UNIVERSIDAD DE ANTIOQUIA
DEPARTAMENTO DE INGENIERIA ELECTRÓNICA Y TELECOMUNICACIONES
LABORATORIO

PRACTICA 10: Comunicación en red usando conectores o sockets

Objetivo:

- Introducir el manejo de conectores o sockets

INTRODUCCION

Un socket o conector es una interfaz de entrada-salida de datos que permite la intercomunicación entre procesos. Los procesos pueden estar ejecutándose en el mismo o en distintos sistemas, unidos mediante una red.

Los conectores se crean en un dominio de comunicación, igual que un archivo es creado dentro de un sistema de archivos; el dominio de comunicación indica donde se encuentran los procesos que se van a intercomunicar. Si los procesos están en el mismo sistema, el dominio de comunicación será AF_UNIX, si los procesos están en distintos sistemas y se hallan unidos mediante una red TCP/IP, el dominio de comunicación será AF_INET.

Existen otros dominios de comunicación como se verá mas adelante, aunque para el alcance de este documento no se profundizará en estos temas.

Cuando hablamos de comunicación en red (usando AF_INET) mediante conectores (o sockets), estamos haciendo referencia a una interfaz o servicio con la capa de transporte (el nivel 4 del modelo OSI). La división de capas de un sistema es transparente al usuario, que puede trabajar con ellas sin necesidad de conocer sus detalles de implementación.

La interfaz de acceso a la capa de transporte no esta aislada de las capas inferiores, por lo que es necesario conocer algunos detalles de estas como la familia o dominio de la conexión y el tipo de conexión.

- La familia de la conexión engloba conectores que tienen características comunes (protocolos, convenios para formar nombres)

- El tipo de conexión nos indica si el circuito por el que se van a comunicar los procesos es virtual (orientado a la conexión) o datagrama (no orientado a la conexión). En el primer caso se buscan enlaces libres que unan los ordenadores a conectar. Los datagramas por el contrario trabajan con paquetes que pueden seguir rutas distintas, por lo que no realizan conexiones permanentes.

DIRECCIONES DE RED

La forma de construir direcciones depende de los protocolos que se empleen en la capa de transporte y de red, sin embargo, hay llamadas al sistema que necesitan un puntero a una estructura de dirección de conector para trabajar. Esta estructura se define en el fichero de cabecera <sys/socket.h>.

La dirección la contienen 14 bytes. Su significado depende de la familia de conectores que se esté empleando.

MODELO CLIENTE - SERVIDOR

Este modelo es muy empleado para construir aplicaciones en una red.

SERVIDOR: Es un proceso que se esta ejecutando en un nodo de la red y que gestiona el acceso a un determinado recurso.

CLIENTE: Es un proceso que se ejecuta en el mismo o en diferente nodo y que realiza peticiones al servidor. Las peticiones están originadas por la necesidad de acceder al recurso que gestiona el servidor.

El servidor se mantiene a la espera de peticiones, hasta que el cliente realiza una. La cumple y vuelve al estado inicial. Basándonos en esto podemos considerar dos tipos de servidores:

- Interactivos: El servidor atiende a la petición a parte de recogerla. Puede originar tiempos de espera largos si el servidor es lento.

- Concurrentes: El servidor recoge la petición de servicio, pero en lugar de atenderlas crea otros procesos que lo hacen. Esto solo se puede aplicar en sistemas multiprocesos como UNIX/Linux y los sistemas Windows modernos. Con este sistema aumenta la velocidad por lo que es recomendable para las aplicaciones donde los tiempos de servicio son variables.

NETWORK BYTE ORDER

Network byte order y Host byte order son dos formas en las que el sistema puede almacenar los datos en memoria. Está relacionado con el orden en que se almacenan los bytes en la memoria RAM. Si al almacenar un short int (2 bytes) o un long int (4 bytes) en RAM, en la posición más alta se almacena el byte menos significativo, entonces está en network byte order, caso contrario es host byte order.

Esto depende del microprocesador que se esté utilizando, podríamos estar programando en un sistema host byte order o network byte order, pero cuando enviamos los datos por la red deben ir en un orden especificado, sino enviaríamos

todos los datos al revés. Lo mismo sucede cuando recibimos datos de la red, debemos ordenarlos al orden que utiliza nuestro sistema. Debemos cumplir las siguientes reglas :

- Todos los bytes que se transmiten hacia la red, sean números IP o datos, deben estar en network byte order.
- Todos los datos que se reciben de la red, deben convertirse a host byte order.

