UNIVERSIDADE DA CORUÑA
Departamento de Tecnoloxías da Información
e as Comunicacións

APÉNDICE:

MANUAL DE SOCKETS EN C

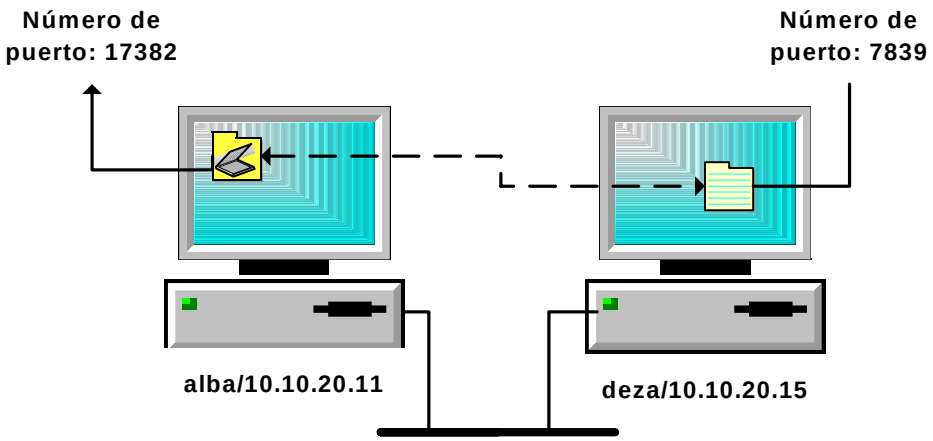
ÍNDICE

1. Introducción

Los sockets constituyen una interfaz de entrada-salida que permite la comunicación entre diferentes procesos. Esto no es un aspecto novedoso si no se tiene en cuenta que los procesos pueden estar ejecutándose en la máquina o en distintos sistemas, unidos mediante una red.

Por lo tanto, los sockets permiten la comunicación entre varios procesos que pueden estar ejecutándose en diferentes máquinas.

Los sockets para TCP/IP permiten la comunicación de dos procesos que estén conectados a través de una red TCP/IP. En una red de este tipo, cada máquina está identificada por medio de su dirección IP (tal como se observa en la siguiente figura), aunque por comodidad se les asignan nombres a cada una de las máquinas. En cada máquina se están ejecutando múltiples procesos simultáneamente, algunos de los cuales se estarán comunicando a través de la red. Cada uno de estos procesos se asocia con un número de puerto, para poder así diferenciar los distintos paquetes que reciba la máquina.



Un socket se identifica unívocamente por: **dirección IP + número de puerto**.
Una comunicación entre sockets se identifica unívocamente mediante: **identificador de socket origen + identificador de socket destino**.

La mayoría de los lenguajes de programación actuales disponen de una interfaz adecuada para la programación de sockets. En concreto, en este

manual se ofrece una introducción al manejo de sockets en el lenguaje de programación C.

2. Sockets en C

2.1. Conceptos básicos

2.1.1. Dominios de comunicación

Los sockets se crean dentro de lo que se denomina un dominio de comunicación, al igual que un archivo se crea dentro de un sistema de ficheros concreto. El dominio de comunicación permite definir en donde se encuentran los procesos que se van comunicar.
Los dominios que se definen en el lenguaje C son los siguientes:

- AF_UNIX: representa el dominio característico de los procesos que se comunican en un mismo sistema UNIX.
- AF_INET: es el dominio que utilizan los procesos que se comunican a través de cualquier red TCP/IP.
- Otros dominios: en función del sistema a emplear se definen otros tipos de dominios.

Destacar que en esta introducción únicamente se hará referencia a sockets creados bajo el dominio AF_INET.

2.1.2. Tipos de sockets

En el dominio AF_INET se definen los siguientes tipos de sockets:

- Sockets Stream
- Sockets Datagram
- Sockets Raw

El tipo de sockets Stream hace uso del protocolo TCP (protocolo de la capa de transporte) que provee un flujo de datos bidireccional, orientado a conexión, secuenciado, sin duplicación de paquetes y libre de errores.

El tipo de sockets Datagram hacen uso del protocolo UDP (protocolo de la capa de transporte), el cual provee un flujo de datos bidireccional, no orientado a conexión, en el cual los paquetes pueden llegar fuera de secuencia, puede haber pérdidas de paquetes o pueden llegar con errores.

El tipo de sockets Raw permiten un acceso a más bajo nivel, pudiendo acceder directamente al protocolo IP del nivel de Red. Su uso está mucho más limitado ya que está pensado principalmente para desarrollar nuevos protocolos de comunicación, o para obviar los protocolos del nivel de transporte.

2.1.3. Ordenación de los bytes

La forma en la que se almacena los bytes en memoria puede variar de un sistema a otro, en incluso, puede variar del sistema a la red.
Existen dos formas de almacenar un byte en memoria:

- En la posición más alta se almacena el bit más significativo.
- En la posición más alta se almacena el bit menos significativo.

Para evitar problemas a la hora de enviar y recibir datos a través de la red es necesario convertir los datos a enviar al formato de la red al enviar, y al formato de la máquina al recibir datos. Para esto existen las siguientes funciones de conversión:

- *htons()* - host to network short: convierte un short int de formato de máquina al formato de red.
- *htonl()* - host to network long: convierte un long int de formato de máquina al formato de red.
- *ntohs()* - network to host short: convierte un short int de formato de red al formato de máquina.
- *ntohl()* - network to host long: convierte un long int de formato de red al formato de máquina.

Puede ser que el sistema donde se este programando almacene los datos en el mismo formato que la red, por lo que no sería necesario realizar ninguna conversión, pero al tratar de compilar y ejecutar el mismo código en un sistema con diferente ordenación el resultado es impredecible.
Ejemplo:

```
#include <netinet/in.h>
...
port = htons ( 3490 );
...
```

2.2. Creación de un socket

Los sockets se crean llamando a la función *socket()*, que devuelve el identificador de socket que es de tipo entero (es equivalente al concepto de identificador de fichero).
En caso de que se haya producido algún error durante la creación del socket, la función devuelve -1 y la variable global errno se establece con un valor que indica el error que se ha producido. La función perror("...") muestra por pantalla un mensaje explicativo sobre el error que ha ocurrido.

