Estructuras y funciones de programación de sockets.

Autor: Enrique Bonet

Introducción.

En estos apuntes se incluye una breve descripción de las estructuras y funciones cuyo uso puede ser necesario para el desarrollo de las prácticas de programación de sockets planteadas en el presente curso.

Las funciones están clasificadas de acuerdo a las tareas que realizan, tales como creación y cierre de sockets, lectura y escritura, etc. A su vez, cada descripción de las funciones está organizada con la siguiente estructura interna:

- Fichero o ficheros de cabecera que han de incluirse para el uso de la función.
- Prototipo de la función.
- Breve descripción de la misma así como de los parámetros que utiliza.
- Ejemplo de uso de la misma[1].

Una descripción más detallada de estas funciones, así como información sobre cualquier otra función no especificada en estos apuntes, puede encontrase consultando las páginas de manual de Linux mediante el comando man.

Creación y cierre.

socket.

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

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

La función *socket* crea un extremo de una comunicación y devuelve un descriptor. Los argumentos son:

dominio	Dominio de comunicación deseado. Permite seleccionar la familia de protocolos que se utilizará en la comunicación.
tipo	Especifica la semántica de la comunicación, sus valores más comunes son *SOCK_STREAM* y *SOCK_DGRAM*.
protocolo	Protocolo particular a utilizar. Generalmente un dominio y tipo solo admite un protocolo particular, por lo que suele tomar el valor 0.

Si tiene éxito, la función devuelve un *int* que indica el valor del conector. En caso de error el valor es -1. Ejemplo:

[1] Cuando en los ejemplos aparezcan tres puntos seguidos en una línea, esto debe entenderse como que es necesario realizar acciones anteriores no indicadas para que el ejemplo funcione. Por ejemplo, para poder cerrar un socket es necesario haberlo creado previamente, etc.

```
int sock;
if ((sock=socket(PF_INET, SOCK_STREAM, 0))<0)
{
      perror("socket");
      exit(0);
}
```

close.

```
#include <unistd.h>

int close(int fd);
```

La función *close* cierra un descriptor de fichero o socket. Su argumento es:

fd	Descriptor del socket a cerrar.

La función *close* devuelve 0 si sucede. En caso de error devuelve el valor -1. Ejemplo:

```
int sock;
...
if (close(sock)!=0)
{
      perror("close");
      exit(0);
}
```

Asociación a puertos y especificación de propiedades.

bind.

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

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

La función *bind* asocia el socket dado por *sockfd* a la dirección local especificada por *addr* para que el socket quede asignado al puerto especificado en la misma. Sus argumentos son:

sockfd	Descriptor del socket creado con anterioridad.
addr	Estructura de datos donde se especifica la dirección y puerto al que se asocia el socket.
addrlen	Longitud de la estructura de datos anterior.

La estructura *sockaddr* no suele ser utilizada, siendo siempre utilizada en su lugar la estructura *sockaddr_in*, cuya declaración se encuentra en el fichero *netinet/in.h*, por lo que para su uso debe incluirse dicho de cabecera como:

```
#include <netinet/in.h>
```

La estructura *sockaddr_in* tiene los siguientes campos:

```
struct in_addr
{
     unsigned long int s_addr;
};
struct sockaddr_in
{
     int sin_family;
     unsigned short int sin_port;
     struct in_addr sin_addr;
};
```

En esta estructura, cada uno de los campos tiene la siguiente utilidad:

sin_family	Dominio de comunicación deseado. Debe coincidir con el que se indicó en la creación del socket.
port	Puerto al que se asocia el socket.
sin_addr.s_addr	Dirección local a la que se asigna el socket. Para asignar el socket a todas las direcciones locales se debe utilizar el valor *INADDR_ANY*.

La función *bind* devuelve 0 en caso de éxito. El valor -1 es devuelto si sucede un error. Ejemplo[2]:

```
int sock;
struct sockaddr_in s;
...
s.sin_family=PF_INET;
s.sin_port=htons(puerto);
s.sin_addr.s_addr=htonl(INADDR_ANY);
if (bind(sock, (struct sockaddr *)&s,
         sizeof(struct sockaddr_in))!=0)
{
     perror("bind");
     exit(0);
}
```

listen.

```
#include <sys/socket.h>

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

La función listen pone el socket s, asociado previamente a una dirección y puerto, en escucha y especifica el tamaño de la cola de espera de aceptación de conexiones del socket. La descripción de sus argumentos es:

s	Descriptor del socket creado con anterioridad.
backlog	Longitud máxima de la cola de conexiones pendientes, el valor máximo que es valido es 128 en Linux, aunque implementaciones como BSD, por ejemplo, lo limitan a 5.

[2] Las funciones *htons* y *htonl* usadas en el ejemplo se explicarán con posterioridad.

La función *listen* devuelve 0 si sucede y -1 en caso de error. Ejemplo:

```
int sock;
...
if (listen(sock,5)!=0)
{
     perror("listen");
     exit(0);
}
```

ioctl.

