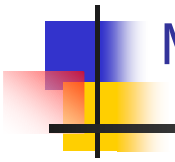
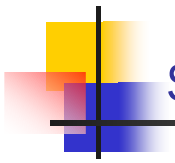


Mecanismos IPC: sockets

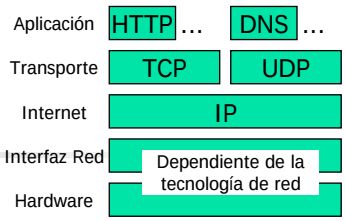
Ampliación de Sistemas Operativos (prácticas)
E.U. Informática en Segovia
Universidad de Valladolid



Sockets

- Los *sockets* son una abstracción lógica que permiten la comunicación bidireccional entre procesos que se ejecutan en una misma máquina o en máquinas distintas conectadas en red
- La comunicación mediante *sockets* es el mecanismo usado por la aplicaciones en red que se basan en tecnología Internet, es decir, en la familia de protocolos TCP/IP
  - De hecho, un *socket* no es más que la implementación del acceso a los servicios del nivel de transporte de la familia TCP/IP
  - En las aplicaciones en red es importante distinguir los recursos locales de los recursos remotos
- Desde el punto de vista de un proceso, un *socket* no es más que un recurso (al estilo de un fichero)
- Es fundamental que este *socket* esté identificado convenientemente para que pueda ser referenciado desde el exterior
  - La forma de identificar el *socket* está íntimamente relacionada con el mecanismo de direccionamiento de la familia de protocolos subyacente, es decir, la familia TCP/IP

Sockets :: identificación

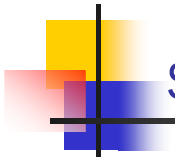


- Visión general familia protocolos TCP/IP
  - Sobre una capa dependiente de la tecnología de red usada, se encuentra el **protocolo de nivel de red IP** (*Internet Protocol*), responsable básicamente del direccionamiento de las estaciones y del enrutamiento de paquetes
    - Para nuestros propósitos, lo importante aquí es que, en la red, una máquina se distingue unívocamente de las demás mediante su dirección de nivel de red, llamada ***dirección IP***
  - Sobre el nivel de red, cubriendo funciones propias del nivel de transporte, encontramos dos **protocolos de transporte**: TCP (*Transmission Control Protocol*) y UDP (*User Datagram Protocol*)
    - TCP garantiza el transporte fiable de datos, mientras que UDP no, pero es más sencillo
    - Para nosotros, lo importante de estos protocolos de transporte es que incorporan un mecanismo de direccionamiento que permite diferenciar los *sockets* de una cierta máquina
    - Es decir, un *socket* que reside en una cierta máquina se distingue unívocamente de los demás *sockets* de la misma máquina, mediante su dirección de nivel de transporte, habitualmente denominada ***puerto***
      - Así, hablaremos de "puertos TCP" o "puertos UDP" dependiendo del protocolo utilizado
    - En definitiva, el puerto nos permite distinguir los procesos relacionados con la red dentro de una misma máquina
  - Como consecuencia de todo lo anterior, un *socket* se identifica indicando la máquina en la que reside y el puerto al que está asociado, es decir:

    socket: (dirección IP, puerto)