El formato de la función socket() es el siguiente:

sockfd = socket (int dominio, int tipo, int protocolo);
sockfd: Identificador de socket. Que luego se utilizara para conectarse, recibir conexiones, enviar y recibir datos, etc.
dominio: Dominio donde se realiza la conexión. En este caso, el dominio será siempre AF_INET.
tipo: Se corresponde con el tipo de socket que se va a crear, y puede tomar los siguientes valores (definidos como constantes en las librerías): SOCK_STREAM, SOCK_DGRAM o SOCK_RAW.
protocolo: Indica el protocolo que se va a utilizar. El valor 0 indica que seleccione el protocolo más apropiado (TCP para SOCK_STREAM, UDP para SOCK_DGRAM).

A continuación se muestra un ejemplo de utilización:

```
#include <sys/types.h>
#include <sys/socket.h>
....
int sockfd;
sockfd =socket ( AF_INET, SOCK_STREAM, 0 );
...
```

2.3. Función bind()

La función socket() únicamente crea un socket, pero no le asigna un nombre. La función *bind()* se utiliza para darle un nombre al socket, esto es una dirección IP y numero de puerto de la máquina local a través de donde se enviarán y se recibirán datos.

El formato de la función es el siguiente:

int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
sockfd: Identificador de socket devuelto por la función socket().
my_addr: Es un puntero a una estructura sockaddr que contiene la IP del host local y el numero de puerto que se va a asignar al socket (esta estructura se detalla en la siguiente sección).
addrlen: debe estar establecido al tamaño de la estructura anterior, utilizando para ello la función sizeof().

Ejemplo :

```
...
struct sockaddr_in sin;
...
bind ( sockfd, (struct sockaddr *) &sin, sizeof (sin) );
```

La función bind() (al igual que la función socket(), y en general todas las funciones relacionadas con sockets) devuelve –1 en caso de que se haya producido alguna situación de error, y establece la variable global errno al número de error producido, por lo que se puede invocar directamente a la función perror() para mostrarlo por pantalla.

2.4. Estructuras de datos

Existen diversas estructuras asociadas con los sockets. A continuación se detallan las más importantes

```
struct sockaddr
{
unsigned short          sa_family;      // AF_*
```

```
char                    sa_data[14];    // Dirección de protocolo.
};

struct sockaddr_in
{
short int               sin_family;     // AF_INET
unsigned short          sin_port;       // Numero de puerto.
struct in_addr          sin_addr;       // Dirección IP.
unsigned char           sin_zero[8];    // Relleno.
};

struct in_addr
{
unsigned long           s_addr;         // 4 bytes.
};
```

La estructura más interesante para TCP/IP es la sockaddr_in, equivalente a la estructura sockaddr, pero que permite referenciar a sus elementos de forma más fácil. Es importante destacar que todas las funciones de sockets esperan recibir como parámetro un puntero a una estructura sockaddr, por lo que es necesario realizar una conversión de tipos (cast) a este tipo, tal y como se ha realizado en el ejemplo anterior.
Los campos de la estructura sockaddr_in son los siguientes:
sin_family: tomará siempre el valor AF_INET.
sin_port: representa el número de puerto, y debe estar en la ordenación de bytes de la red.
sin_addr: este campo representa la dirección IP y se almacena en un formato específico, que se detalla a continuación.
sin_zero: se utiliza simplemente de relleno para completar la longitud de sockaddr.

Para convertir una dirección IP en formato texto (por ejemplo, “193.144.57.67”) a un unsigned long con la ordenación de bytes adecuada se utiliza la función inet_addr(). Esta función convierte únicamente direcciones IP

a formato numérico, **NO** convierte nombres de máquinas. En caso de error devuelve –1 y activa la variable global errno.

Ejemplo:

```
....
struct sockaddr_in sin;
....
sin.sin_addr.s_addr=inet_addr("193.144.57.67");
```

La función inversa se denomina inet_ntoa(), que convierte una dirección en formato numérico en una cadena de caracteres.

Ejemplo:

```
....
printf("%s", inet_ntoa(sin.sin_addr));
....
```

2.4.1. Asignación de valores

Previamente a la llamada a la función bind() es necesario asignar valores a una variable de tipo sockaddr_in para indicar el nombre del socket.

Ejemplo:

```
...
struct sockaddr_in sin;
...
sin.sin_family = AF_INET;
sin.sin port = htons ( 1234 );     // Numero de puerto en donde recibirá paquetes el programa
sin.sin_addr.s_addr = inet_addr ("132.241.5.10"); // IP por donde recibirá paquetes el programa
...
```

Existen algunos casos especiales a la hora de asignar valores a ciertos campos.
En caso de asignar el valor cero a sin_port, el sistema fijará el número de puerto al primero que encuentre disponible.

Para realizar la asignación de la dirección IP de forma automática (sin tener por que conocer previamente la dirección IP en donde se va a ejecutar el programa) se puede utilizar la constante: INADDR_ANY. Esto le indica al sistema que el programa recibirá mensajes por cualquier IP válida de la máquina, en caso de disponer de varias.

Ejemplo:

```
...
sin.sin_port = 0;
sin.sin_addr.s_addr = htonl (INADDR_ANY);
 ...
```

2.5. Servicio orientado a conexión (TCP)

A continuación se describen los pasos a realizar para la creación de un servicio orientado a conexión, tanto en la parte cliente como en la parte del servidor. En la siguiente figura se muestran los pasos a realizar en ambos casos, invocando a diversas funciones, cuyo funcionamiento será detallado a continuación.

Servicio orientado a conexión (TCP)	
Cliente	**Servidor**
socket()	**socket()**
bind()	**bind()**
	listen()
connect()	**accept()**
send()	**recv()**
recv()	**send()**

Ambos, cliente y servidor, deben crean un socket mediante la funcion socket(), para poder comunicarse.
La llamada a la función bind() en el cliente es opcional, ya que en caso de no ser invocada el sistema la ejecutará automáticamente asignándole un puerto libre al azar. En cambio, en el servidor es obligatorio ejecutar la llamada a la función bind() para reservar un puerto concreto y conocido.

