Tema 5
El Modelo Cliente-Servidor

Capítulos:

- **Conceptos básicos.**
- **Características y estructura de un cliente.**
  - **Obtención de información.**
  - **Algoritmos del cliente TCP y UDP.**
  - **Ejemplos.**
- **Características y estructura de un servidor.**
  - **Clases de servidores.**
  - **Algoritmo de un servidor secuencial UDP.**
  - **Algoritmo de un servidor secuencial TCP.**
  - **Algoritmo de un servidor concurrente UDP.**
  - **Algoritmo de un servidor concurrente TCP.**
  - **Otros servidores.**

Bibliografía
[COM00] "Internetworking with TCP/IP", vol. 3, *Cap. 2 y del 7 al 15.*

Conceptos básicos.

- El TCP/IP y los sockets permiten la **comunicación entre** dos **aplicaciones** de forma local o remota.
- Las aplicaciones deben establecer la forma en la que van a distribuir el trabajo entre las máquinas participantes.
- El modelo **Cliente-Servidor** define la estructura de las aplicaciones comunicantes y su sincronización.
  - Las aplicaciones se dividen en dos partes: El **cliente** y el **servidor**.
  - La sincronización se resuelve haciendo que el servidor se ejecute de forma indefinida, esperando al cliente.

Clientes

- **Cliente**: Programa que inicia la comunicación y que es dirigido por el usuario.
  - Sencillos, no requieren privilegios del sistema.
- Cliente **estándar** frente a **no estándar**.
  - Los clientes estándar invocan los servicios bien-conocidos de TCP/IP (e-mail, terminal remota, etc.).
  - Los clientes no estándar, utilizan servicios localmente conocidos (ej.: automatrícula, acceso bibliotecas, etc.).

Servidores

- **Servidor**: Está ejecutándose indefinidamente a la espera de peticiones de servicio de los clientes.
  - Necesitan acceder a recursos del sistema (privilegios de acceso), son más complejos y robustos. Deben tener en cuenta:
    - Autenticación.
    - Autorización.
    - Seguridad y privacidad de la información.
    - Protección del sistema donde reside.
- Servidores **con conexión** frente a **sin conexión**.
  - Los primeros utilizan los servicios de TCP y los segundos utilizan los de UDP.
  - ¿Cuándo elegir uno u otro?
    - Depende de los requerimientos de la aplicación:
      - Fiabilidad, entrega ordenada, entorno de la aplicación (local, remoto), “overhead” computacional de los protocolos, el uso de multicast y/o broadcast, etc.

Información de estado.

- Servidores con información de estado.
  - Mantienen información del estado del servicio con sus clientes, haciendo un trabajo lo más eficiente posible.
    - **Ventajas**:
      - Los mensajes que intercambian con los clientes son más cortos.
      - La velocidad de respuesta se eleva.
    - **Inconvenientes**:
      - La información de estado consume recursos.
      - Si no hay fiabilidad (UDP) la información de estado puede ser incorrecta.
      - Incrementa la complejidad del servidor.
    - **Ejemplo**: **Un servidor de ficheros**.
      - Los clientes envían peticiones al servidor para leer o escribir en ficheros. Por cada cliente se guarda información del estado de los últimos accesos.

Características y estructura de un cliente

- Son más sencillos que los servidores (excepción → navegador web).
- En general, no requieren privilegios del sistema.
- No necesitan dialogar con varios servidores, normalmente sólo lo hacen con uno.
- El desarrollo de clientes suele ser bastante rápido, una vez se conoce el protocolo (diálogo) que debe utilizar para comunicarse con el servidor.
- Dos clases de clientes:
  - Con conexión (TCP).
  - Sin conexión (UDP).

Obtención de información

- ¿ Cómo se obtiene la dirección del servidor ?
  - El usuario pasa esta información en la línea de comandos cuando ejecuta la aplicación cliente.
    - Ej.: $ ftp 193.147.136.95
  - La dirección del servidor se encuentra en un fichero local (fichero de configuración del cliente).
  - Se utiliza otra aplicación para encontrar el servidor que requiere el cliente.

- ¿ Y si sólo se conoce el nombre del servidor ?
  - A cada dirección IP se le puede asignar un nombre.
    - Ej.: 193.147.136.95 ←→ "obelix.umh.es"
  - Servicio de traducción de nombres a direcciones IP: DNS.

Resolución de nombres

- Existen funciones que permiten invocar el servicio de nombres: ***Gethostbyname*** y ***Gethostbyaddr***.

```
#include <netdb.h>
struct hostent * gethostbyname (char *name);
struct hostent * gethostbyaddr (char *addr,
int addrlen,int addrtype);

struct hostent {
   char *h_name;      /* official name of host */
   char **h_aliases;  /* alias list */
   int  h_addrtype;   /* host address type */
   int  h_length;     /* length of address */
   char **h_addr_list; /* list of addresses */
};
#define   h_addr  h_addr_list[0]
```

Buscan la información en el fichero "/etc/hosts". Si no la encuentra, inicia una búsqueda sobre un servidor de nombres (DNS).

Información asociada a un puerto

- También existen nombres de servicio asociados a direcciones de puertos bien-conocidas.
  - Funciones ***Getservbyname*** y ***Getservbyport***.

```
#include <netdb.h>
struct servent * getservbyname (char *name, char *prot);
struct servent * getservbyport (int port, char *prot);

struct servent {
  char    *s_name;        /* official service name */
  char    **s_aliases;    /* alias list */
  int     s_port;         /* port number */
  char    *s_proto;       /* protocol to use */
}
```

Buscan la información en el fichero "/etc/services" (en W9x esta información se encuentra en "c:\windows\services")

Información asociada a protocolos

- Cada protocolo tiene asociado un identificador que lo distingue de otros en el sistema.
  - Funciones: ***Getprotobyname*** y ***Getprotobynumber***.

```
#include <netdb.h>
struct protoent * getprotobyname (char *name);
struct protoent * getprotobynumber (int proto);

struct protoent {
   char    *p_name;        /* official protocol name */
   char    **p_aliases;    /* alias list */
   int     p_proto;        /* protocol number */
}
```

La información de los protocolos de un sistema está almacenada en el fichero "/etc/protocols".

Algoritmos de clientes TCP y UDP

Algoritmo de un cliente TCP

1. Obtener la dirección IP y el número de puerto donde estará esperando el servidor con el que queremos contactar, y construir la dirección de *socket* remota.