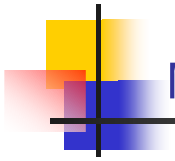
Sockets :: direcciones, puertos

- Las direcciones IP están formadas por 32 bits (4 bytes). Habitualmente se representan escribiendo el valor en decimal de cada byte separado por un punto
  - Por ejemplo, `157.88.25.4` es la dirección IP del servidor web de la UVa
- Existen direcciones IP especiales, por ejemplo, `127.0.0.1` o dirección local de la propia máquina
  - Toda máquina con capacidad de comunicaciones TCP/IP responde, al menos, a la dirección `127.0.0.1`, aunque ni siquiera esté conectada a la red
- Para facilitar el direccionamiento de una máquina, se desarrolló el servicio de nombres de dominio (DNS, *Domain Name Service*) que asocia la dirección IP de un *host* con un nombre de la forma *host*.dominio
  - Por ejemplo el nombre `www.uva.es` (*host* = `www`, dominio = `uva.es`) está asociado a la dirección `157.88.25.4`
  - Otro ejemplo, el nombre `localhost` está asociado a la dirección `127.0.0.1`
- Los puertos son valores de 16 bits, y por tanto cubren el rango 0..65535. Algunos de estos puertos están asignados para las aplicaciones estándar de Internet
  - El sistema operativo reserva un conjunto de puertos para su uso (hasta el 1024)
    - Por ejemplo, un servidor web atiende por defecto el puerto 80. Es decir, el servidor web de la UVa atiende el puerto 80 de la máquina cuya dirección IP es 157.88.25.4
  - El resto de los puertos (por encima del 1024) pueden ser usados libremente por las aplicaciones



Sockets :: tipos de sockets

- Como consecuencia de tener dos protocolos de transporte, TCP y UDP, existen también dos tipos de sockets:
  - *sockets* TCP, "*stream*" u "orientados a la conexión"
    - La comunicación entre *sockets* TCP es fiable (sin pérdida de paquetes), ordenada (TCP nos proporciona los paquetes en la secuencia correcta) y sin duplicación de paquetes. TCP, y consecuentemente, este tipo de *sockets*, es orientado a la conexión
    - Esto quiere decir que los *sockets* de los extremos a comunicar deben establecer un circuito virtual a través del cual se producirá el intercambio de información
    - El establecimiento del circuito virtual se suele denominar conexión, y es el paso previo al intercambio de datos
    - Una vez finalizado éste, se procede a la desconexión
  - *sockets* UDP, "*datagram*" o "no orientados a la conexión"
    - La comunicación entre *sockets* UDP no garantiza la llegada de todos los paquetes, ni su orden, ni su unicidad
    - La aplicación que use *sockets* de este tipo tendrá que encargarse de resolver estos problemas potenciales
    - Para la comunicación entre *sockets* UDP no se establece un circuito virtual, sino que en su lugar se realiza un intercambio de datagramas, es decir, de paquetes sueltos



Modelo Cliente-Servidor

- Modelo estándar para la ejecución de aplicaciones en red y sistemas operativos distribuidos
- Servidor
  - proceso que se está ejecutando en un nodo de la red y que gestiona el acceso a un determinado recurso
- Cliente
  - proceso que se ejecuta en el mismo o diferente nodo y que realiza peticiones de servicio a un servidor
- De acuerdo con la forma de prestar el servicio, se pueden considerar 2 tipos de servidores
  - Servidores Interactivos
    - El servidor recoge la petición de servicio y además se encarga el mismo de atenderla
  - Servidores Concurrentes
    - El servidor recoge cada una de las peticiones de servicio y crea otros procesos que se encargan de atenderlas
- Ejemplos de servidores
  - devolver la hora al cliente, imprimir un fichero para el cliente, leer o escribir para el cliente en un fichero, etc.

Modelo Cliente-Servidor :: servidores

- Servidor interactivo

```
int sockfd, newsockfd;

if ( (sockfd = socket (... )) < 0 )
   err_sys("socket error");
if ( bind(sockfd, ...) < 0 )
   err_sys("bind error");
if ( listen(sockfd, 5) < 0)
   err_sys("listen error");

for (;;) {
   newsocketfd= accept(socketfd,...);
   if (newsocketfd < 0)
        err_sys("accept error");

   /* procesar la petición */
   doservice(newsockfd);
   close(newsocketfd);
}
```