El servidor habilita su socket para poder recibir conexiones, llamando a la funcion listen(). En el cliente este paso no es necesario, ya que no recibirá conexiones de otros procesos.
El servidor ejecuta la funcion accept() y permanece en estado de espera hasta que un cliente se conecte. El cliente usa la funcion connect() para realizar el intento de conexión, en ese momento la funcion accept() del servidor devuelve un parámetro que es un nuevo identificador de socket, el cual es utilizado para realizar la transferencia de datos por la red con el cliente, dejando así libre el socket previamente creado para poder atender nuevas peticiones.
Una vez establecida la conexión se utilizan las funciones send() y recv() con el descriptor de socket del paso anterior para realizar la transferencia de datos.
Para finalizar la conexión se utilizan las funciones close() o shutdown().

2.6. Servicio no orientado a conexión (UDP)

A continuación se describen los pasos a realizar para la creación de un servicio no orientado a conexión.

Servicio no orientado a conexión (UDP)	
Cliente	**Servidor**
socket() **bind()** **sendto()** **recvfrom()**	**socket()** **bind()** **recvfrom()** **sendt()**

El proceso de creación y de asignación de nombres es similar al caso anterior. En cambio el envío y recepción de datos se realiza por medio de las funciones sendto() y recvfrom(), que presentan un comportamiento diferente a las anteriores.

2.7. Funciones orientadas a conexión

La función listen() es invocada únicamente desde el servidor, y habilita al socket para poder recibir conexiones. Únicamente se aplica a sockets de tipo SOCK_STREAM.

El formato de la función es el siguiente:

int listen (int sockfd, int backlog);

sockfd: Es el identificador de socket obtenido en la función socket() que será utilizado para recibir conexiones.

backlog: Es el numero máximo de conexiones en la cola de entrada de conexiones. Las conexiones entrantes quedan en estado de espera en esta cola hasta que sean aceptadas mediante la función accept().

La función accept() es utilizada en el servidor una vez que se ha invocado a la función listen(). Esta función espera hasta que algún cliente establezca una conexión con el servidor. Es una llamada bloqueante, esto es, la función no finalizará hasta que se haya producido una conexión o sea interrumpido por una señal.

Es conveniente destacar que una vez que se ha producido la conexión, la función accept() devuelve un nuevo identificador de socket que será utilizado para la comunicación con el cliente que se ha conectado.

El formato de la función es el siguiente:

int accept (int sockfd, structu sockaddr *addr, int *addrlen)

sockfd: Es el identificador de socket habilitado para recibir conexiones.

addr: Puntero a una estructura sockadd_in, en donde se almacenará la información (dirección IP y número de puerto) del proceso que ha realizado la conexión.

addrlen: Debe ser establecido al tamaño de la estructura sockaddr, mediante la llamada sizeof(struct sockaddr).

El campo addrlen debe contener un puntero a un valor entero que represente el tamaño la estructura addr. Si la función escribe un menor número de bytes, el valor de addrlen es modificado a la cantidad de bytes escritos.

Ejemplo:

```
...
int sockfd, new_sockfd;
struct sockaddr_in  my_addr;
```

```
struct sockaddr_in  remote_addr;
int addrlen;
...
// Creación del socket.
sockfd = socket (AF_INET, SOCK_STREAM, 0 );
...
// Asignar valores a la estructura my_addr.
bind (sockfd, (struct sockaddr *) &my_addr, sizeof(struct sockaddr) );
// Se habilita el socket para poder recibir conexiones.
listen ( sockfd, 5);
addrlen = sizeof (struct sockaddr );
...
// Se llama a accept() y el servidor queda en espera de conexiones.
new_sockfd = accept ( sockfd, &remote_addr, &addrlen);
...
```

La función connect() inicia la conexión con el servidor remoto, por parte del cliente. El formato de la función es el siguiente:

int connect (int sockfd, struct sockaddr *serv_addr, int addrlen)

sockfd: Es el identificador de socket devuelto por la función socket().

serv_addr: Es una estructura sockaddr que contiene la dirección IP y numero de puerto destino.

addrlen: Debe ser inicializado al tamaño de la estructura serv_addr pasada como parámetro.

Una vez que la conexión ha sido establecida, se inicia el intercambio de datos, utilizando para ello las funciones send() y recv(). También es posible emplear otras funciones como write() y read() propias de las operaciones de entrada/salida a través de ficheros.

El formato de la función send() es el siguiente:

send (int sockfd, const void *msg, int len, int flags)

sockfd: Idenficador de socket para enviar datos.

msg: Puntero a los datos a ser enviados.

len: Número de bytes a enviar.
flags: Por defecto, 0. Para más información, consultar la ayuda en línea.

La función send() devuelve el número de bytes enviados, la cual podrá ser menor que la cantidad indicada en el parámetro len.

La función recv() se utiliza para recibir datos, y posee un formato similar a la anterior:

recv (int sockfd, void *buf, int len, unsigned int flags)
sockfd: Identificador de socket para la recepción de los datos.
buf: Puntero a un buffer donde se almacenarán los datos recibidos.
len: Número máximo de bytes a recibir
flags: Por defecto, 0. Para más información, consultar la ayuda en línea.

La función recv() es bloqueante, no finalizando hasta que se ha recibido algún tipo de información. Se debe resaltar que el campo len indica el número máximo de bytes a recibir, pero no necesariamente se han de recibir exactamente ese número de bytes. La función recv() devuelve el número de bytes que se han recibido.

Ejemplo:

```
...
char mens_serv[100];
...
mens_clien = “Ejemplo”;
send(sid, mens_clien, strlen(mens_clien)+1, 0);
...
recv(sid, mens_serv, 100, 0);
...
```

2.8. Funciones no orientadas a conexión

Para el envío y recepción de datos en modo no orientado a conexión (esto es, utilizando UDP) únicamente se utilizan las funciones sendto() y recvfrom().

El formato de la función sendto() es el siguiente:

int sendto(int sockfd, const void *msg, int len, unsigned int flags, const struct sockaddr *to, int tolen)
sockfd: Identificador de socket para el envío de datos.
msg: Puntero a los datos a ser enviados.
len: Longitud de los datos en bytes.
flags: Por defecto, 0. Para más información, consultar la ayuda en línea.
to: Puntero a una estructura sockaddr_in que contiene la dirección IP y numero de puerto destino.
tolen: Debe ser inicializado al tamaño de struct sockaddr, mediante la función sizeof().