```
#include <sys/ioctl.h>

int ioctl(int d, int peticion, &);
```

La función *ioctl* manipula los valores de los parámetros de un socket. La función *ioctl* posee un número variable de argumentos según el valor a modificar, por ello solo explicaremos el caso que puede ser necesario utilizar en los programas propuestos. Dicho caso consiste en la modificación del modo de funcionamiento del socket.

Un socket puede funcionar en modo bloqueante, en el cual espera hasta que se produzca la petición solicitada (lectura de datos, escritura de datos, etc.), o bien en modo no bloqueante, en el cual intenta la petición solicitada y si esta disponible la realiza, terminando inmediatamente, sin ningún tipo de espera, en caso contrario. En nuestro caso particular la función *ioctl* toma la forma:

```
int ioctl(int d, int peticion, int &tipo);
```

Donde los valores de los argumentos son:

d	Descriptor del socket creado con anterioridad.
peticion	Propiedad a cambiar, en nuestro caso de explicación, modo de funcionamiento del socket, que se indica mediante la constante FIONBIO.
tipo	Modo de funcionamiento. Indica funcionamiento bloqueante (valor 0) o no bloqueante (valor 1).

La función *ioctl* devuelve 0 si sucede y -1 falla. Ejemplo:

```
int sock, modo;
...
modo=0; /* Modo bloqueante de funcionamiento del socket */
if (ioctl(sock,FIONBIO,&modo)!=0)
{
     perror("ioctl");
     exit(0);
}
```

Aceptación y petición de conexiones.

accept.

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

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

La función *accept* acepta una petición de conexión al socket especificado por s. Los parámetros son:

s	Descriptor del socket creado con anterioridad.
addr	Estructura de datos que contendrá la dirección y puerto del descriptor que se ha conectado a este socket después de la llamada.
addrlen	Puntero al entero que contiene la longitud de la estructura de datos anterior.

La estructura *sockaddr* no suele ser utilizada, siendo siempre utilizada en su lugar la estructura *sockaddr_in*, explicada con anterioridad.

La función devuelve un entero no negativo que es el descriptor del socket aceptado o -1 si sucede un error. El socket original (parámetro s) permanece en cualquier caso inalterado, pudiendo ser utilizado en posteriores llamadas a la función. Ejemplo:

```
int sock, sock_conectado, tam;
struct sockaddr_in s;
...
tam=sizeof(struct sockaddr_in);
if ((sock_conectado=accept(sock,(struct sockaddr *)&s,&tam))<0)
{
      perror("accept");
      exit(0);
}
```

connect.

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

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

La función *connect* solicita poder conectar el socket especificado por *sockfd* a un socket remoto cuya direccíon y puerto se especifica en *serv_addr*. Los parámetros son:

sockfd	Descriptor del socket creado con anterioridad.
serv_addr	Estructura de datos donde se especifica la dirección y puerto con el que deseamos establecer la conexión.
addrlen	Longitud de la estructura de datos anterior.

La estructura *sockaddr* no suele ser utilizada, siendo siempre utilizada en su lugar la estructura *sockaddr_in*, explicada con anterioridad.

La función devuelve el valor -1 si error o 0 si su llamada tiene éxito. Ejemplo[3]:

```
int sock;
struct sockaddr_in s;
...
s.sin_family=PF_INET;
s.sin_port=htons(13);
if (inet_aton("147.156.1.1", &s.sin_addr)==0)
{
     perror("inet_aton");
     exit(0);
}
if (connect(sock, (struct sockaddr *)&s,
            sizeof(struct sockaddr_in))<0)
{
     if (errno==ECONNREFUSED) /* Servicio no disponible */
          ...
     else /* Error de otro tipo */
     {
          perror("connect");
          exit(0);
     }
}
```

Comentar en este ejemplo que la variable errno es una variable global definida en <*errno.h*> y que contiene el valor del último error que ha sucedido en la llamada a una función de librería.

Comprobación del estado de un socket.

select.