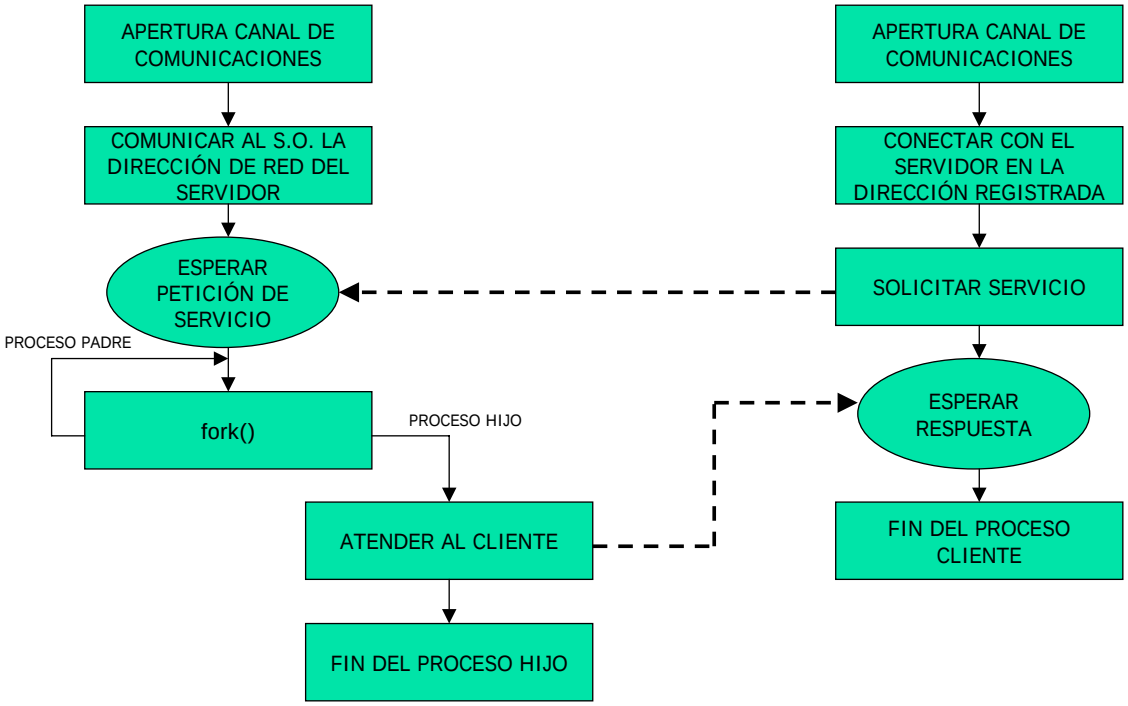
- Servidor concurrente

```
int sockfd, newsockfd;

if ( (sockfd = socket (... )) < 0 )
    err_sys("socket error");
if ( bind(sockfd, ...) < 0 )
    err_sys("bind error");
if ( listen(sockfd, 5) < 0)
    err_sys("listen error");

for (;;) {
    newsocketfd= accept(socketfd,...);
    if (newsocketfd < 0)
          err_sys("accept error");

    if ( (fork()==0) {
          close(sockfd);          /* HIJO */
          /* procesar la petición */
          doservice(newsockfd);
          exit(0);
    }
    close(newsocketfd);           /*PADRE*/
}
```