2. Crear un *socket* de tipo SOCK_STREAM (SOCKET)

3*. Especificar la dirección del *socket* local que acabamos de crear, dejando que el sistema seleccione un puerto disponible (asignación dinámica). (BIND)

4. Establecer la conexión con el servidor. (CONNECT)

5. Protocolo de aplicación. Dialogo entre cliente y servidor basado en mensajes petición/respuesta. (READ, WRITE, SEND, RECV)

6. Cerrar la conexión, eliminando el *socket*. (CLOSE, SHUTDOWN)

(*) opcional

Consideraciones sobre el algoritmo

- **Paso 5**: Recepción de datos en TCP
  - TCP puede agrupar o dividir los mensajes que le pasamos.
  - La llamada *read* quedará bloqueada intentando obtener tantos bytes como quepan en su buffer.
    - Sin embargo, la llegada de un segmento con el bit PSH activo hará que se desbloquee.
  - También es necesario verificar si ha habido algún error (*read* devuelve -1) ó si se ha cerrado la conexión (*read* devuelve 0)
- **Paso 6**: Cierre de una conexión TCP
  - En algunos casos puede existir ambigüedad:
    - Los servidores no deciden el cierre de conexión, esperan recibir las peticiones de los clientes.
    - Los clientes no saben cuando deben cerrar si las respuestas del servidor son de tamaño variable.
  - Para cerrar la conexión se requiere una sincronización entre ambas aplicaciones:
    - Con un mensaje explícito (FTP: BYE).
    - Cerrando parcialmente la conexión (SHUTDOWN).

Algoritmo de un cliente UDP

Algoritmo de un cliente UDP

1. Obtener la dirección IP y el número de puerto donde estará esperando el servidor con el que queremos contactar, y construir la dirección de *socket* remota.

2. Crear un *socket* de tipo SOCK_DGRAM (SOCKET)

3. Especificar la dirección del *socket* local que acabamos de crear, dejando que el sistema seleccione un puerto disponible (asignación dinámica). (BIND)

4*. Especificar el servidor con el que queremos trabajar. (CONNECT)

5. Protocolo de aplicación. Dialogo entre cliente y servidor basado en mensajes petición/respuesta. (SENDTO, RECVFROM, READ*, WRITE*, SEND*, RECV*).

6. Eliminar el *socket*. (CLOSE)

(*) opcional

Consideraciones sobre el algoritmo

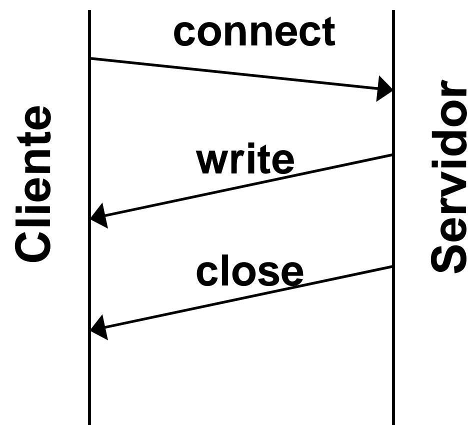
- **Paso 4**: *CONNECT* sobre un *socket* UDP (opcional).
  - Define la asociación entre un cliente y servidor UDP.
    - Comunicación restringida al servidor asociado.
    - No conlleva intercambio de mensajes.
    - Se pueden usar *READ*, *WRITE*, *SEND* y *RECV*.
- **Paso 5**: Transferencia compacta.
  - Por cada *SENDTO* se envía todo el mensaje en un solo datagrama, pudiendo recogerlo con un solo *RECVFROM*.
- **Paso 6**: Fin de servicio.
  - Al eliminar un *socket*, el protocolo UDP no informa al otro extremo, rechazando los mensajes dirigidos al puerto que estaba asociado a ese *socket*.

Procedimiento ConnectSock

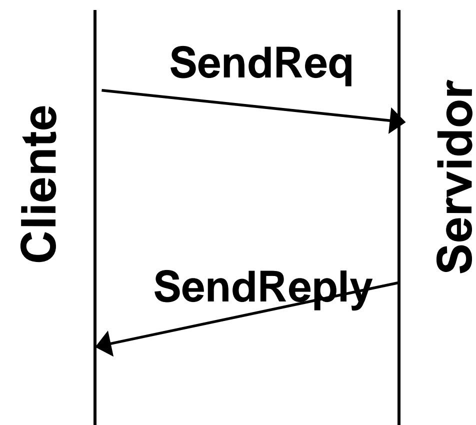
```
int connectsock (host,service,protocol)
char *host, *service, *protocol;
{     struct hostent        *phe;
      struct servent        *pse;
      struct protoent       *ppe;
      struct sockaddr_in    sin;
      int                   s,type;
memset ((char *)&sin,0, sizeof (sin));
sin.sin_family = AF_INET;

if (pse = getservbyname (service, protocol))
    sin.sin_port = pse->s_port;
else
  if  ((sin.sin_port = htons ((u_short) atoi (service))) == 0)
     errexit ("can't get %s service entry\n", service);

if (phe = gethostbyname (host))
   memcpy (phe->h_addr, (char *)&sin.sin_addr, phe->h_length);
else
  if (((sin.sin_addr.s_addr = inet_addr (host)) == INADDR_NONE)
       errexit ("can't get %s host entry\n", host);

if ((ppe = getprotobyname (protocol)) == 0)
   errexit ("Bad %s protocol\n", protocol);

if (strcmp (protocol, "udp") == 0)
      type = SOCK_DGRAM;
else   type = SOCK_STREAM;

s = socket (AF_INET, type, ppe->p_proto);
if (s<0)  errexit ("can'create socket %s\n",
      sys_errlist[errno]);

if (connect (s, (struct sockaddr *)&sin, sizeof (sin))
      <0 )
      errexit ("can't connect to %s.\n", host);
return s; }
```

Servicio DAYTIME

- Servicio estándar de Internet que suministra la fecha y hora actual en una cadena de caracteres [RFC 867].
- Puerto bien-conocido de DAYTIME: 13.
- Servicio implementado sobre TCP y sobre UDP.