La función sendto() devuelve el número de bytes enviados, el cual puede ser menor que el valor indicado en len.

El formato de la función recvfrom() es el siguiente:

int recvfrom (int sockfd, void *buf, int len, unsigned int flags, struct sockaddr *from, int *fromlen)
sockfd: Identificador de socket para la recepción de datos.
buf: Puntero al buffer donde se almacenarán los datos recibidos.
len: Número máximo de bytes a recibir.
flags: Por defecto, 0. Para más información, consultar la ayuda en línea.
from: Puntero a una estructura sockaddr_in (no tiene por que estar previamente inicializada) en donde se rellenarán la dirección IP y número de puerto del proceso que envía los datos.
fromlen: Debe ser inicializado al tamaño de la estructura from, utilizando la función sizeof().

El campo fromlen debe contener un puntero a un valor entero que represente el tamaño la estructura from. Si la función escribe un menor número de bytes, el valor de fromlen es modificado a la cantidad de bytes escritos.

La llamada a la función recvfrom() es bloqueante, no devolviendo el control hasta que se han recibido datos. Al finalizar devuelve el número de bytes recibidos.

Ejemplo:

```
…
char mens_serv[100];
mens_clien = ”Ejemplo”;
... // Previamente se ha rellanado la estructura sserv con la dirección IP y número de puerto del servidor
sendto(sid,mens_clien, strlen(mens_clien)+1,0,(struct sockaddr *)&sserv,sizeof(sserv));
…
long = sizeof(sserv);
recvfrom(sid,mens_serv,100,0,(struct sockaddr *)& sserv,&long);
…
```

2.9. Funciones de cierre de sockets

Básicamente se definen dos funciones para cerrar sockets: close() y shutdown().
El formato de la función close() es el siguiente:

close (sockfd)

Después de utilizar close(), el socket queda deshabilitado para realizar lecturas o escrituras.

El formato de la función shutdown() es el siguiente:

shutdown (sockfd, int how)

Permite deshabilitar la comunicación en una determinada dirección o en ambas direcciones, en base al valor del parámetro how, que puede tomar los siguientes valores:

- 0 : Se deshabilita la recepción.
- 1 : se deshabilita el envío.

- 2 : se deshabilitan la recepción y el envío, igual que en la función close().

2.10. Otras funciones

La función gethostname() devuelve el nombre del sistema en donde se está ejecutando el programa. El formato es el siguiente:

int gethostname (char *hostname, size_t size)

hostname: Puntero a un array de caracteres en donde se almacenará el nombre de la máquina.
size: Longitud en bytes del array hostname.

La función gethostname() devuelve 0 cuando se ha ejecutado con éxito.

La función gethostbyname() se utiliza para convertir un nombre de máquina en una dirección IP. El formato de la función es el siguiente:

struct hostent *gethostbyname (const char *name)
name: Nombre de la máquina que se quiere convertir a dirección IP.

La función devuelve un puntero a una estructura hostent, que esta formada como sigue:

```
struct hostent
{
 char  *h_name;
 char  **h_aliases;
 int  h_addrtype;
 int  h_length;
 char  **h_addr_list;
};
#define h_addr h_addr_list[0]
```

h_name: Nombre oficial de la máquina.

h_aliases: Array de nombres alternativos.

h_addrtype: Tipo de dirección que se retorno (AF_INET).

h_length: Longitud de la dirección en bytes.

h_addr_list: Array de direcciones de red para la máquina.

h_addr: La primer dirección en h_addr_list.

En el caso de producirse algún error devuelve NULL y establece la variable h_errno con el numero de error.

3. Información adicional

3.1. Función socket

NAME

socket - create an endpoint for communication

SYNOPSIS

#include <sys/types.h>

#include <sys/socket.h>

int

socket(int domain, int type, int protocol)

DESCRIPTION

Socket() creates an endpoint for communication and returns a descriptor.

The domain parameter specifies a communications domain within which communication will take place; this selects the protocol family which should be used. These families are defined in the include file <sys/socket.h>. The currently understood formats are

AF_UNIX

(UNIX internal protocols),

AF_INET

(ARPA Internet protocols),

AF_ISO

(ISO protocols),

AF_NS

(Xerox Network Systems protocols), and

AF_IMPLINK
(IMP host at IMP link layer).
The socket has the indicated type, which specifies the semantics of communication. Currently defined types are:
SOCK_STREAM
SOCK_DGRAM
SOCK_RAW
SOCK_SEQPACKET
SOCK_RDM
A SOCK_STREAM type provides sequenced, reliable, two-way connection based byte streams. An out-of-band data transmission mechanism may be supported. A SOCK_DGRAM socket supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length). A SOCK_SEQPACKET socket may provide a sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length; a consumer may be required to read an entire packet with each read system call. This facility is protocol specific, and presently implemented only for PF_NS. SOCK_RAW sockets provide access to internal network protocols and interfaces. The types SOCK_RAW, which is available only to the super-user, and SOCK_RDM, which is planned, but not yet implemented, are not described here.

The protocol specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner. The protocol number to use is particular to the communication domain in which communication is to take place; see protocols(5).
Sockets of type SOCK_STREAM are full-duplex byte streams, similar to pipes. A stream socket must be in a connected state before any data may be sent or received on it. A connection to another socket is created with a connect(2) call. Once connected, data may be transferred using read(2) and write(2) calls or some variant of the send(2) and recv(2) calls. When a session

has been completed a close(2) may be performed. Out-of-band data may also be transmitted as described in send(2) and received as described in recv(2). The communications protocols used to implement a SOCK_STREAM insure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered broken and calls will indicate an error with -1 returns and with ETIMEDOUT as the specific code in the global variable errno. The protocols optionally keep sockets ``warm'' by forcing transmissions roughly every minute in the absence of other activity. An error is then indicated if no response can be elicited on an otherwise idle connection for a extended period (e.g. 5 minutes). A SIGPIPE signal is raised if a process sends on a broken stream; this causes naive processes, which do not handle the signal, to exit.
SOCK_SEQPACKET sockets employ the same system calls as SOCK_STREAM sockets. The only difference is that read(2) calls will return only the amount of data requested, and any remaining in the arriving packet will be discarded.
SOCK_DGRAM and SOCK_RAW sockets allow sending of datagrams to correspondents named in send(2) calls. Datagrams are generally received with recvfrom(2), which returns the next datagram with its return address.