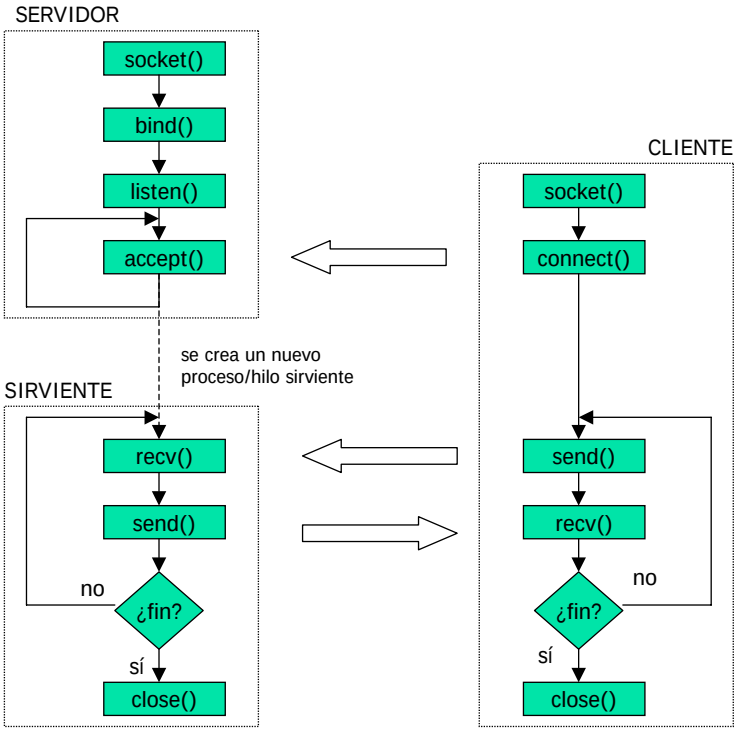

Modelo Cliente-Servidor :: escenario normal

- Servidor
  - Abre el canal de comunicaciones e informa a la red tanto de la dirección a la que responderá, como de su disposición para aceptar peticiones de servicio
  - Espera a que un cliente le pida servicio en la dirección que él tiene declarada
  - Cuando recibe una petición de servicio
    - si es un servidor interactivo, atenderá al cliente
    - si es un servidor concurrente, creará un proceso hijo o lanzará un hilo para que le de servicio al cliente
  - Continuará en espera de nuevas peticiones de servicio
- Cliente
  - Abre el canal de comunicaciones y se conecta a la dirección de red atendida por el servidor. Esta dirección debe ser conocida por el cliente
  - Envía al servidor un mensaje de petición de servicio y espera hasta recibir la respuesta
  - Cierra el canal de comunicaciones y termina la ejecución

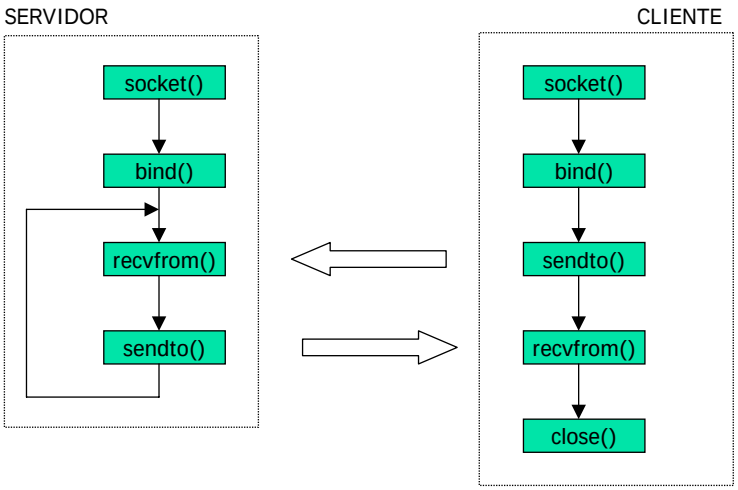
Modelo Cliente-Servidor :: Diagrama de Flujo



Servicio Orientado a Conexión

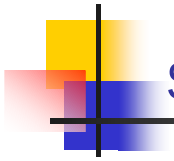


Servicio No Orientado a Conexión



Sockets POSIX

- Los *sockets* en UNIX/LINUX pueden distinguirse por su dominio de comunicación, esto es, atendiendo a dónde residen los procesos que se van a comunicar a través de ellos:
  - Dominio UNIX (AF_UNIX): ambos procesos residen en la misma máquina
  - Dominio Internet (AF_INET): los procesos residen en máquinas posiblemente distintas conectadas mediante una red tipo TCP/IP
- Realmente, el dominio AF_UNIX no es necesario, pues con *sockets* de dominio AF_INET se pueden comunicar dos procesos locales
  - Sin embargo, si restringimos el dominio a AF_UNIX, la comunicación será más eficiente al emplear menos recursos.
- Por su parte, los *sockets* AF_INET admiten la clasificación que ya conocida, en función del protocolo de transporte usado:
  - sockets TCP (SOCK_STREAM)
  - sockets UDP (SOCK_DGRAM)

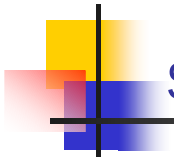


Sockets POSIX :: Byte Order (1)

