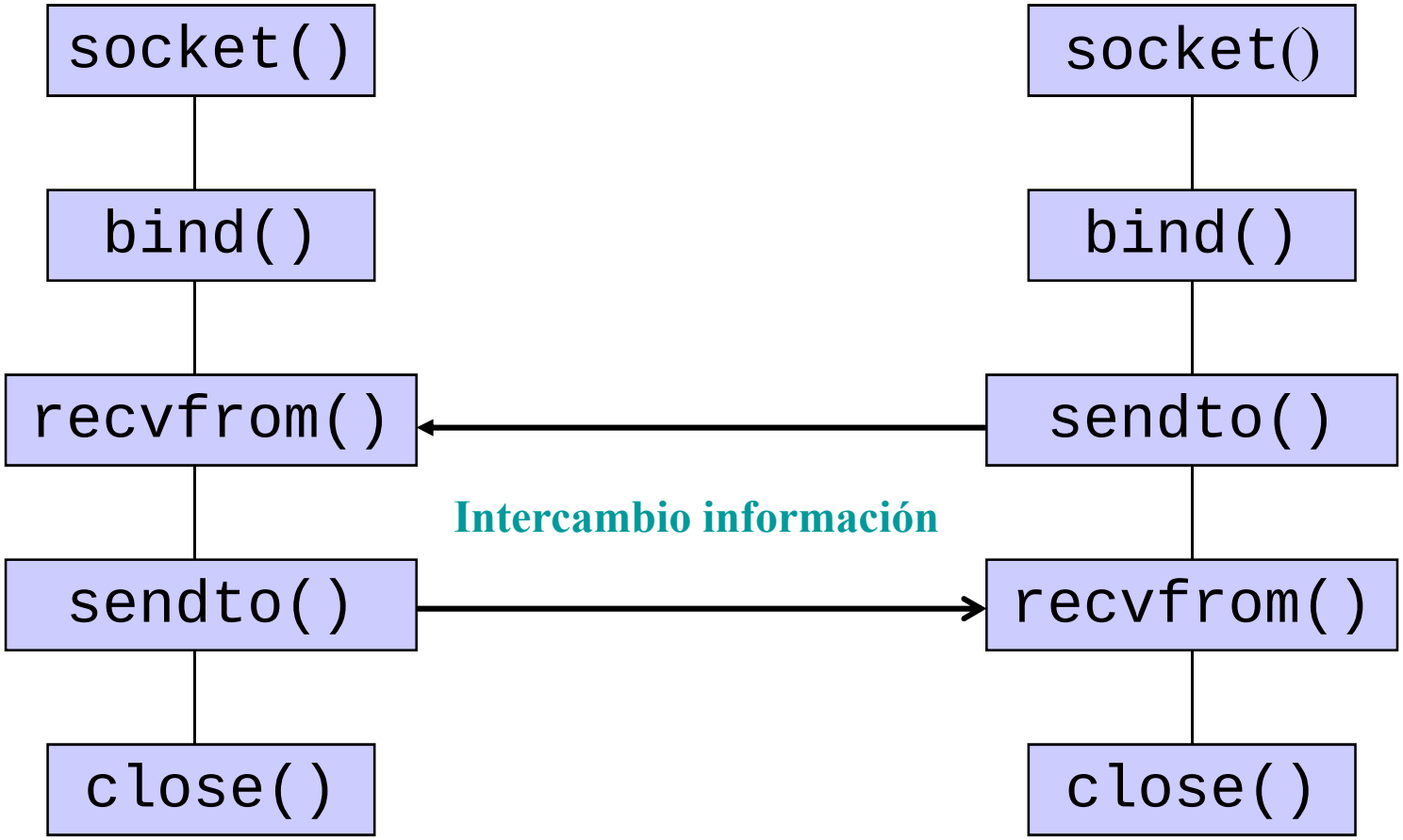
La API

Con conexión



La API

Sin conexión



Los sockets y Unix

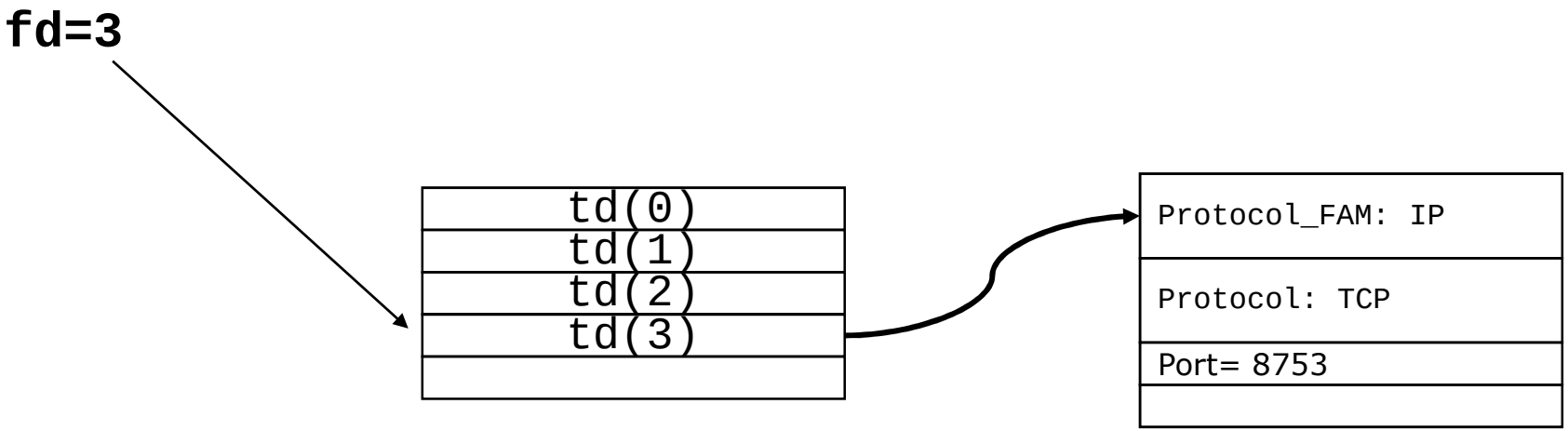
Por diseño, para reutilizar las llamadas al sistema:

socket ≡ descriptor de fichero

Los descriptores de ficheros son enteros, luego socket ≡ entero.

Dónde se puede usar un descriptor de fichero se puede usar un socket (NO implica resultados correctos).

Un descriptor de fichero es un índice en una tabla que nos lleva a las estructuras de control del fichero. Ej:



Ficheros de cabecera necesarios

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

Prototipo de socket()

```
s = socket (familia, servicio, protocolo);
int s, familia, servicio, protocolo;
```

Valores típicos:

```
 familia:   PF_INET
 servicio:  SOCK_DGRAM; /* sin conexión */
            SOCK_STREAM; /* con conexión */
 protocolo: 0
```

Funcionalidad de `bind()`

Asigna *dirección local* al socket:

`<protocolo, dir. final local, dir. final remota>`

Caso de IP:

Dir. final = **`<IP máq. local, puerto>`**

Socket completo:

```
<protocolo, IP máq. local, puerto local,
 IP máq. remota, puerto remoto>
```

Prototipo de `bind()`

```
err = bind (s, direc_fin, lon_dir);
int s, lon_dir;
struct sockaddr *direc_fin;
```

s = socket

direc_fin = (ver siguiente)

lon_dir = tamaño del struct direc_fin

La estructura sockaddr_in

```
struct sockaddr_in {
          short sin_family;
unsigned short sin_port;
struct in_addr sin_addr;
           char sin_zero[8];
}

struct in_addr {
     unsigned long s_addr;
}
```

Inicialización de sockaddr_in

Ejemplo:

```
struct sockaddr_in local;
     ....
local.sin_family = AF_INET;
local.sin_port = htons (15000);
local.sin_addr.s_addr = htonl (INADDR_ANY);
```

Ejemplo asignando IP

```
local.sin_addr.s_addr = inet_addr("156.35.151.2");
```

Inicialización de `sockaddr_in`

Funciones de manejo de enteros:

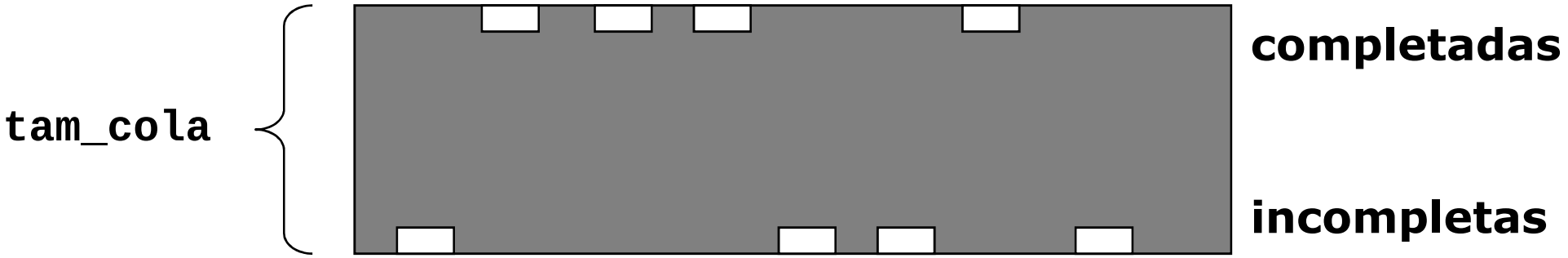
`htons`	➔ Host to network short. 16 bits
`htonl`	➔ Host to network long. 32 bits
`ntohs`	➔ Network to host short. 16 bits
`ntohl`	➔ Network to host long. 32 bits

Prototipo de `listen()`

```
err = listen (s, tam_cola);
int s, tam_cola;
```

Valores típicos:

tam_cola: SOMAXCONN



Prototipo de `accept()`

```
n_sock = accept (s, quien, l_quien);
int n_sock, s, *l_quien;
struct sockaddr *quien;
```

Usos típicos:

```
ns = accept (s, &dir_remota, &l_dir);
ns = accept (s, NULL, NULL);
```

Nota: Si no necesitamos saber la dirección, podemos pasar **`NULL`**. De lo contrario, **`l_quien`** debe apuntar a un entero *correctamente inicializado*.

Prototipo de `connect()`

```
err = connect (s, dir_serv, lon_dir);
int s, lon_dir;
struct sockaddr *dir_serv;
```

Inicialización de **`dir_serv`**: véase **`bind()`**

Prototipos de `write()/read()`

```
nbytes = write (s, buffer, l_buffer);
int s, nbytes, l_buffer;
char * buffer;

nbytes = read (s, buffer, l_buffer);
int s, nbytes, l_buffer;
char * buffer;
```

nbytes
- *< 0* ➔ *error*
- *> 0* ➔ *bytes leídos*
- *= 0* ➔ *conexión cerrada*

Prototipos de sendto()/recvfrom()

```
ret = sendto (s,buffer,l_buf,flags,destino,l_destino);
ret = recvfrom (s,buffer,l_buf,flags,origen,l_origen);

int s, l_buf, flags, l_destino, *l_origen, ret;
char * buffer;
struct sockaddr *destino, *origen;
```

¡Cuidado! **l_origen** es un puntero a través del cual se retorna el tamaño de la estructura **origen**. No puede ser null, pero puede apuntar a un cero.

Prototipos de `close()/shutdown()`

```
err = close (s);

Err = shutdown (s, modo);

int s, modo;
```

modo
- = *0* → *No más lecturas.*
- = *1* → *No más escrituras.*
- = *2* → *Ni lecturas ni escrituras.*

Lección 4
Concurrencia en los servidores

Tipos de Servidores

Atendiendo a la persistencia:

- Con estado. Servicios no idempotentes.
- Sin estado. Servicios idempotentes.

Atendiendo a la tecnología:

- Orientados a la conexión.
- Orientados a los datagramas.