```
#include <sys/types.h>
#include <sys/time.h>
#include <unistd.h>

int select(int n, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

La función *select* comprueba el estado de un socket. La comprobación afecta a tres conjuntos distintos de descriptores de socket (*readfds*, *writefds* y *exceptfds*). Los parámetros son:

n	Valor, incrementado en una unidad, del descriptor de socket más alto, de cualquiera de los tres conjuntos, que se desea comprobar.
readfds	Conjunto de sockets que serán comprobados para ver si existen caracteres para leer. Si el socket es de tipo *SOCK_STREAM* y no esta conectado, también se modificará este conjunto si llega una petición de conexión.

[3] Las funciones *htons* e *inet_aton* usadas en el ejemplo se explicarán con posterioridad.

writefds	Conjunto de sockets que serán comprobados para ver si se puede escribir en ellos.
exceptfds	Conjunto de sockets que serán comprobados para ver si ocurren excepciones.
timeout	Limite superior de tiempo antes de que la llamada a *select* termine. Si *timeout* es *NULL*, la función *select* no termina hasta que se produzca algún cambio en uno de los conjuntos (llamada bloqueante a *select*).

La declaración de la estructura timeval es la siguiente:

```
struct timeval
{
      unsigned long int tv_sec; /* Segundos */
      unsigned long int tv_usec; /* Millonesimas de segundo */
};
```

Si no se desea comprobar alguna de las condiciones que proporciona la función *select* (lectura, escritura y excepciones), puede sustituirse el puntero al conjunto por un puntero *NULL*.

Para manejar el conjunto *fd_set* se proporcionan cuatro macros:

```
FD_ZERO(fd_set *set);
FD_SET(int fd, fd_set *set);
FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
```

Donde la función de cada macro es:

FD_ZERO	Inicializa el conjunto *fd_set* especificado por *set*
FD_SET	Añade un descriptor de socket dado por *fd* al conjunto dado por *set*.
FD_CLR	Borra un descriptor de socket dado por *fd* al conjunto dado por *set*.
FD_ISSET	Comprueba si el descriptor de socket dado por *fd* se encuentra en el conjunto especificado por *set*.

La función devuelve el valor -1 en caso de error y un número, cuyo valor es 0 si se produce el timeout antes de que suceda ninguna modificación en el estado de los descriptores incluidos en los conjuntos, o un valor mayor que 0 si se produce una modificación en algún descriptor. El número mayor que 0 indica el número de descriptores que han sufrido una modificación de forma simultanea, pues basta un solo cambio para salir de la función. Ejemplo:

```
int sock, n;
fd_set conjunto;
struct timeval timeout;
...
FD_ZERO(&conjunto);
FD_SET(sock, &conjunto);
timeout.tv_sec=1;
timeout.tv_usec=0;
if ((n=select(sock+1, &conjunto, NULL, NULL, &timeout))<0)
{
```

```
      perror("select");
      exit(0);
}
```

Lectura y escritura.

read.

```
#include <unistd.h>

int read(int fd, void *buf, int lon);
```

La función *read* lee datos del socket especificado por *fd*. Sus parámetros son:

fd	Descriptor del socket creado con anterioridad.
buf	Buffer que contendrá los datos leídos.
lon	Longitud del buffer en bytes, indica además el tamaño máximo en bytes de los datos a leer.

La función devuelve -1 en caso de error y el número de bytes leídos, que pueden ser 0, si tiene éxito la llamada. Generalmente 0 indica el final de los datos en el socket. Ejemplo:

```
#define TAM 100
...
int sock, n;
char buffer[TAM];
...
if ((n=read(sock, buffer, TAM))<0)
{
      perror("read");
      exit(0);
}
```

recv.

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

int recv(int s, void *buf, int lon, int flags);
```

La función *recv* lee datos del socket especificado por *s*. Sus parámetros son:

s	Descriptor del socket creado con anterioridad.
buf	Buffer que contendrá los datos leídos
lon	Longitud del buffer en bytes, indica además el tamaño máximo en bytes de los datos a leer.
flags	Opciones de recepción, generalmente valor 0.

La función devuelve -1 en caso de error y el número de bytes leídos, que pueden ser 0, si tiene éxito la llamada. Generalmente 0 indica el final de los datos en el socket. Ejemplo:

```
#define TAM 100
...
int sock, n;
char buffer[TAM];
...
if ((n=recv(sock, buffer, TAM, 0))<0)
{
     perror("recv");
     exit(0);
}
```

recvfrom.

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

int recvfrom(int s, void *buf, int lon, int flags,
         struct sockaddr *desde, int *londesde);
```

La función *recvfrom* lee datos del socket especificado por *s*. Sus argumentos son:

s	Descriptor del socket creado con anterioridad.
buf	Buffer que contendrá los datos leídos
lon	Longitud del buffer en bytes, indica además el tamaño máximo en bytes de los datos a leer.
flags	Opciones de recepción, generalmente valor 0.
desde	Puntero a la estructura de datos que contendrá la dirección y el puerto desde el que se han recibido los datos leídos.
londesde	Puntero a la longitud de la estructura de datos anterior.