An fcntl(2) call can be used to specify a process group to receive a SIGURG signal when the out-of-band data arrives. It may also enable nonblocking I/O and asynchronous notification of I/O events via SIGIO.
The operation of sockets is controlled by socket level options. These options are defined in the file <sys/socket.h>. Setsockopt(2) and getsockopt(2) are used to set and get options, respectively.
RETURN VALUES
A -1 is returned if an error occurs, otherwise the return value is a descriptor referencing the socket.
ERRORS
The socket() call fails if:
[EPROTONOSUPPORT] The protocol type or the specified protocol is not supported within this domain.

[EMFILE]
The per-process descriptor table is full.
[ENFILE]
The system file table is full.
[EACCESS]
Permission to create a socket of the specified type and/or protocol is denied.
[ENOBUFS]
Insufficient buffer space is available. The socket cannot be created until sufficient resources are freed.
SEE ALSO
accept(2), bind(2), connect(2), getprotoent(3), getsockname(2), getsockopt(2), ioctl(2), listen(2), read(2), recv(2), select(2), send(2), shutdown(2), socketpair(2), write(2)

3.2. Funciones inet

NAME
inet_addr, inet_network, inet_makeaddr, inet_lnaof, inet_netof, inet_ntoa - Internet address manipulation

SYNOPSIS
#include <arpa/inet.h>
in_addr_t inet_addr(const char *cp);
in_addr_t inet_lnaof(struct in_addr in);
struct in_addr inet_makeaddr(in_addr_t net, in_addr_t lna);
in_addr_t inet_netof(struct in_addr in);
in_addr_t inet_network(const char *cp);
char *inet_ntoa(struct in_addr in);
DESCRIPTION
The inet_addr() function converts the string pointed to by cp, in the Internet standard dot notation, to an integer value suitable for use as an Internet address.
The inet_lnaof() function takes an Internet host address specified by in and extracts the local network address part, in host byte order.

The inet_makeaddr() function takes the Internet network number specified by net and the local network address specified by lna, both in host byte order, and constructs an Internet address from them.
The inet_netof() function takes an Internet host address specified by in and extracts the network number part, in host byte order.
The inet_network() function converts the string pointed to by cp, in the Internet standard dot notation, to an integer value suitable for use as an Internet network number.
The inet_ntoa() function converts the Internet host address specified by in to a string in the Internet standard dot notation.
All Internet addresses are returned in network order (bytes ordered from left to right).
Values specified using dot notation take one of the following forms:
a.b.c.d
When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address.

a.b.c
When a three-part address is specified, the last part is interpreted as a 16-bit quantity and placed in the rightmost two bytes of the network address. This makes the three-part address format convenient for specifying Class B network addresses as 128.net.host.
a.b
When a two-part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the rightmost three bytes of the network address. This makes the two-part address format convenient for specifying Class A network addresses as net.host.
a
When only one part is given, the value is stored directly in the network address without any byte rearrangement.
All numbers supplied as parts in dot notation may be decimal, octal, or implies hexadecimal; otherwise, a leading 0 implies octal; otherwise, the number is interpreted as decimal).
RETURN VALUES

Upon successful completion, inet_addr() returns the Internet address. Otherwise, it returns (in_addr_t)-1.

Upon successful completion, inet_network() returns the converted Internet network number. Otherwise, it returns (in_addr_t)-1.

The inet_makeaddr() function returns the constructed Internet address.

The inet_lnaof() function returns the local network address part.

The inet_netof() function returns the network number.

The inet_ntoa() function returns a pointer to the network address in Internet-standard dot notation.

ERRORS

No errors are defined.

USAGE

The return value of inet_ntoa() may point to static data that may be overwritten by subsequent calls to inet_ntoa().

SEE ALSO

endhostent(2), endnetent(2), inet(2)

3.3. Funciones htonl, htons, ntohl, ntohs

NAME

htonl, htons, ntohl, ntohs - convert values between host and network byte order

SYNOPSIS

#include <arpa/inet.h>

in_addr_t htonl(in_addr_t hostlong);

in_port_t htons(in_port_t hostshort);

in_addr_t ntohl(in_addr_t netlong);

in_port_t ntohs(in_port_t netshort);

DESCRIPTION

These functions convert 16-bit and 32-bit quantities between network byte order and host byte order.

RETURN VALUES

The htonl() and htons() functions return the argument value converted from host to network byte order.

The ntohl() and ntohs() functions return the argument value converted from network to host byte order.

ERRORS

No errors are defined.

USAGE

These functions are most often used in conjunction with Internet addresses and ports as returned by gethostent(3XN) and getservent(3XN).

On some architectures these functions are defined as macros that expand to the value of their argument.

SEE ALSO

endhostent(2), endservent(2), inet(2)

3.4. Función bind

NAME

bind - bind a name to a socket

SYNOPSIS

#include <sys/types.h>

#include <sys/socket.h>

int

bind(int s, struct sockaddr *name, int namelen)

DESCRIPTION

Bind() assigns a name to an unnamed socket. When a socket is created with socket(2) it exists in a name space (address family) but has no name assigned. Bind() requests that name be assigned to the socket.

NOTES

Binding a name in the UNIX domain creates a socket in the file system that must be deleted by the caller when it is no longer needed (using unlink(2)).

The rules used in name binding vary between communication domains. Consult the manual entries in section 4 for detailed information.

RETURN VALUES

If the bind is successful, a 0 value is returned. A return value of -1 indicates an error, which is further specified in the global errno.

ERRORS

The bind() call will fail if:

[EBADF]

S is not a valid descriptor.

[ENOTSOCK]

S is not a socket.

[EADDRNOTAVAIL]

The specified address is not available from the local machine.

[EADDRINUSE]

The specified address is already in use.

[EINVAL]

The socket is already bound to an address.