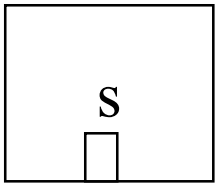
Atendiendo a la temporización:

- Servidores iterativos. Atiende a los clientes de uno en uno, secuencialmente.
- Servidores concurrentes. Atiende a varios clientes simultáneamente.
  - ❖ Concurrencia real
  - ❖ Concurrencia aparente

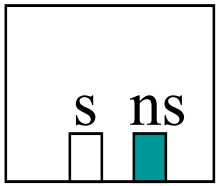
Concurrencia Real

Un proceso independiente por petición. Secuencia:

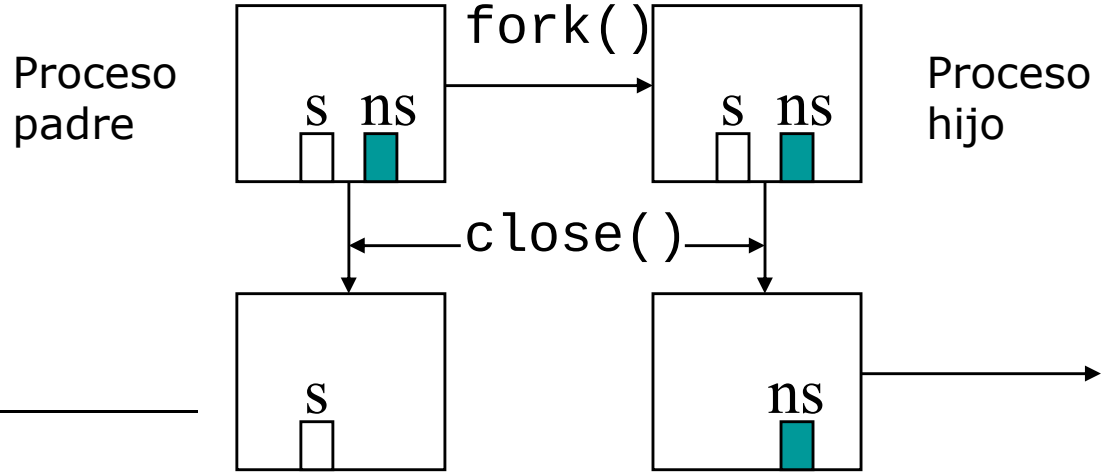
1. Crear socket



2. Aceptar petición con `accept()`



3. Crear proceso con `fork()`



4. Atender al cliente y salir con `exit()`

Concurrencia Real. Problemas

Problema: El problema de los procesos errantes (*zombies*):

Solución:

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

Problema: Con `fork()`, el código del padre y el del hijo tienen que ser iguales.

Solución:

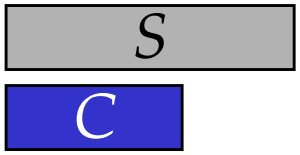
`fork()` → `execve()`

Concurrencia Real ¿Merece la pena?

El coste de la concurrencia:

S: Tiempo de servicio

C: Tiempo de creación de procesos

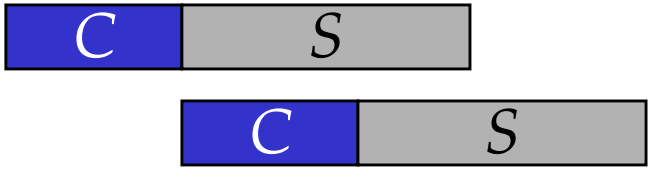


Dos peticiones consecutivas de servicio.

- Servidor Iterativo

Tiempo total= $2 * S$

- Servidor concurrente



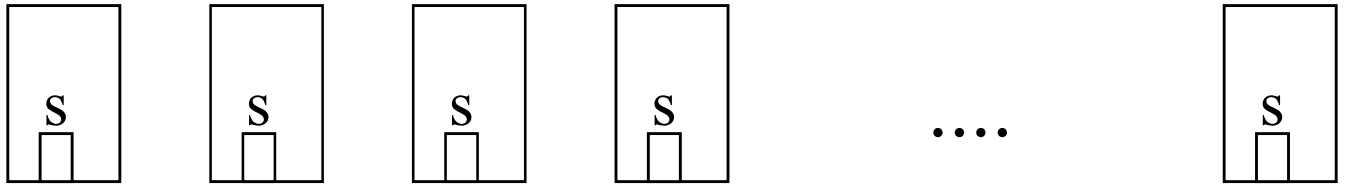
Tiempo total= $2 * C + S$

Si $C \approx S$ no hay ganancia con la concurrencia a menos que....