- El flujo de datos entre sockets TCP está orientado al intercambio de bytes (char en lenguaje C), pero ¿en qué orden deben enviarse los bytes individuales que componen cada uno de los datos que ocupan más de un byte: short o long?
  - Sistemas basados en microprocesadores distintos usan diferentes convenciones a la hora de almacenar internamente estos tipos de datos, como es el caso de los Intel x86 (primero el byte menos significativo, convención *little endian*) y Motorola 680xx (primero el byte más significativo, convención *big endian*)
- Para solucionar el problema, se distinguen dos tipos de ordenamiento: el ***host order*** (el orden con el que se almacena en el equipo, que será *little* o *big endian*, dependiendo del sistema) y el ***network order*** (el orden establecido por convención para las redes tipo Internet, concretamente *big endian*), y se establecen las siguientes reglas:
  - todos los datos que se envíen hacia la red deben convertirse al *network order*
  - todos los datos que se reciban de la red deben convertirse al *host order*

Sockets POSIX :: Byte Order (2)

- De esta forma, si el extremo A quiere transmitir un dato (de longitud mayor a un byte) al extremo B,
  - lo primero que hace es convertir el dato desde el *host order* de A al *network order*
  - una vez recibido el dato en B en *network order*, se transforma al *host order* de B
  - De esta forma, independientemente de cuáles sean los *host order* de A y B, la comunicación se realiza correctamente
    - La clave está en que se ha definido un *network order* universal, que siempre se respeta.
- El problema de orden de envío es relevante sólo cuando la unidad de información tiene una longitud superior a un byte.
- Las funciones `short htons (short)` y `long htonl(long)` realizan la conversión de *host order* a *network order* de un entero tipo `short` o `long`, respectivamente
- Mientras que short `ntohs(short)` y `long ntohl(long)` realizan las conversiones opuestas, es decir, de *network order* a *host order*
  - Estas funciones están definidas en el fichero de cabecera `<netinet/in.h>`



Sockets POSIX :: tipos de datos

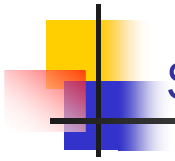
- **`struct in_addr`**
  - **Estructura que representa una dirección IP**
  - `unsigned long s_addr`: entero de 32 bits que almacena la dirección IP
    - Debe estar en *network order*
    - La constante `INADDR_ANY` puede usarse para especificar la dirección local
- **`struct sockaddr_in`**
  - **Estructura que representa la dirección de un *socket* de dominio Internet**
  - `short int sin_family`: dominio del *socket*, que, en este tipo de datos, debe especificar el dominio Internet, mediante la constante `AF_INET`
  - `unsigned short sin_port`: número de puerto asociado al *socket*
    - Debe estar en *network order*
  - `struct in_addr sin_addr`: dirección IP de la máquina correspondiente al *socket*
  - `unsigned char sin_zero[8]`: debe estar inicializado a 0s (por ejemplo, usando `memset()` o `bzero()`).

Sockets POSIX :: funciones

- **`socket()`**: Función que crea o "abre" un socket

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

- Parámetros y valor de retorno:
  - `int dominio`: dominio del *socket* a crear (`PF_UNIX`, `PF_INET`, `PF_INET6`, ...)
  - `int tipo`: tipo de *socket*: `SOCK_STREAM`, `SOCK_DGRAM`, ...
  - `int protocolo`: protocolo asociado al *socket*
    - Se recomienda el valor 0 para que la función escoja automáticamente el protocolo más adecuado según el tipo de *socket* especificado
  - Devuelve (`int`): descriptor del nuevo *socket*
    - El valor -1 indica que no se pudo crear el *socket*.