Versión TCP



Versión UDP

Cliente TCP del servicio DAYTIME.

```
#include <stdio.h>
#include <socket.h>
#define     LINELEN    128

char *service = "daytime";
char *host = "localhost";

int  main (argc, argv)
int argc;
char *argv[];
{
      int s, n;
      char buff[LINELEN];

      switch (argc) {
        case 1:  host = "localhost";          break;
        case 3:  service = argv[2];
        case 2:  host = argv[1];  break;
        default:
          fprintf (stderr, "Uso:
                DayTime [host [port]]\n");
          exit (1);
      }
```

```
s = connectTCP (host, service);

printf ("\n Fecha y Hora: ");

while (n = read (s, buff, LINELEN) > 0)
      printf ("%s", buff);

close(s);
}
```

```
int connectTCP (host, service)
char *host;
char *service;
{
    return connectsock (host, service, "tcp");
}
```

Cliente UDP del servicio DAYTIME

```
#include <stdio.h>
#include <socket.h>
#define      LINELEN     128
#define  MSG   "¿ Que hora es ?"

char *service = "daytime";
char *host = "localhost";

int  main (argc, argv)
int argc;
char *argv[];
{
      int s, n;
      char buff[LINELEN];
      switch (argc) {
      case 1:  host = "localhost";
      break;
      case 3:  service = argv[2];
      case 2:  host = argv[1];  break;
      default:
          fprintf (stderr, "Uso:
                DayTime [host [port]]\n");
          exit (1);
      }
```

```
s = connectUDP (host, service);

printf ("\n %s: ", MSG);
write (s, MSG, strlen (MSG));

n = read (s, buff, LINELEN);
if (n > 0)  printf ("%s", buff)
    else  errexit ("Read Failed: %s \n",
       sys_errlist[errno];
close(s);
}
```

```
int connectUDP (host, service)
char *host;
char *service;
{
    return connectsock (host, service, "udp");
}
```

Características y estructura de un servidor

- Un **servidor** debe estar dispuesto a dar servicio a un conjunto de clientes.
- Su diseño dependerá de la naturaleza del servicio que ofrece.
- Los servidores necesitan privilegios de acceso a los recursos del sistema.
- Deben ser robustos y seguros.
- Existen diferentes clases de servidores:
  - Secuenciales o concurrentes.
  - Con conexión o sin conexión.

Servidores secuenciales y concurrentes

- Servidores secuenciales.
  - Procesan una solicitud cada vez.
  - Las peticiones que lleguen mientras se está atendiendo a un cliente deben ser encoladas.
- Servidores Concurrentes.
  - Pueden atender a varios clientes al mismo tiempo.
  - La concurrencia puede ser real o aparente.
    - Usando varios procesos (multitarea) o uno sólo (servidores concurrentes de proceso único).
  - Son más complejos y utilizan más recursos del sistema.

Servidores secuenciales y concurrentes

- ¿Cuándo usaremos un servidor secuencial?
  - Cuando el tiempo medio de respuesta que observa el cliente (TR) sea pequeño.
    - TR = (N/2 + 1) * TS + TCOMM
      - Depende del tiempo de servicio (TS), del tamaño de las colas (N) y de la frecuencia de acceso de los clientes.
    - Ejemplos: El DAYTIME es el típico servicio secuencial. Un servidor FTP, ¿podría ser secuencial?.
- ¿Cuándo usaremos servidores concurrentes?
  - Cuando el tiempo medio de respuesta (TR) es grande o desconocido a priori.
  - Cuando el tiempo de servicio de una petición (TS) es muy variable (Ej.: FTP, comandos ***get*** y ***cd***).
  - Cuando la demanda de acceso sea elevada.

Servidores con conexión vs. sin conexión

- Los servidores con conexión utilizan TCP mientras que los servidores sin conexión usan UDP.
- ¿Cuándo usaremos un servidor con conexión?
  - Cuando la aplicación requiera fiabilidad total.
  - Siempre que trabajemos en Internet, y no queramos incluir mecanismos de control de errores.
- ¿Cuándo usaremos un servidor sin conexión?
  - Cuando la aplicación requiera el uso de difusiones.
  - Cuando la sobrecarga computacional sea excesiva para los requerimientos de la aplicación (tiempo real).
  - Cuando el ámbito de la aplicación sea local (LANs: baja tasa de errores) y no se requiera alta fiabilidad.

Clases de servidores

- Existen 4 clases básicas de servidores:
  - **Secuencial con conexión** (TCP).
    - Ej.: DAYTIME, TIME, ECHO, FINGER, etc.
  - **Secuencial sin conexión** (UDP).
    - Ej.: DAYTIME, TIME, ECHO, FINGER, etc.
  - **Concurrente con conexión** (TCP).
    - Ej.: TELNET, WWW, FTP, NNTP, SMTP, etc.
  - **Concurrente sin conexión** (UDP).
    - Ej.: TFTP.

Algoritmo de un servidor secuencial UDP

Algoritmo de un servidor secuencial UDP

1. Crear un socket de tipo SOCK_DGRAM (SOCKET)

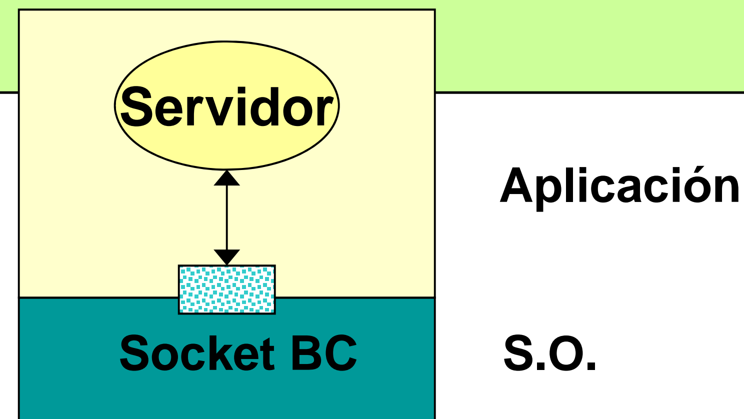
2. Asociar una dirección de socket bien-conocida (BIND).

3. Esperar la recepción de un mensaje de petición de servicio sobre el socket recién creado (RECVFROM).

4. Procesar la petición de servicio recibida y enviar los resultados al cliente que hizo la petición (un mensaje o varios). (SENDTO).

5.Volver al paso 3.