La estructura *sockaddr* no suele ser utilizada, siendo siempre utilizada en su lugar la estructura *sockaddr_in*, explicada con anterioridad.

La función devuelve el valor -1 si error, o el número de bytes leídos si su llamada tiene éxito. Generalmente 0 indica el final de los datos en el socket. Ejemplo:

```
#define TAM 100
...
int sock, n, tam;
struct sockaddr_in s
char buffer[TAM];
...
tam=sizeof(struct sockaddr_in);
if ((n=recvfrom(sock, buffer, TAM, 0, (struct sockaddr *)&s,
                &tam))<0)
{
     perror("recvfrom");
```

```
     exit(0);
}
```

write.

```
#include <unistd.h>

int write(int fd, void *buf, int num);
```

La función *write* escribe hasta *num* bytes de datos al socket especificado por *fd*. Sus parámetros son:

fd	Descriptor del socket creado con anterioridad.
buf	Buffer que contiene los datos a escribir.
num	Número de bytes a escribir en el socket.

La función devuelve -1 en caso de error y el número de bytes realmente escritos si tiene éxito. Es necesario tener en cuenta que la función no tiene porque poder escribir todos los bytes solicitados en una sola llamada. Ejemplo:

```
#define TAM 100
...
int sock;
char buffer[TAM];
...
if (write(sock, buffer, strlen(buffer))==-1)
{
     perror("write");
     exit(0);
}
```

send.

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

int send(int s, const void *buf, int num, int flags);
```

La función *send* escribe hasta *num* bytes de datos al socket especificado por *s*. Sus parámetros son:

s	Descriptor del socket creado con anterioridad.
buf	Buffer que contiene los datos a escribir en el socket.
num	Número de bytes a escribir en el socket.
flags	Opciones de envío, generalmente valor 0.

La función devuelve -1 en caso de error y el número de bytes realmente escritos si tiene éxito. Es necesario tener en cuenta que la función no tiene porque poder escribir todos los bytes solicitados en una sola llamada. Ejemplo:

```
#define TAM 100
```

```
...
int sock;
char buffer[TAM];
...
if (send(sock, buffer, strlen(buffer), 0)==-1)
{
     perror("send");
     exit(0);
}
```

sendto.

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

int sendto(int s, const void *buf, int num, int flags,
          const struct sockaddr *to, int tolen);
```

La función *sendto* escribe hasta *num* bytes de datos mediante el socket especificado por *s*. Sus parámetros son:

s	Descriptor del socket creado con anterioridad.
buf	Buffer que contiene los datos a escribir en el socket.
num	Número de bytes a escribir en el socket.
flags	Opciones de envío, generalmente valor 0.
to	Puntero a la estructura de datos que contiene la dirección IP y el puerto al que se desean escribir los datos.
tolen	Longitud de la estructura de datos anterior.

La estructura *sockaddr* no suele ser utilizada, siendo siempre utilizada en su lugar la estructura *sockaddr_in*, explicada con anterioridad.

La función devuelve -1 en caso de error y el número de bytes realmente escritos si tiene éxito. Es necesario tener en cuenta que la función no tiene porque poder escribir todos los bytes solicitados en una sola llamada. Ejemplo[4]:

```
#define TAM 100
...
int sock;
struct sockaddr_in s;
char buffer[TAM];
...
s.sin_family=PF_INET;
s.sin_port=htons(13);
if (inet_aton(  147.156.1.1  ,&s.sin_addr)==0)
{
     perror("inet_aton");
     exit(0);
}
if (sendto(sock, buffer, strlen(buffer), 0,
          (struct sockaddr *)&s, sizeof(struct sockaddr_in))<0)
```

[4] Las funciones *htons* e *inet_aton* usadas en el ejemplo se explicarán con posterioridad.

```
{
      perror("sendto");
      exit(0);
}
```

Conversión entre formatos de representación de direcciones IP.

inet_aton.