Para realizar estas conversiones utilizamos las funciones que se describen a continuación.

Herramientas de conversión

htons() -> host to network short - convierte un short int de host byte order a network byte order.

htonl() -> host to network long - convierte un long int de host byte order a network byte order.

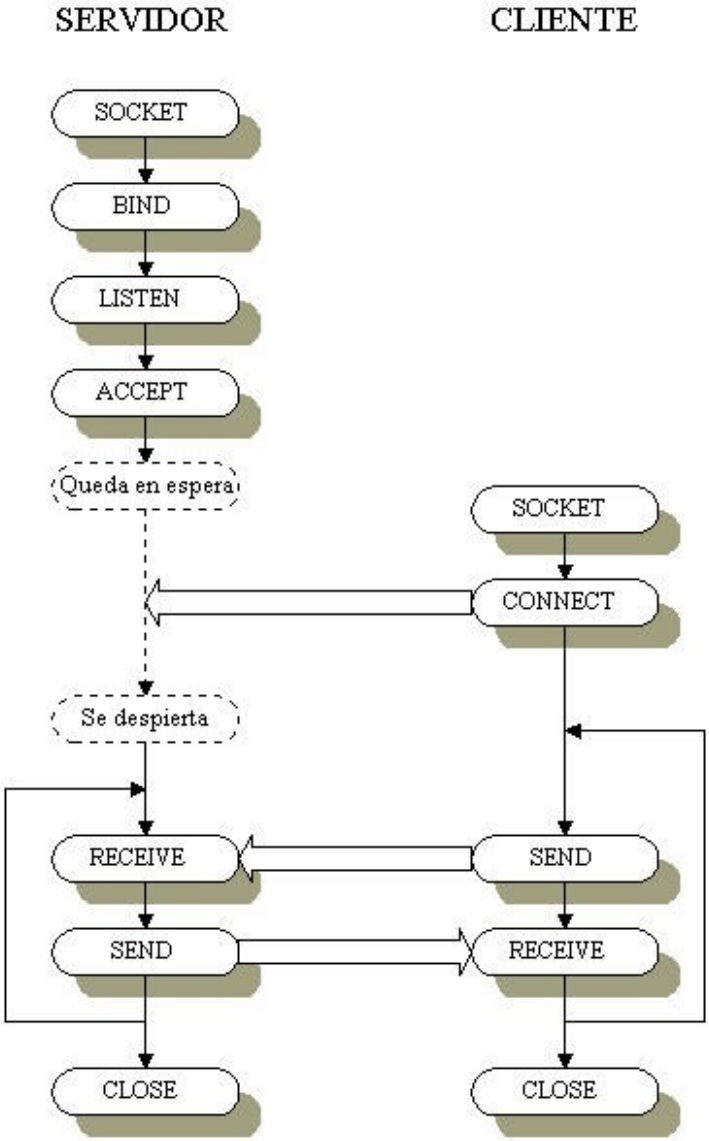
ntohs() -> network to host short - convierte un short int de network byte order a host byte order.

ntohl() -> network to host long - convierte un long int de network byte order a host byte order.