- **Estructura de un servidor secuencial UDP**



Servicio TIME

- Servicio TIME [RFC 868].
  - Análogo al servicio DAYTIME.
    - La fecha y hora se codifica en un entero de 32 bits.
      - Representa el número de segundos transcurridos desde las 00:00 del 1/1/1900 (***epoch*** *date*).
  - Ventajas respecto a DAYTIME:
      - El servicio es más rápido (sólo se envían 32 bits).
      - No hay que convertir la fecha y hora de un formato a otro.
      - No existen problemas de desplazamiento horarios (GMT).
  - El puerto bien-conocido para TIME es el 37.
  - El comportamiento del servidor TIME es análogo al del servicio DAYTIME.
  - Uso: Sincronización de relojes de hosts.

Servidor secuencial UDP de TIME

```
#define UNIXEPOCH   2208988800
int main (argc, argv)
int argc;
char *argv[];
{
struct sockaddr_in fsin;
char        *service = "time";
char buff[1];
int   sock, alen;
time_t      now;

switch (argc) {
  case 1:  break;
  case 2:  service = argv[1]; break;
  default:
      printf("usage:UDPtimed [port]\n");
      exit (1);
}

sock = passiveUDP (service);
```

```
while (1) {
  alen = sizeof(fsin);
  if (recvfrom (sock, buff, sizeof (buff), 0, (struct
      sockaddr *)&fsin, &alen) < 0)
    errexit("recvfrom: %s", sys_errlist[errno]);
  time (&now);
  now= htonl ((u_long) (now + UNIXEPOCH));
  (void) sendto (sock, (char *)&now, sizeof(now), 0,
      (struct sockaddr *)&fsin, sizeof(fsin));
  }  /* Fin del while */
}
```

```
int passiveUDP (service)
char  *service;
{
      return passivesock (service, "udp", 0);
}
```

Algoritmo de un servidor secuencial TCP

Algoritmo de un servidor secuencial TCP

1. Crear un socket de tipo SOCK_STREAM (SOCKET).

2. Asociar una dirección de socket bien-conocida (BIND).

3. Permitir la recepción de peticiones de conexión (LISTEN).

4. Esperarse activamente a recibir una petición de conexión (ACCEPT). Cuando se establece una nueva conexión se obtiene un nuevo socket, el cual está asociado a la conexión recién establecida.

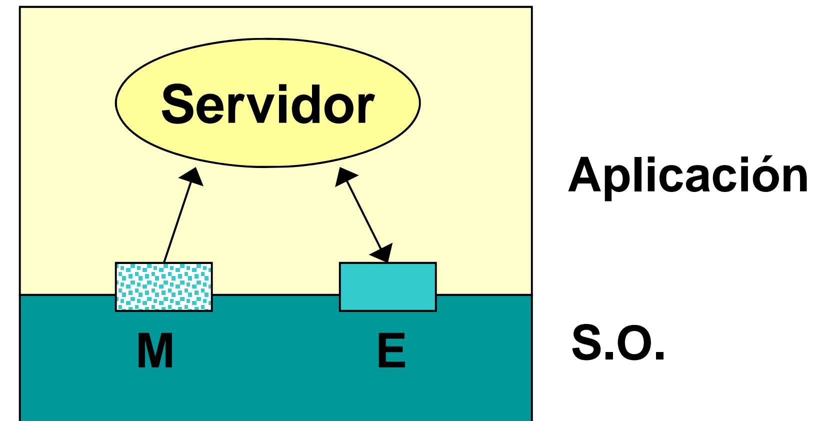
5. Diálogo cliente servidor basado en un protocolo de tipo petición-respuesta. Transferencia de información (READ, WRITE).

6. Al finalizar el servicio se debe cerrar el socket asociado a la conexión con el cliente (CLOSE).

7. Volver al paso 4.

Estructura de un servidor secuencial TCP

- Estructura.
- **Paso 4**: ACCEPT
  - Se dispone de un *socket* (Maestro) sobre el que se esperan las peticiones de los clientes.



  - Cuando un cliente establece conexión, se crea un nuevo socket (Esclavo) asociado a esa conexión
- **Paso 6**: CLOSE.
  - Cuando se llama a *close*, el TCP transfiere los datos aún no enviados antes de iniciar el protocolo de cierre.
  - Problema: Si el cierre lo inicia el cliente, éste podrá controlar los recursos (conexiones, puertos, sockets, etc.) que reserva el servidor.
    - ✧Ej.: Servidor FTP: Implementa un mecanismo para detectar y **eliminar conexiones sin actividad.**

Servidor secuencial TCP de DAYTIME

```
#define QLEN  5
int main (argc, argv)
int argc;
char *argv[];
{
struct sockaddr_in fsin;
char          *service = "daytime";
int   msock, ssock, alen;
time_t        now;
char          *pts;

switch (argc) {
case 1:  break;
case 2:  service = argv[1]; break;
default:
      printf("usage:TCPdaytimed [port]\n");
      exit(1);
}
msock = passiveTCP (service, QLEN);
```

```
while (1) {
  alen = sizeof(fsin);
  ssock = accept (msock, (struct sockaddr *) &fsin,
      &alen);
  if (ssock < 0) errexit ("accept failed: %s\n",
      sys_errlist [errno];

  time (&now);
  pts = ctime (&now);
  write (ssock, pts, strlen (pts));
  close (ssock);
 }  /* Fin del while */
}
```

```
int passiveTCP (service, qlen)
char  *service;
int qlen;
{
    return passivesock (service, "tcp", qlen);
}
```

Servidores concurrentes de UDP y TCP

- La mayoría de servidores concurrentes se basan en las capacidades multiproceso del S.O. donde residen.
- En los servidores concurrentes hablaremos de dos tipos de procesos: **maestro** y **esclavo**.
  - Normalmente existe un único proceso maestro que recoge las peticiones de servicio de los clientes y crea procesos esclavos que las atienden.
  - Un proceso maestro nunca habla directamente con el cliente.
  - Cuando un proceso esclavo finaliza el servicio con un cliente, desaparece del sistema.