[EACCES]

The requested address is protected, and the current user has inadequate permission to access it.

[EFAULT]

The name parameter is not in a valid part of the user address space.

The following errors are specific to binding names in the UNIX domain.

[ENOTDIR]

A component of the path prefix is not a directory.

[EINVAL]

The pathname contains a character with the high-order bit set.

[ENAMETOOLONG]

A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT]

A prefix component of the path name does not exist.

[ELOOP]

Too many symbolic links were encountered in translating the pathname.

[EIO]

An I/O error occurred while making the directory entry or allocating the inode.

[EROFS]

The name would reside on a read-only file system.

[EISDIR]

An empty pathname was specified.

SEE ALSO

connect(2), listen(2), socket(2), getsockname(2)

3.5. Función accept

NAME

accept - accept a connection on a socket

SYNOPSIS

#include <sys/types.h>

#include <sys/socket.h>

int

accept(int s, struct sockaddr *addr, int *addrlen)

DESCRIPTION

The argument s is a socket that has been created with socket(2), bound to an address with bind(2), and is listening for connections after a listen(2). The accept() argument extracts the first connection request on the queue of pending connections, creates a new socket with the same properties of s and allocates a new file descriptor for the socket. If no pending connections are present on the queue, and the socket is not marked as non-blocking, accept() blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, accept() returns an error as described below. The accepted socket may not be used to accept more connections. The original socket s remains open.

The argument addr is a result parameter that is filled in with the address of the connecting entity, as known to the communications layer. The exact format of the addr parameter is determined by the domain in which the communication is occurring. The addrlen is a value-result parameter; it should initially contain the amount of space pointed to by addr; on return it will contain the actual length (in bytes) of the address returned. This call is used with connection-based socket types, currently with SOCK_STREAM.

It is possible to select(2) a socket for the purposes of doing an accept() by selecting it for read.

For certain protocols which require an explicit confirmation, such as ISO or DATAKIT, accept() can be thought of as merely dequeueing the next

connection request and not implying confirmation. Confirmation can be implied by a normal read or write on the new file desciptor, and rejection can be implied by closing the new socket.

One can obtain user connection request data without confirming the connection by issuing a recvmsg(2) call with an msg_iovlen of 0 and a nonzero msg_controllen, or by issuing a getsockopt(2) request. Similarly, one can provide user connection rejection information by issuing a sendmsg(2) call with providing only the control information, or by calling setsockopt(2).
RETURN VALUES
The call returns -1 on error. If it succeeds, it returns a non-negative integer that is a descriptor for the accepted socket.
ERRORS
The accept() will fail if:
[EBADF]
The descriptor is invalid.
[ENOTSOCK]
The descriptor references a file, not a socket.
[EOPNOTSUPP]
The referenced socket is not of type SOCK_STREAM.
[EFAULT]
The addr parameter is not in a writable part of the user address space.
[EWOULDBLOCK]
The socket is marked non-blocking and no connections are present to be accepted.
SEE ALSO
bind(2), connect(2), listen(2), select(2), socket(2)

3.6. Función connect

NAME
connect - initiate a connection on a socket
SYNOPSIS
#include <sys/types.h>
#include <sys/socket.h>

int

connect(int s, struct sockaddr *name, int namelen)

DESCRIPTION

The parameter s is a socket. If it is of type SOCK_DGRAM, this call specifies the peer with which the socket is to be associated; this address is that to which datagrams are to be sent, and the only address from which datagrams are to be received. If the socket is of type SOCK_STREAM, this call attempts to make a connection to another socket. The other socket is specified by name, which is an address in the communications space of the socket. Each communications space interprets the name parameter in its own way. Generally, stream sockets may successfully connect() only once; datagram sockets may use connect() multiple times to change their association. Datagram sockets may dissolve the association by connecting to an invalid address, such as a null address.

RETURN VALUES

If the connection or binding succeeds, 0 is returned. Otherwise a -1 is returned, and a more specific error code is stored in errno.

ERRORS

The connect() call fails if:

[EBADF]

S is not a valid descriptor.

[ENOTSOCK]

S is a descriptor for a file, not a socket.

[EADDRNOTAVAIL]

The specified address is not available on this machine.

[EAFNOSUPPORT]

Addresses in the specified address family cannot be used with this socket.

[EISCONN]

The socket is already connected.

[ETIMEDOUT]

Connection establishment timed out without establishing a connection.

[ECONNREFUSED]

The attempt to connect was forcefully rejected.

[ENETUNREACH]

The network isn't reachable from this host.

[EADDRINUSE]

The address is already in use.

[EFAULT]

The name parameter specifies an area outside the process address space.

[EINPROGRESS]

The socket is non-blocking and the connection cannot be completed immediately. It is possible to select(2) for completion by selecting the socket for writing.

[EALREADY]

The socket is non-blocking and a previous connection attempt has not yet been completed.

The following errors are specific to connecting names in the UNIX domain. These errors may not apply in future versions of the UNIX IPC domain.

[ENOTDIR]

A component of the path prefix is not a directory.

[EINVAL]

The pathname contains a character with the high-order bit set.

[ENAMETOOLONG]

A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT]

The named socket does not exist.

[EACCES]

Search permission is denied for a component of the path prefix.

[EACCES]

Write access to the named socket is denied.

[ELOOP]

Too many symbolic links were encountered in translating the pathname.

SEE ALSO

accept(2), select(2), socket(2), getsockname(2)

3.7. Función listen

NAME

listen - listen for connections on a socket

SYNOPSIS

#include <sys/socket.h>

int

listen(int s, int backlog)

DESCRIPTION

To accept connections, a socket is first created with socket(2), a willingness to accept incoming connections and a queue limit for incoming connections are specified with listen(), and then the connections are accepted with accept(2). The listen() call applies only to sockets of type SOCK_STREAM or SOCK_SEQPACKET.

The backlog parameter defines the maximum length the queue of pending connections may grow to. If a connection request arrives with the queue full the client may receive an error with an indication of ECONNREFUSED, or, if the underlying protocol supports retransmission, the request may be ignored so that retries may succeed.

RETURN VALUES

A 0 return value indicates success; -1 indicates an error.