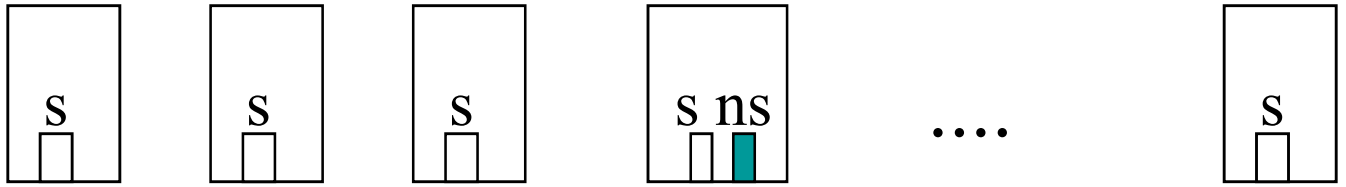
Concurrencia Real. Creación previa

Creación previa de servidores:

1. Creación del socket
2. Llamadas a `fork()`



3. Bloqueo generalizado en `accept()`
4. Al llegar una petición, el S.O. desbloquea todos los procesos. Sólo uno aceptará la conexión.



Concurrencia con hilos

Conceptos

- Un hilo de ejecución es un “proceso ligero”.
- Cada hilo tiene su propio contador de programa (qué ejecuta) y puntero de pila (variables locales, parámetros)
- Todos los hilos **comparten** el resto de la memoria (variables globales), descriptores abiertos, recursos... y se pueden comunicar entre sí a través de estos recursos.
- El conjunto de hilos que comparte memoria es un proceso.
- Un hilo se crea con una llamada específica a la API de hilos
- El hilo comienza su ejecución por una función que especifiquemos, y termina cuando esa función retorna.
- ¡El hilo no debe llamar a exit() pues finalizaría todo el proceso!

Concurrencia con hilos

Creación del hilo (biblioteca pthreads)

```
ret=pthread_create(hilo, attr,
                   rutina_inicio, param);
```

```
int ret;
pthread_t *hilo;
pthread_attr_t *attr;
void *(*rutina_inicio)(void*);
void *param;
```

- Código de error
- Manejador del hilo creado
- Usar NULL
- Puntero a función de inicio del hilo
- Parámetro a pasar a esa función

Concurrencia con hilos

Terminación del hilo (biblioteca pthreads)

```
pthread_exit(puntero_a_dato);
void *puntero_a_dato;
```

...que el "padre" puede recibir si llama a pthread_join()

- Si no se llama a pthread_exit(), el hilo es terminado de todas formas al hacer return() o finalizar la rutina.
- Pero si no se llama desde main() , el proceso principal finaliza sin esperar por otros hilos.

Concurrencia con hilos

Ejemplo

```
#include <pthread.h>

void *ejemplo(int *dato)
{
   int i;
   for (i=0; i<*dato; i++)
     printf("Hilo imprime %d\n", i);
}
main()
{
  pthread_t hilo1, hilo2;
  int dato1=5, dato2=8;

  pthread_create(&hilo1, NULL, ejemplo, &dato1);
  pthread_create(&hilo2, NULL, ejemplo, &dato2);
  pthread_exit(NULL);
}
```

Concurrencia con hilos

Uso en servidores concurrentes

- Las ideas antes vistas con fork() son aplicables a hilos
- Pero hay una diferencia importante:
  - Usando fork(), cada hijo es un proceso. Cada uno tiene su propia zona de variables, independiente.
  - Usando threads, todos los hilos corren en el mismo proceso. Las variables globales son compartidas.
  - No debe accederse a las variables globales sin mecanismos de exclusión mutua (funciones pthread_mutex*)

Concurrencia Aparente

- Un único proceso.
- Evita bloquearse, salvo cuando no hay petición que atender:
  - No realiza llamadas a `accept()` bloqueantes.
  - No realiza llamadas a `read()` si no hay datos.