Algoritmo del servidor concurrente UDP

Proceso Maestro

1. Crear un socket de tipo SOCK_DGRAM (SOCKET)

2. Asociar una dirección de socket bien-conocida (BIND).

3. Esperar la recepción de un mensaje de petición de servicio sobre el socket recién creado (RECVFROM).

4. Crear un proceso esclavo y pasarle la petición de servicio recibida.

5. Volver al paso 3.

Proceso Esclavo

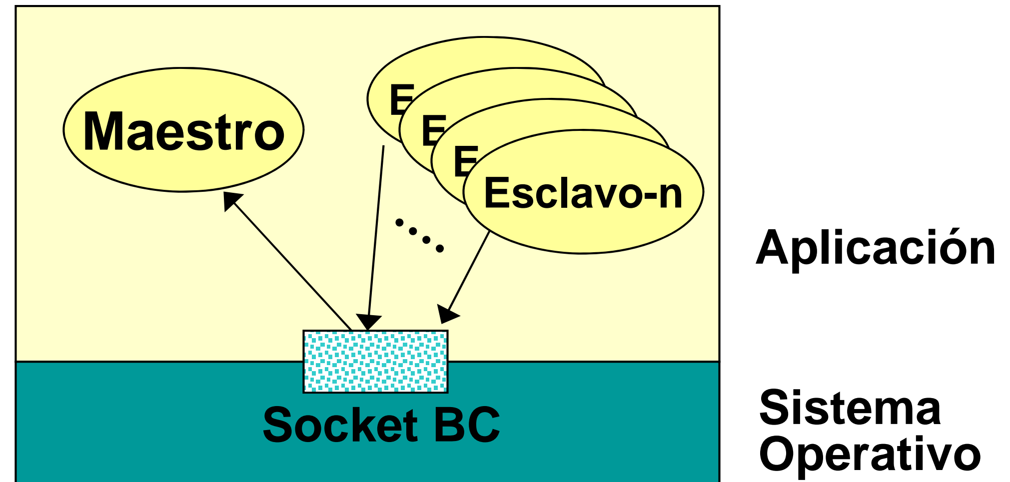
1. Recoge la petición de servicio que le pasa el proceso maestro así como la dirección del cliente.

2. Procesa la petición enviando los resultados al cliente en uno o varios mensajes. (SENDTO)

3. Finalizado el servicio, el proceso esclavo se autodestruye (EXIT).

Estructura del servidor concurrente UDP

- Estructura.



- Esta clase de servidores no son habituales.
  - Existen pocas aplicaciones en Internet que se ajusten a la estructura de esta clase de servidores. La mayoría de servidores concurrentes suelen ser con conexión (TCP).
  - El coste que supone crear un nuevo proceso a menudo no compensa el tiempo de servicio de una petición.
  - Una excepción:
    - TFTP (Trivial File Transfer Protocol).

Algoritmo del servidor concurrente TCP

Proceso Maestro

1. Crear un socket de tipo SOCK_STREAM (SOCKET)

2. Asociar una dirección de socket bien-conocida (BIND).

3. Permitir la recepción de peticiones de conexión (LISTEN).

4. Esperarse activamente a recibir una petición de conexión (ACCEPT).

5. Cuando se establece una nueva conexión, se crea un proceso esclavo (FORK) al que se le pasará el nuevo socket devuelto por ACCEPT.

6. Volver al paso 4.

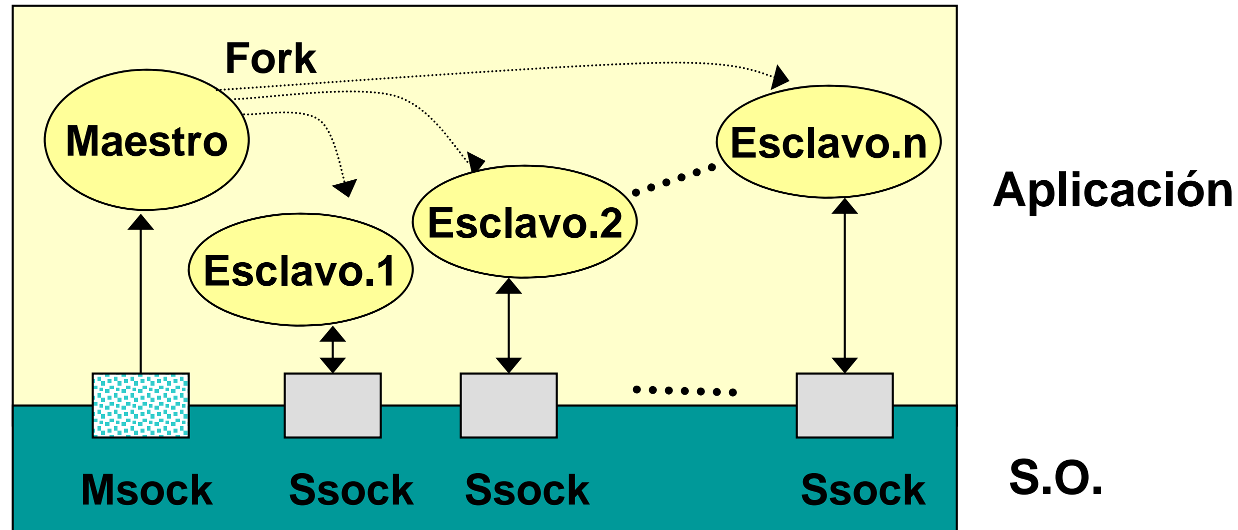
Proceso Esclavo

1. Recoge el nuevo socket resultante del establecimiento de conexión.

2. Intercambio de mensajes entre el cliente y el servidor. Protocolo de aplicación. (READ, WRITE)

3. Finalizado el servicio, el proceso esclavo cierra la conexión (CLOSE) y se autodestruye (EXIT).

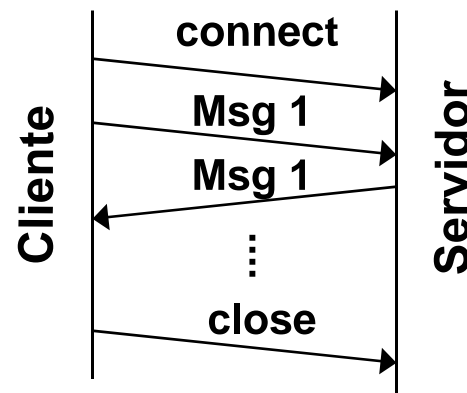
Estructura de un servidor concurrente TCP.



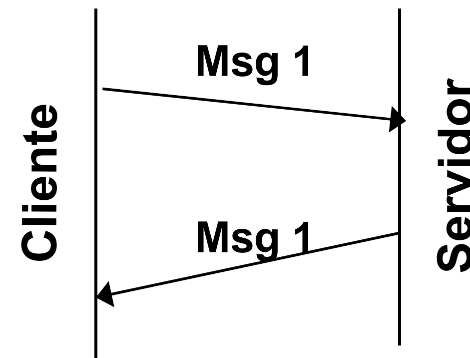
- La llamada al sistema ***fork()*** (Unix).
  - Con esta llamada un proceso puede crear otro proceso.
  - El proceso creado (proceso hijo) es una copia del proceso creador (proceso padre).
  - ***fork()*** devuelve un entero que permite distinguir ambos procesos.