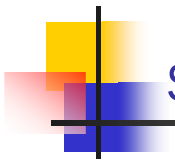


Sockets POSIX :: funciones

- **`bind()`**: Asigna una dirección (dirección IP y puerto) al *socket* especificado. Esto hace que sea posible referenciar al *socket* desde un proceso remoto

```
int bind(int sockfd, struct sockaddr *addr, socklen_t addrlen);
```

- Parámetros y valor de retorno:
  - `int sockfd`: descriptor del *socket* al que se le va a asignar una dirección
  - `struct sockaddr *addr`: dirección a asignar al socket
    - Con *sockets* TCP/IP, `addr` debe ser una variable de tipo `struct sockaddr_in`, por lo que es necesario realizar un *cast*, de la forma `(struct sockaddr *) addr`, al hacer la llamada a `bind()`
    - La función bind() no es exclusiva de sockets TCP, lo cual explica que este parámetro sea de tipo struct sockaddr, más general que struct sockaddr_in
  - `int addrlen`: tamaño de la estructura `struct sockaddr`, calculado con el operador `sizeof()`
  - Devuelve (`int`): 0 si la operación tuvo éxito; -1 indica un error.



Sockets POSIX :: funciones

- **`listen()`**: Prepara un *socket* para ser utilizado en modo servidor, asignándole una cola de conexiones pendientes de procesar

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

- Parámetros y valor de retorno:
  - `int sockfd`: descriptor del *socket* sobre el que se va a actuar
  - `int backlog`: tamaño máximo de la cola de conexión
  - Devuelve (`int`): 0 si la operación tuvo éxito; -1 indica un error

Sockets POSIX :: funciones

- **`accept()`**: Pone a un *socket* servidor en estado de escucha, monitorizando la cola de solicitudes de conexión
  - Cuando recibe una solicitud desde un cliente, y ésta es aceptada, crea un *socket* (con el mismo número de puerto que el *socket* servidor) que es conectado al *socket* cliente
  - La llamada a la función `accept()` es bloqueante

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

- Parámetros y valor de retorno:
  - `int sockfd`: descriptor del *socket* sobre el que se va a actuar
  - `struct sockaddr *addr` (parám. salida): dirección del *socket* cliente que ha solicitado la conexión
    - Para *sockets* TCP/IP, `addr` debe ser una variable de tipo struct sockaddr_in, por lo que habrá que hacer un *cast* al llamar a esta función
  - `int *addrlen` (parám. salida): tamaño de la estructura devuelta en `addr`
  - Devuelve (`int`): descriptor del *socket* creado
    - Este *socket* sólo es válido para la conexión recién iniciada
    - El valor -1 indica que no se pudo crear el *socket*

Sockets POSIX :: funciones

- **`connect()`**: Trata de establecer una conexión entre el *socket* local (cliente) `sockfd` y el *socket* remoto (servidor) identificado por `addr`

```
int connect(int sockfd, struct sockaddr *addr, int
  addrlen);
```

- Parámetros y valor de retorno:
  - `int socketfd`: descriptor del *socket* local (cliente)
  - `struct sockaddr *addr`: dirección del *socket* servidor con el que se desea conectar. Para sockets TCP/IP, `addr` debe ser una variable de tipo `struct sockaddr_in`, por lo que habrá que hacer un *cast* al llamar a esta función
  - `int addrlen`: tamaño de la estructura dada en *addr*
  - Devuelve (`int`): 0 si la operación tuvo éxito; -1 indica un error.

Sockets POSIX :: funciones

- **`send()`**: Envía una secuencia de bytes a través del *socket* `sockfd`

```
int send(int sockfd, const char *msg, int len, int
  flags);
```

- Parámetros y valor de retorno:
  - `int sockfd`: descriptor del *socket* a través del cual se quieren enviar datos
  - `const char *msg`: secuencia de bytes (`char`) a enviar
  - `int len`: número de bytes a enviar de `msg`
  - `int flags`: en nuestro caso, no se utilizarán `flags`, de forma que este parámetro debe ser 0
  - Devuelve (`int`): el número de bytes realmente enviados; -1 indica un error

Sockets POSIX :: funciones

- **`sendto()`**: Envía una secuencia de bytes a través del *socket* `sockfd`
  - No se requiere un socket orientado a conexión

```
int  sendto(int sockfd, const void *msg, size_t len, int flags,
  const struct sockaddr *to, socklen_t tolen);
```