ERRORS

Listen (will, fail, if:)

[EBADF]

The argument s is not a valid descriptor.

[ENOTSOCK]

The argument s is not a socket.

[EOPNOTSUPP] The socket is not of a type that supports the operation listen().

SEE ALSO

accept(2), connect(2), socket(2)

3.8. Funciones send y sendto

NAME

send, sendto, sendmsg - send a message from a socket

SYNOPSIS

#include <sys/types.h>

#include <sys/socket.h>

int

send(int s, const void *msg, int len, int flags)

int

sendto(int s, const void *msg, int len, int flags, const struct sockaddr *to, int tolen)

int

sendmsg(int s, const struct msghdr *msg, int flags)

DESCRIPTION

Send(), sendto(), and sendmsg() are used to transmit a message to another socket. Send() may be used only when the socket is in a connected state, while sendto() and sendmsg() may be used at any time.

The address of the target is given by to with tolen specifying its size. The length of the message is given by len. If the message is too long to pass atomically through the underlying protocol, the error EMSGSIZE is returned, and the message is not transmitted.

No indication of failure to deliver is implicit in a send(). Locally detected errors are indicated by a return value of -1.

If no messages space is available at the socket to hold the message to be transmitted, then send() normally blocks, unless the socket has been placed in non-blocking I/O mode. The select(2) call may be used to determine when it is possible to send more data.

The flags parameter may include one or more of the following:

#define MSG_OOB

0x1 /* process out-of-band data */ #define MSG_DONTROUTE 0x4 /* bypass routing, use direct interface */

The flag MSG_OOB is used to send ``out-of-band'' data on sockets that support this notion (e.g. SOCK_STREAM); the underlying protocol must also support ``out-of-band'' data. MSG_DONTROUTE is usually used only by diagnostic or routing programs.

See recv(2) for a description of the msghdr structure.

RETURN VALUES

The call returns the number of characters sent, or -1 if an error occurred.

ERRORS

Send(), sendto(), and sendmsg() fail if:

[EBADF]

An invalid descriptor was specified.

[ENOTSOCK]

The argument s is not a socket.

[EFAULT]

An invalid user space address was specified for a parameter.

[EMSGSIZE]

The socket requires that message be sent atomically, and the size of the message to be sent made this impossible.

[EWOULDBLOCK] The socket is marked non-blocking and the requested operation would block.

[ENOBUFS]

The system was unable to allocate an internal buffer. The operation may succeed when buffers become available.

[ENOBUFS]

The output queue for a network interface was full. This generally indicates that the interface has stopped sending, but may be caused by transient congestion.

SEE ALSO

fcntl(2), recv(2), select(2), getsockopt(2), socket(2), write(2)

3.9. Funciones recv y recvfrom

NAME

recv, recvfrom, recvmsg - receive a message from a socket

SYNOPSIS

#include <sys/types.h>

#include <sys/socket.h>

int

recv(int s, void *buf, int len, int flags)

int

recvfrom(int s, void *buf, int len, int flags, struct sockaddr *from, int *fromlen)

int

recvmsg(int s, struct msghdr *msg, int flags)

DESCRIPTION

Recvfrom() and recvmsg() are used to receive messages from a socket, and may be used to receive data on a socket whether or not it is connectionoriented.

If from is non-nil, and the socket is not connection-oriented, the source address of the message is filled in. Fromlen is a value-result parameter, initialized to the size of the buffer associated with from, and modified on return to indicate the actual size of the address stored there.

The recv() call is normally used only on a connected socket (see connect(2)) and is identical to recvfrom() with a nil from parameter. As it is redundant, it may not be supported in future releases.

All three routines return the length of the message on successful completion. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from (see socket(2)).

If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is nonblocking (see fcntl(2)) in which case the value -1 is returned and the external variable errno set to EWOULDBLOCK. The receive calls normally return any data available, up to the requested amount, rather than waiting for receipt of the full amount requested; this behavior is affected by the socket-level options SO_RCVLOWAT and SO_RCVTIMEO described in getsockopt(2).

The select(2) call may be used to determine when more data arrive.

The flags argument to a recv call is formed by or'ing one or more of the values:

MSG_OOB

process out-of-band data

MSG_PEEK

peek at incoming message

MSG_WAITALL

wait for full request or error The MSG_OOB flag requests receipt of out-of-band data that would not be received in the normal data stream. Some protocols place expedited data at the head of the normal data queue, and thus this flag cannot be used with such protocols. The MSG_PEEK flag causes the

receive operation to return data from the beginning of the receive queue without removing that data from the queue. Thus, a subsequent receive call will return the same data. The MSG_WAITALL flag requests that the operation block until the full request is satisfied. However, the call may still return less data than requested if a signal is caught, an error or disconnect occurs, or the next data to be received is of a different type than that returned.
The recvmsg() call uses a msghdr structure to minimize the number of directly supplied parameters. This structure has the following form, as defined in <sys/socket.h>:

```
struct msghdr {
caddr_t msg_name;
/* optional address */ u_int msg_namelen; /* size of address */ struct iovec
*msg_iov; /* scatter/gather array */ u_int msg_iovlen; /* # elements in msg_iov
*/
caddr_t msg_control;
/* ancillary data, see below */
u_int
msg_controllen; /* ancillary data buffer len */
int
msg_flags; /* flags on received message */ };
```

Here msg_name and msg_namelen specify the destination address if the socket is unconnected; msg_name may be given as a null pointer if no names are desired or required. Msg_iov and msg_iovlen describe scatter gather locations, as discussed in read(2). Msg_control, which has length msg_controllen, points to a buffer for other protocol control related messages or other miscellaneous ancillary data. The messages are of the form:

```
struct cmsghdr {
u_int cmsg_len; /* data byte count, including hdr */
int
cmsg_level; /* originating protocol */
int
cmsg_type; /* protocol-specific type */ /* followed by u_char cmsg_data[]; */ };
```