Servicio de ECHO [RFC 862].

- Servicio estándar de Internet que se utiliza para depuración de programas de red y medidas de tiempos de respuesta.
  - Devuelve los mensajes que envían los clientes.
  - El puerto bien-conocido para ECHO es el 7.
  - Se implementa sobre TCP y UDP.



Versión TCP



Versión UDP

Servidor concurrente TCP de ECHO

```
char buff[LINELEN];
char *service;

int main (argc, argv)
int argc;
char *argv[];
{
  int msock, ssock, n;
  struct sockaddr_in cli_addr;
  int cli_size;

  switch (argc) {
    case 1: service = "servecho";
    break;
    case 2: service = argv[1];
    break;
    default:
    fprintf (stderr, "Uso: echod
    [port]\n");
    exit(1);
  }
msock = passiveTCP (service, 5);
  if (msock < 0)
    errexit ("Error passiveTCP:
    %s\n.", sys_errlist [errno]);
```

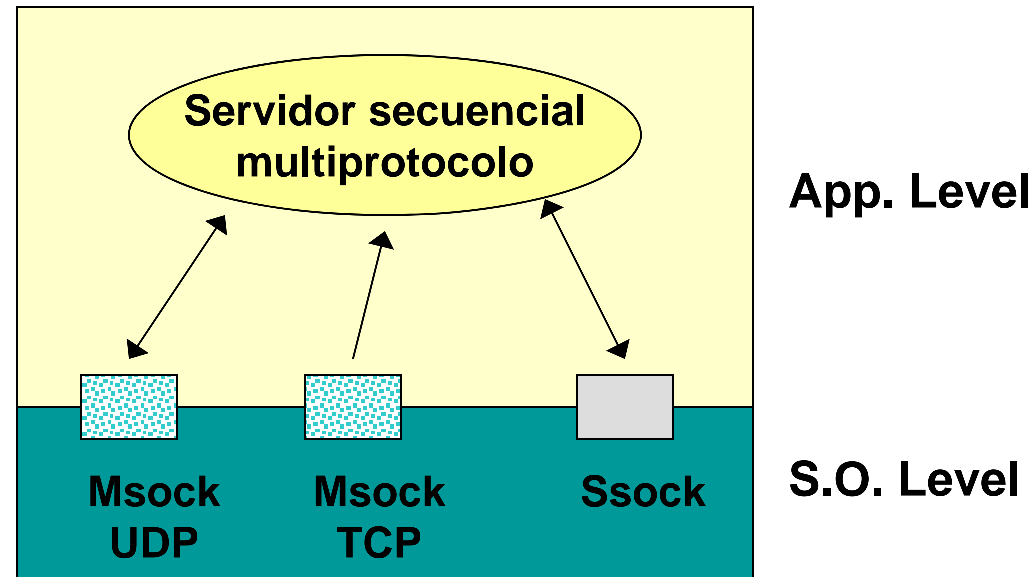
```
while (1) {
  cli_size = sizeof (cli_addr);
  ssock = accept (msock, &cli_addr,
    &cli_size);
  if (fork ())  close (ssock);
  else {    /* Proceso hijo */
    close (msock);
    do_echo (ssock);
    exit (0);
  }
 }  /* fin del while */
}
```

```
int do_echo(sock)
int sock;
{
  int       n;
  while ( (n = read (sock, buff,
    LINELEN)) > 0)
    write (sock, buff, n);
  close(sock);
}
```

Otros servidores

- Servidores MultiProtocolo.
  - Ofrecen un servicio (p.ej.: ECHO) sobre UDP y TCP al mismo tiempo.
  - Muchos servicios estándar de Internet se ofrecen tanto en UDP como en TCP.
  - Parece más lógico, implementar un único servidor que acepte peticiones de clientes TCP y UDP.
    - Menos procesos en el sistema.
    - Facilidad de mantenimiento.
    - El servicio es el mismo (reutilización de código).
  - Los servidores multiprotocolo puede ser secuenciales (más habitual) o concurrentes.

Estructura de un servidor multiprotocolo



- El servidor debe atender al mismo tiempo las peticiones que le lleguen sobre los *sockets* maestros de TCP y UDP.
- Para ello, se utiliza la llamada al sistema ***SELECT***.
  - ***SELECT*** analiza la actividad de los descriptores (*sockets*) que le indiquemos.

Servidor multiprotocolo DAYTIME

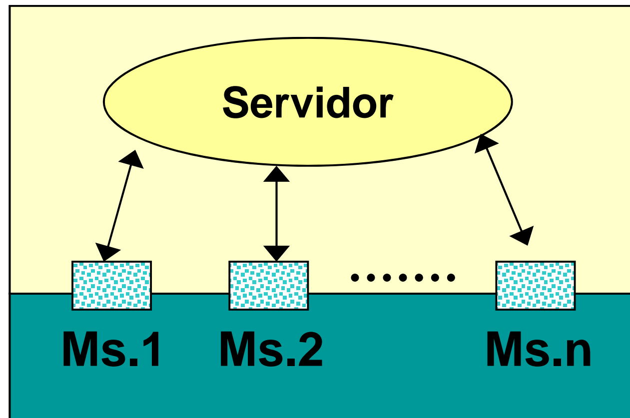
```
int main (int argc, char *argv[])
{
struct  sockaddr_in fsin;
char *service ="daytime", buff[128];
int       tsock, usock, ssock, alen,
    nfds;
fd_set    rfds;
switch (argc) {
  case 1:  break;
  case 2:  service = argv[1]; break;
  default:
  printf("usage: daytimed [port]\n");
   }
tsock = passivesock
    (service,"tcp",qlen);
usock = passivesock (service,"udp", 0);
nfds = MAX (tsock, usock) +1 ;
FD_ZERO (&rfds);
while (1) {
  FD_SET (tsock, &rfds);
  FD_SET (usock, &rfds);
```

```
if ( select (nfds, &rfds, (fd_set *)0, (fd_set
    *)0,
    (struct timeval *) 0) < 0)
    errexit ("select error: %s\n", sys_errlist
    [errno]);

if ( FD_ISSET (tsock, &rfds)) {
  alen = sizeof (fsin);
  ssock = accept (tsock, (struct sockaddr *)
    &fsin, &alen);
  if (ssock < 0)
    errexit ("accept: %s\n", sys_errlist [errno]);
  daytime (buff);
  write (ssock, buff, strlen(buff));
  close (ssock);
 }

if (FD_ISSET (usock, &rfds)) {
  alen = sizeof (fsin);
  if (recvfrom (usock, buff, sizeof(buff),0,
           (struct sockaddr *)&fsin, &alen) < 0)
    errexit ("recvfrom: %s\n",sys_errlist[errno]);
  daytime (buff);
  sendto (usock, buff, strlen (buff), 0, (struct
    sockaddr *) &fsin, sizeof (fsin));
 }
}  /* fin del while */
```