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

int inet_aton(const char *cp,struct in_addr *inp);
```

La función *inet_aton* convierte la dirección de Internet dada por *cp* desde la notación estándar de números y puntos (por ejemplo 147.156.1.1) a la representación binaria en orden de bytes de red y la guarda en la estructura a la que apunta *inp*. Sus parámetros son:

cp	Cadena de caracteres con la dirección Internet a convertir.
inp	Estructura que contendrá la dirección convertida.

La estructura *in_addr* ha sido explicada con anterioridad.

La función devuelve 0 en caso de error y un valor distinto de 0 si la dirección proporcionada es válida. Ejemplo:

```
struct sockaddr_in s;
...
if (inet_aton(  147.156.1.1  , &s.sin_addr)==0)
{
      perror("inet_aton");
      exit(0);
}
```

inet_ntoa.

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

char *inet_ntoa(struct in_addr in);
```

La función *inet_ntoa* convierte la dirección de Internet representada en formato binario en orden de bytes de red a una cadena de caracteres en la notación estándar de números y puntos. Sus parámetros son:

in	Estructura que contiene las dirección a convertir.

La función devuelve un puntero a la cadena de caracteres con la dirección Internet en la notación estándar de números y puntos, devolviendo un puntero a NULL en caso de fallar. El puntero devuelto hace referencia a una variable estática que es sobrescrita en cada llamada a la función. Ejemplo:

```
struct sockaddr_in s;
char *p;
...
if ((p=inet_ntoa(s.sin_addr))==NULL)
{
      perror("inet_ntoa");
      exit(0);
}
```

Conversión entre formatos de representación de datos en el computador y en la red.

Los microprocesadores que incorporan los ordenadores poseen dos formas diferentes de representar los números, primero el byte más significativo y luego el menos significativo (Big-Endian) o al revés (Little-Endian).

Esta diferencia, que dentro de un microprocesador no deja de ser una anécdota, adquiere su importancia en el mundo de las redes, pues si cada ordenador manda los datos como los representa internamente (Big-Endian o Little-Endian), el ordenador que recibe dichos datos puede interpretarlos correctamente o incorrectamente, dependiendo de si su formato de representación coincide con el del emisor o no.

Por ejemplo, si un ordenador indica que desea establecer una conexión con el puerto 80 (servidor web) de otro ordenador, si lo envía en formato Big-Endian los bytes que enviará serán: 0x00 0x50, mientras que en formato Little-Endian enviará los bytes 0x50 0x00. Si el otro ordenador representa los datos en formato diferente, al recibir, por ejemplo, 0x00 0x50 lo traducirá como el puerto 0x50 0x00, esto es 20480, que obviamente no corresponde al servidor web.

Para solventar este problema, se definió un formato conocido como Orden de los Bytes en la Red (Network Byte Order), que establece un formato común para los datos que se envían por la red, como son el número de puerto, etc., de forma que todos los datos, antes de ser enviados a las funciones que manejan las conexiones en la red, deben ser convertidos a este formato de red[5]. A continuación, podemos ver las cuatro funciones que permiten manejar estos datos[6].

htonl.

```
#include <netinet/in.h>

unsigned long int htonl(unsigned long int hostlong);
```

[5] El formato Network Byte Order coincide con el formato Big-Endian.

[6] Aunque algunas funciones se definan como unsigned long int, y su tamaño teóricamente cambie entre los procesadores de 32 y de 64 bits, todas ellas procesan internamente 32 bits y devuelven idéntico valor, independientemente del procesador.

La función *htonl* convierte el entero de 32 bits dado por *hostlong* desde el orden de bytes del hosts al orden de bytes de la red. Ejemplo:

```
unsigned long int red, host;
...
red=htonl(host);
```

htons.

```
#include <netinet/in.h>

unsigned short int htons(unsigned short int hostshort);
```

La función *htons* convierte el entero de 16 bits dado por *hostshort* desde el orden de bytes del hosts al orden de bytes de la red. Ejemplo:

```
unsigned short int red, host;
...
red=htons(host);
```

ntohl.

```
#include <netinet/in.h>

unsigned long int ntohl(unsigned long int netlong);
```

La función *ntohl* convierte el entero de 32 bits dado por *netlong* desde el orden de bytes de la red al orden de bytes del hosts. Ejemplo:

```
unsigned long int host, red;
...
host=ntohl(red);
```

ntohs.

```
#include <netinet/in.h>

unsigned short int ntohs(unsigned short int netshort);
```

La función ntohs convierte el entero de 16 bits dado por *netshort* desde el orden de bytes de la red al orden de bytes del hosts. Ejemplo:

```
unsigned short int host, red;
...
host=ntohs(red);
```