As an example, one could use this to learn of changes in the data-stream in

XNS/SPP, or in ISO, to obtain user-connection-request data by requesting a recvmsg with no data buffer provided immediately after an accept() call.
Open file descriptors are now passed as ancillary data for AF_UNIX domain sockets, with cmsg_level set to SOL_SOCKET and cmsg_type set to SCM_RIGHTS.
The msg_flags field is set on return according to the message received. MSG_EOR indicates end-of-record; the data returned completed a record (generally used with sockets of type SOCK_SEQPACKET). MSG_TRUNC indicates that the trailing portion of a datagram was discarded because the datagram was larger than the buffer supplied. MSG_CTRUNC indicates that some control data were discarded due to lack of space in the buffer for ancillary data. MSG_OOB is returned to indicate that expedited or out-of-band data were received.
RETURN VALUES
These calls return the number of bytes received, or -1 if an error occurred.
ERRORS
The calls fail if:
[EBADF]
The argument s is an invalid descriptor.
[ENOTCONN]
The socket is assoicated with a connection-oriented protocol and has not been connected (see connect(2) and accept(2)).
[ENOTSOCK]
The argument s does not refer to a socket.
[EWOULDBLOCK] The socket is marked non-blocking, and the receive operation would block, or a receive timeout had been set, and
the timeout expired before data were received.
[EINTR]
The receive was interrupted by delivery of a signal before any data were available.
[EFAULT]
The receive buffer pointer(s) point outside the process's address space.
SEE ALSO
fcntl(2), read(2), select(2), getsockopt(2), socket(2)

3.10. Función close

NAME

close - delete a descriptor

SYNOPSIS

#include <unistd.h>

int

close(int d)

DESCRIPTION

The close() call deletes a descriptor from the per-process object reference table. If this is the last reference to the underlying object, the object will be deactivated. For example, on the last close of a file the current seek pointer associated with the file is lost; on the last close of a socket(2) associated naming information and queued data are discarded; on the last close of a file holding an advisory lock the lock is released (see further flock(2)).

When a process exits, all associated file descriptors are freed, but since there is a limit on active descriptors per processes, the close() function call is useful when a large quanitity of file descriptors are being handled.

When a process forks (see fork(2)), all descriptors for the new child process reference the same objects as they did in the parent before the fork. If a new process is then to be run using execve(2), the process would normally inherit these descriptors. Most of the descriptors can be rearranged with dup2(2) or deleted with close() before the execve is attempted, but if some of these descriptors will still be needed if the execve fails, it is necessary to arrange for them to be closed if the execve succeeds. For this reason, the call ``fcntl(d, F_SETFD, 1)'' is provided, which arranges that a descriptor will be closed after a successful execve; the call ``fcntl(d, F_SETFD, 0)'' restores the default, which is to not close the descriptor.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the global integer variable errno is set to indicate the error.

ERRORS

Close() will fail if:

[EBADF]

D is not an active descriptor.

[EINTR]

An interupt was received.

SEE ALSO

accept(2), flock(2), open(2), pipe(2), socket(2), socketpair(2), execve(2), fcntl(2)

3.11. Función shutdown

NAME

shutdown - shut down part of a full-duplex connection

SYNOPSIS

#include <sys/socket.h>

int

shutdown(int s, int how)

DESCRIPTION

The shutdown() call causes all or part of a full-duplex connection on the socket associated with s to be shut down. If how is 0, further receives will be disallowed. If how is 1, further sends will be disallowed. If how is 2, further sends and receives will be disallowed.

DIAGNOSTICS

A 0 is returned if the call succeeds, -1 if it fails.

ERRORS

The call succeeds unless:

[EBADF]

S is not a valid descriptor.

[ENOTSOCK] S is a file, not a socket.

[ENOTCONN] The specified socket is not connected.

SEE ALSO

connect(2), socket(2)

3.12. Función gethostbyname

NAME

endhostent, gethostbyaddr, gethostbyname, gethostent, sethostent - network host database functions

SYNOPSIS

#include <netdb.h>

extern int h_errno;

void endhostent(void);

struct hostent *gethostbyaddr(const void *addr, size_t len, int type);

struct hostent *gethostbyname(const char *name);

struct hostent *gethostent(void);

void sethostent(int stayopen);

DESCRIPTION

The gethostent(), gethostbyaddr(), and gethostbyname() functions each return a pointer to a hostent structure, the members of which contain the fields of an entry in the network host database.

The gethostent() function reads the next entry of the database, opening a connection to the database if necessary.

The gethostbyaddr() function searches the database from the beginning and finds the first entry for which the address family specified by type matches the h_addrtype member and the address pointed to by addr occurs in h_addrlist, opening a connection to the database if necessary. The addr argument is a pointer to the binary-format (that is, not nullterminated) address in network byte order, whose length is specified by the len argument. The datatype of the address depends on the address family. For an address of type AF_INET, this is an in_addr structure, defined in <netinet/in.h>.

The gethostbyname() function searches the database from the beginning and finds the first entry for which the host name specified by name matches the h_name member, opening a connection to the database if necessary.

The sethostent() function opens a connection to the network host database, and sets the position of the next entry to the first entry. If the stayopen argument is non-zero, the connection to the host database will not be closed after each call to gethostent() (either directly, or indirectly through one of the other gethost*() functions).

The endhostent() function closes the connection to the database.

RETURN VALUES

On successful completion, gethostbyaddr(), gethostbyname() and gethostent() return a pointer to a hostent structure if the requested entry was found, and a

null pointer if the end of the database was reached or the requested entry was not found. Otherwise, a null pointer is returned.
On unsuccessful completion, gethostbyaddr() and gethostbyname() functions set h_errno to indicate the error.

ERRORS
No errors are defined for endhostent(), gethostent() and sethostent().
The gethostbyaddr() and gethostbyname() functions will fail in the following cases, setting h_errno to the value shown in the list below. Any changes to errno are unspecified.
HOST_NOT_FOUND
No such host is known.
TRY_AGAIN
A temporary and possibly transient error occurred, such as a failure of a server to respond.
NO_RECOVERY
An unexpected server failure occurred which can not be recovered.
NO_DATA
The server recognized the request and the name but no address is available. Another type of request to the name server for the domain might return an answer.
USAGE
The gethostent(), gethostbyaddr(), and gethostbyname() functions may return pointers to static data, which may be overwritten by subsequent calls to any of these functions.
These functions are generally used with the Internet address family.
SEE ALSO
endservent(2), htonl(2), inet_addr(2)