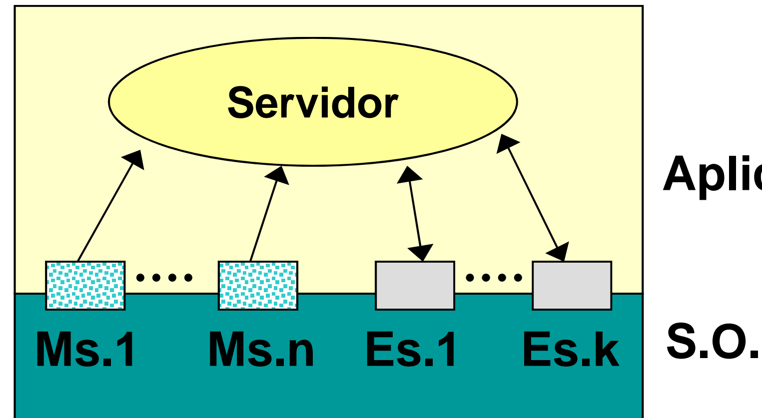
Servidores multiservicio

- Son servidores que suministran varios servicios simultáneamente.
- La motivación y ventajas que ofrecen este tipo de servidores son las mismas que las de los servidores multiprotocolo.
  - Hacen buen uso de los recursos del sistema.
  - Facilitan el mantenimiento.
- Pueden ser secuenciales, concurrentes, con conexión o sin conexión.

Estructura de algunos servidores multiservicio



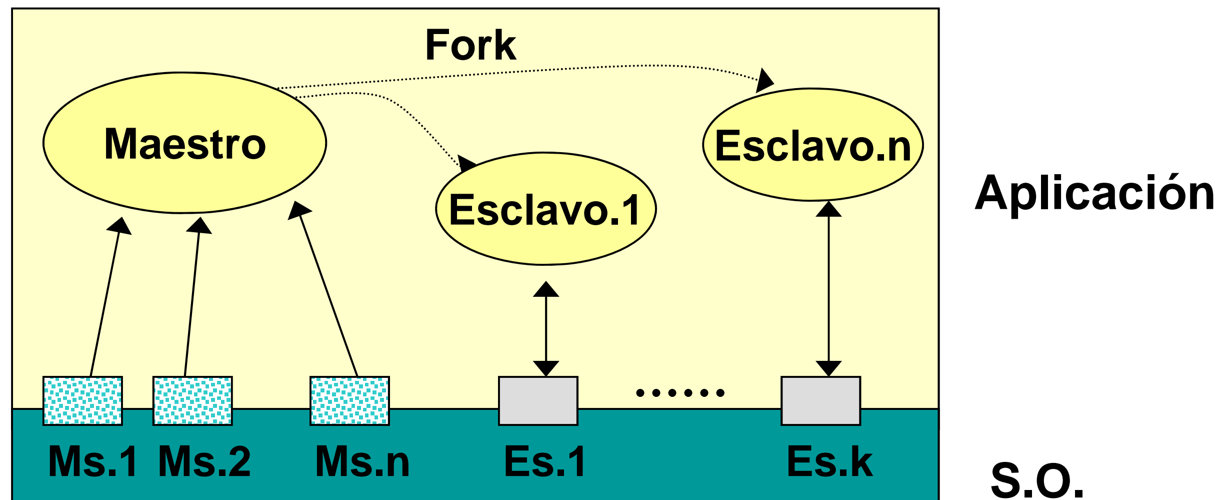
secuencial UDP



secuencial TCP

Aplicación

S.O.



Concurrente TCP

Aplicación

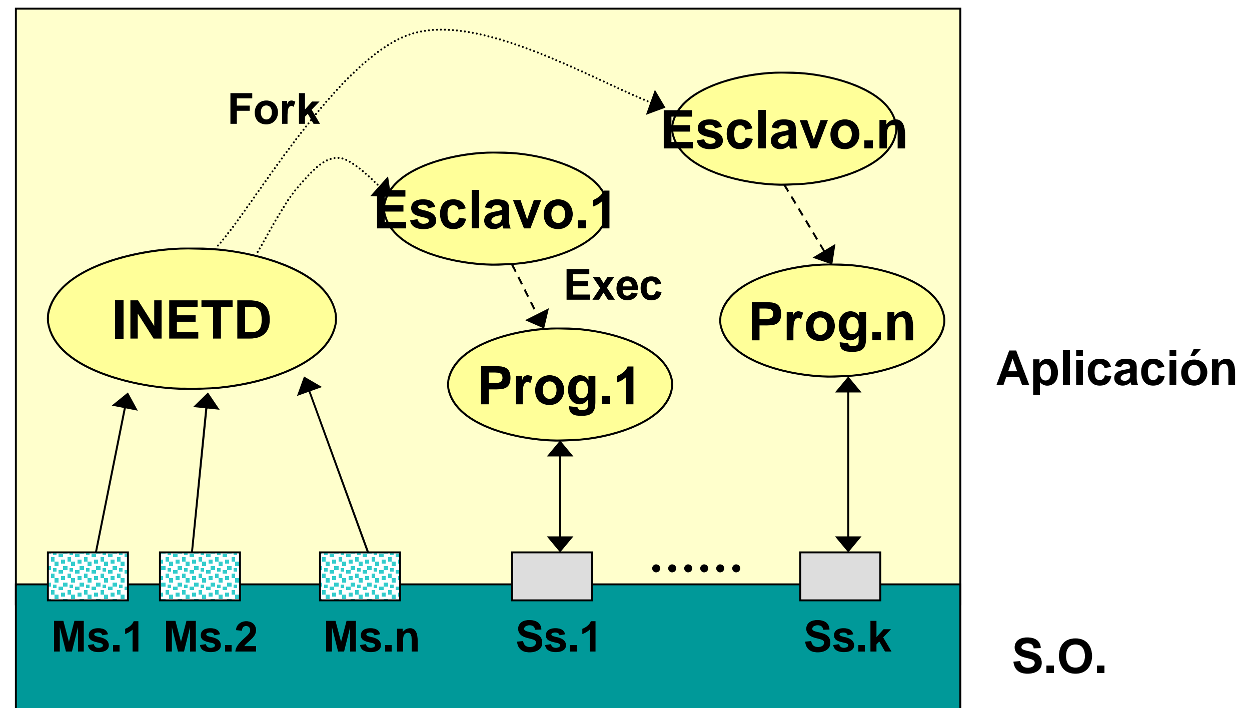
S.O.