Posibilidades:

1. Usar `ioctl()` para hacer las llamadas a `accept()` y `read()` no bloqueantes.
2. Usar `select()`.

Concurrencia Aparente. La función `select()`

¿Qué puede hacer `select()`?

- Puede observar un conjunto de sockets esperando por la llegada de datos.
- Puede observar otro conjunto de sockets esperando a que estén listos para enviar datos.
- Puede esperar por condiciones excepcionales (Ctrl-C, p.e.) en algunos sockets.
- Puede hacer todo lo anterior simultáneamente, por tiempo indefinido o durante una cantidad de tiempo prefijada.
- Informa del número de sockets que cumplen alguna de las condiciones de observación.

Prototipo de `select()`

```
retcod = select (maxfd, lect_fd, esc_fd, exc_fd, tiempo);


int retcods, maxfd;
struct fd_set *lect_fd, *esc_fd, *exc_fd;
struct timeval *tiempo;
```

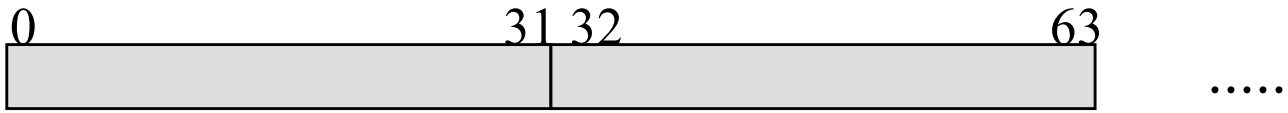
```
struct timeval {
        long tv_sec;      /* segundos */
        long tv_usec;     /* microsegundos */
}
```

Si `tiempo` es `NULL`, `select` espera por tiempo indefinido.
En otro caso, espera por el tiempo indicado.

Prototipo de `select()`

```
retcod = select (maxfd, lect_fd, esc_fd, exc_fd, tiempo);
int maxfd; /* Descriptor más alto a observar (+1) */
struct fd_set *lect_fd, *esc_fd, *exc_fd;
```

Las estructuras `fd_set` permiten indicar qué sockets hay que vigilar:



Macros para el manejo de estructuras `fd_set`:

```
FD_ZERO ( &fdset);     /* Inicializa a cero */
FD_SET (fd, &fdset);   /* Pone a 1 el bit (socket) fd*/
FD_CLR (fd, &fdset);   /* Pone a 0 el bit (socket) fd*/
FD_ISSET (fd, &fdset);/* Comprueba si el bit indicado
                         por fd está a 1 */
```