- Parámetros y valor de retorno:
  - `int sockfd`: descriptor del *socket* a través del cual se quieren enviar datos
  - `const char *msg`: secuencia de bytes (`char`) a enviar
  - `int len`: número de bytes a enviar de `msg`
  - `int flags`: en nuestro caso, no se utilizarán `flags`, de forma que este parámetro debe ser 0
  - `struct sockaddr *to`: dirección del *socket* destinatario al que se desea enviar los datos
  - `int tolen`: tamaño de la estructura dada en `to`
  - Devuelve (`int`): el número de bytes realmente enviados; -1 indica un error

Sockets POSIX :: funciones

- **`recv()`**: Recibe una secuencia de bytes a través del *socket* `sockfd`
  - La función posee bloqueo, esto es, no se devuelve el control hasta que se recibe algo

```
int recv(int sockfd, char *buf, int len, int flags);
```

- Parámetros y valor de retorno:
  - `int sockfd`: descriptor del *socket* a través del cual se quieren recibir datos
  - `char *buf` (parám. de salida): *buffer* en el que se almacenan los datos recibidos
  - `int len`: tamaño del *buffer* `buf`
  - `int flags`: en nuestro caso, no se utilizarán flags, de forma que este parámetro debe ser 0
  - Devuelve (`int`): el número de bytes recibidos; -1 indica un error

Sockets POSIX :: funciones

- **`recvfrom()`**: Recibe una secuencia de bytes a través del *socket* `sockfd`
  - La invocación a la función recvfrom() es bloqueante
  - No se requiere que sea un socket orientado a conexión

```
int  recvfrom(int  sockfd, void *buf, size_t lon, int flags,
   struct sockaddr *from, socklen_t *lonfrom);
```

- Parámetros y valor de retorno:
  - `int sockfd`: descriptor del *socket* a través del cual se quieren recibir datos
  - `char *buf` (parám. de salida): *buffer* en el que se almacenan los datos recibidos
  - `int len`: tamaño del *buffer* `buf`
  - `int flags`: en nuestro caso, no se utilizarán flags, de forma que este parámetro debe ser 0
  - `struct sockaddr *from` (páram. de salida): dirección del *socket* originario de los datos recibidos
  - `int *fromlen` (parám. de salida): tamaño de la estructura dada en `from`
  - Devuelve (`int`): el número de bytes recibidos; -1 indica un error

Sockets POSIX :: funciones

- **`close()`**: Cierra el socket

```
int close(int sockfd);
```

- Parámetros y valor de retorno:
  - `int sockfd`: descriptor del *socket* a cerrar
  - Devuelve (`int`): 0 en caso de éxito; -1 indica un error

Sockets POSIX :: comentarios finales

- Para poder utilizar estas funciones hay que incluir los ficheros de cabecera `<sys/types.h>` y `<sys/socket.h>`
- Es destacable que muchas de estas funciones utilizan la convención de devolver el valor -1 ante cualquier posible error
  - Cuando esto se produce, el sistema modifica el valor de una variable global, denominada `errno`, asignándole un valor que indica el tipo de error que se ha producido
  - La función `perror()` emite un mensaje de error en función del valor de `errno`
  - Estas características están disponibles al incluir el fichero de cabecera`<stdio.h>`. Para más información, consultar las páginas de manual `man`
- Otras funciones interesantes, aunque no fundamentales, relacionadas con sockets son:
  - `inet_addr()` e `inet_ntoa()`, para el tratamiento de direcciones IP
  - `gethostname()`, `gethostbyname()` y `getpeername()` para el uso del servicio de nombres de dominio (DNS), traduciendo direcciones IP en nombres/dominios de hosts y viceversa
  - `getprotoent()` y `getprotobyname()` para la selección de protocolos a la hora de crear sockets
  - `select()`, para la monitorización de un conjunto de sockets
  - `fcntl()`, para modificar las propiedades de un socket (como por ejemplo, hacer no bloqueantes las llamadas a `accept()` o `recv()`), etc.
  - Para más información sobre ellas, consultar las páginas de manual `man`