El superservidor INETD

- BSD UNIX desarrolló un superservidor destinado a atender todas las peticiones de servicios Internet.
- Se clasifica como un servidor multiprotocolo y multiservicio que admite cualquier variante.
- Usa un fichero de configuración "/etc/inetd.conf" que almacena información sobre los servicios que debe atender y la forma en que debe hacerlo.
  - Fichero de texto. Cada línea define un servicio.

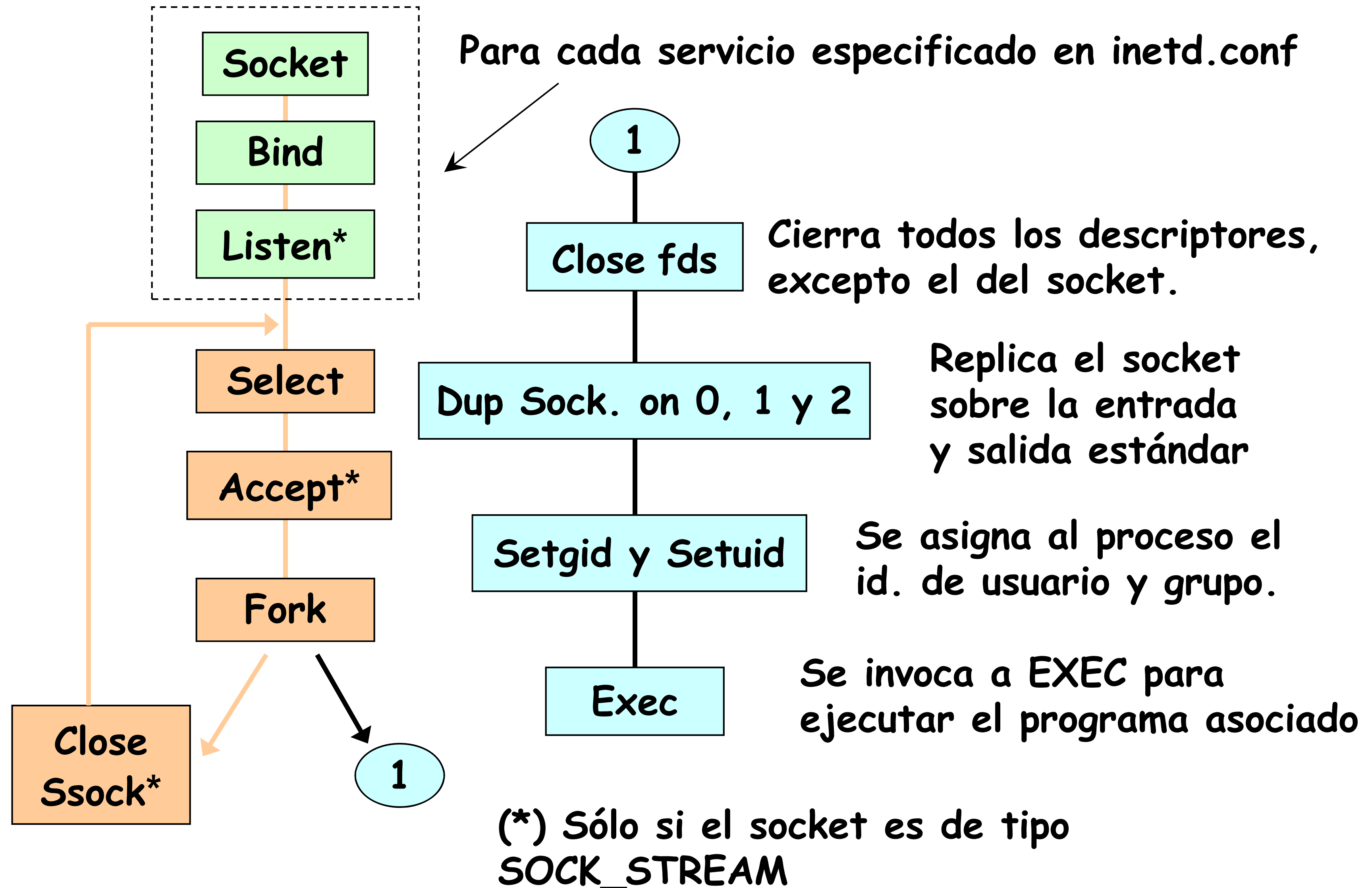
Service	Socket-T	Proto	Wait?	Name	Program Path	Prog. Argument
ftp	stream	tcp	nowait	root	/usr/sbin/ftpd	/usr/sbin/ftpd -l
telnet	stream	tcp	nowait	root	/usr/sbin/telnetd	/usr/sbin/telnetd

Estructura del servidor INETD



- Se encarga de recoger las peticiones de los distintos servicios que tiene configurados.
- Crea un proceso esclavo que a su vez invoca el programa que implementa el servicio demandado.

Diagrama de bloques de INETD



Consideraciones acerca de INETD

- Se lanza en el arranque del sistema.
  - De forma que sólo atenderá los servicios descritos en el fichero de configuración.
- Además implementa internamente servicios básicos de Internet (en versión multiprotocolo):
  - Echo [RFC 862], Discard [RFC 863], Chargen [RFC 864], Daytime [RFC 867] y Time [RFC 868].
- Todos los sistemas UNIX usan INETD para ofrecer servicios Internet.
- Su mantenimiento es muy sencillo, permitiendo actualizar versiones de servidores sin interrumpir al INETD.