Ejemplo de uso de `select()`

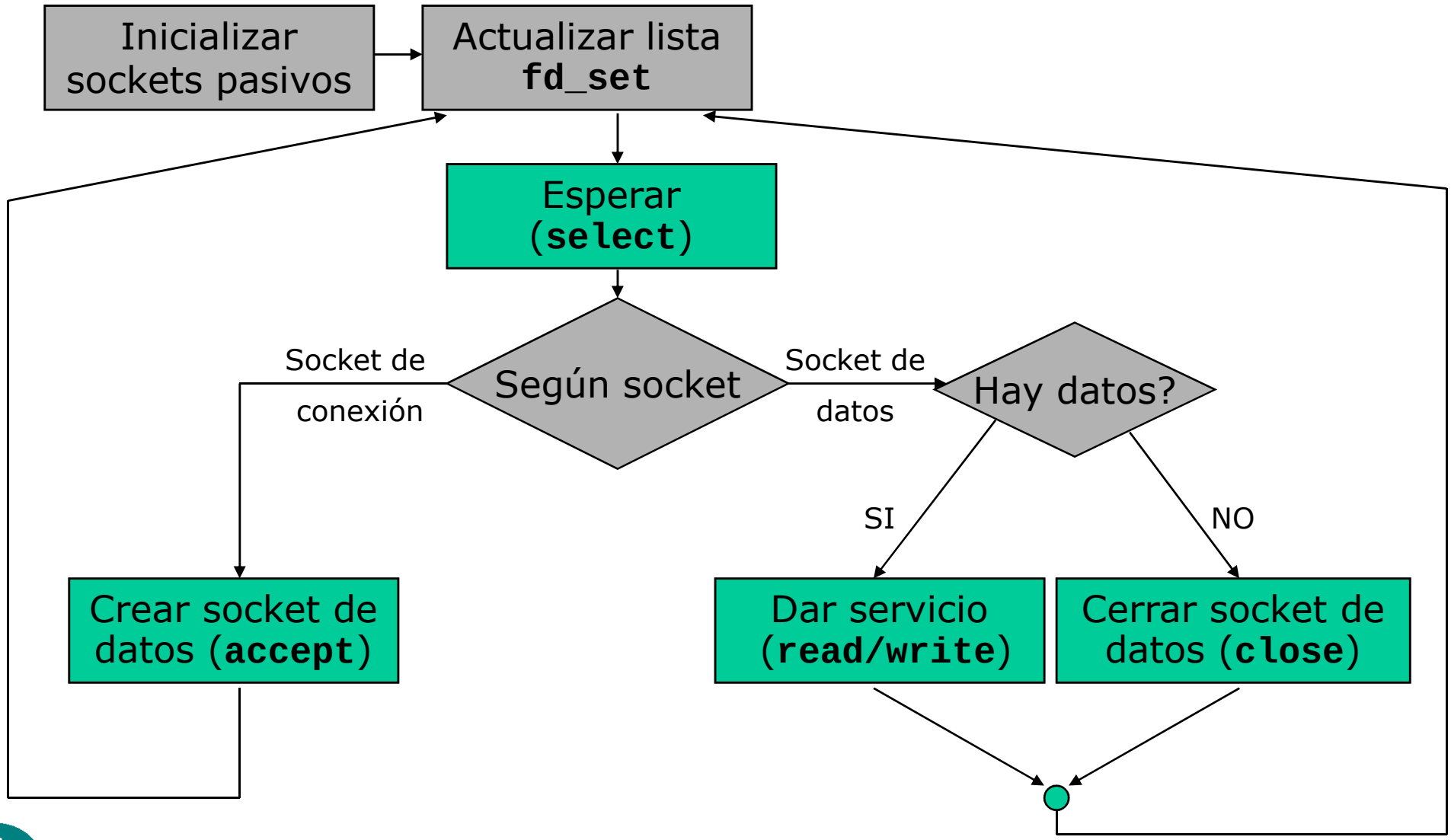
```
            ....
int s1, s2, maxfd, ns;
fd_set selector;
struct timeval tiempo;
      .......
      s1=socket(....);  s2=socket(....);
      listen(s1,...); listen(s2,...);
maxfd=s1>s2?s1+1:s2+1;  FD_ZERO(selector);
FD_SET(s1,&selector); FD_SET(s2, &selector);
tiempo.tv_sec=15; tiempo.tv_usec=500000;
retcod = select (maxfd, &selector,NULL,NULL, &tiempo);
```

Ejemplo de uso de `select()`

```
retcod = select (maxfd, &selector,NULL,NULL, &tiempo);
if (FD_ISSET(s1, &selector) ) {
      ns = accept(s1, NULL, NULL);
      ....
}
if (FD_ISSET(s2, &selector) ) {
      ns = accept(s2, NULL, NULL);
      ....
}
```

¡Cuidado! La estructura **`fd_set`** cambia su valor tras el retorno de **`select`**. No se puede reutilizar sin inicialización.

Concurrencia aparente. Esquema



Análisis del interfaz de sockets.

Problemas de la interfaz de sockets:

1. Determinación de la dirección del servicio. Hay que conocer la dirección IP y el puerto de protocolo.
2. Inicialización de los sockets.
3. Acceso a la información. Representación heterogénea entre distintas máquinas.
4. Envío y recepción de mensajes. Problemas de lecturas y escrituras recortadas.
5. Gestión de los errores. Basada en la variable global `errno`. No hay distinción entre sockets orientados a la conexión y a los datagramas. Un socket es un entero.
6. Portabilidad.
7. Seguridad.