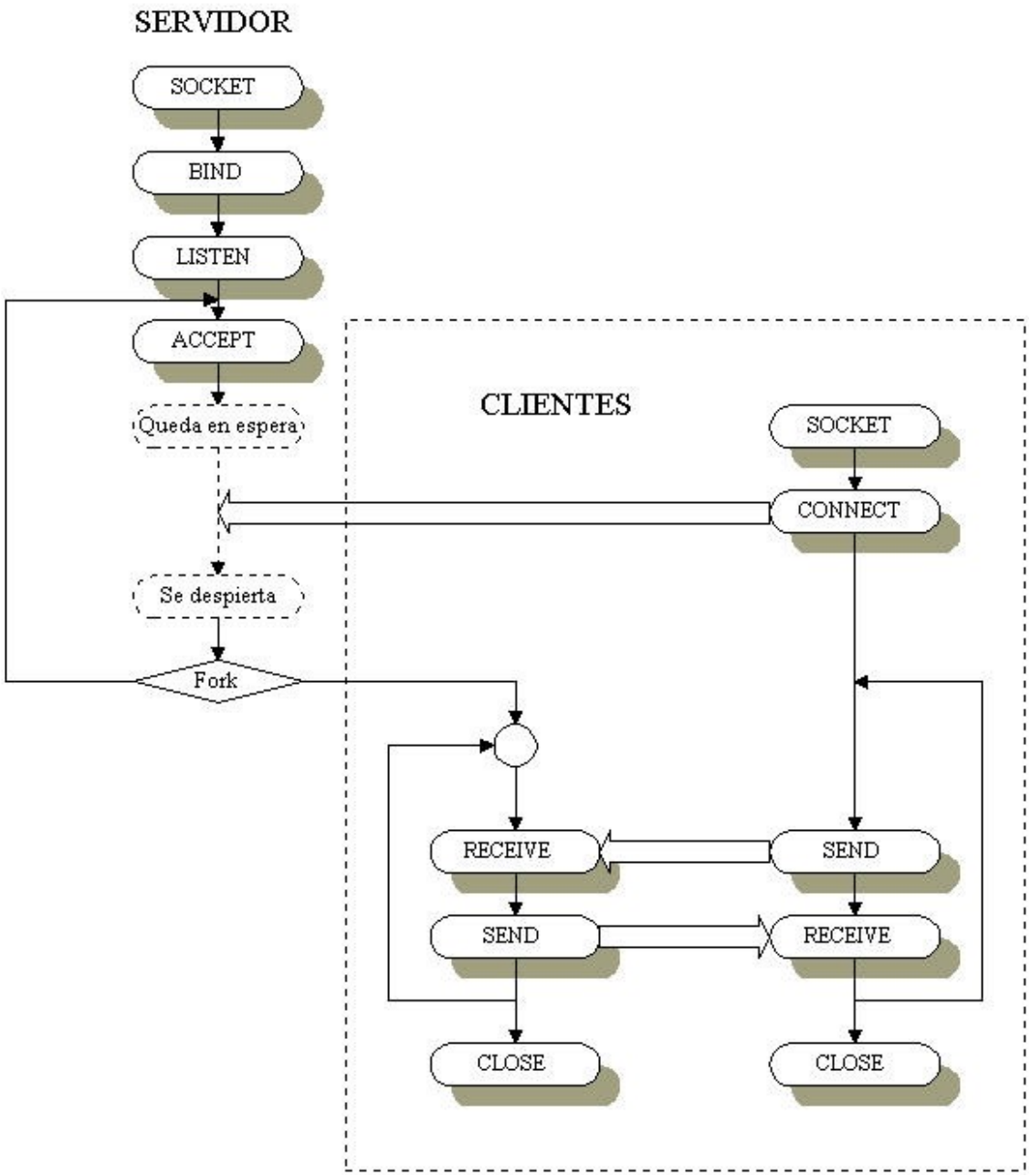
Puede ser que el sistema donde se esté programando almacene los datos en network byte order y no haga falta realizar ninguna conversión, pero si tratamos de compilar el mismo código fuente en otra plataforma host byte order no funcionará. Como conclusión, para que nuestro código fuente sea portable se debe utilizar siempre las funciones de conversión.

Para más información : man 3 byteorder

ESQUEMA GENERAL DE UN SERVIDOR Y DE UN CLIENTE INTERACTIVO



ESQUEMA GENERAL DE UN SERVIDOR Y DE UN CLIENTE CONCURRENTE



LLAMADAS PARA EL MANEJO DE CONECTORES

Los siguientes son los pasos a grosso modo que debe seguir un servidor y un cliente para poder realizar las comunicaciones.

Para el servidor:

Socket: Apertura del canal.

Bind: Publicidad de la dirección.

Listen: Disposición para aceptar conexiones.

Accept: Aceptar una conexión. Bloquea el proceso hasta que se recibe una petición de conexión.

Para el cliente:

Socket: Apertura del canal.

Connect: Petición de conexión.

Close: Cierra el canal.

Para ambos:

Read, Send: Lectura de la petición de servicio para el servidor, y lectura de la respuesta para el cliente.

Write, Receive: Envío de los datos al cliente por parte del servidor y petición de servicio del cliente.

DECLARACION DE UN NUEVO CONCERTOR (SOCKET)

La llamada par abrir un canal bidireccional de comunicaciones es socket.

Esta crea un punto terminal para conectarse a un canal y devuelve un descriptor (similar al descriptor en los archivos).

El descriptor del conector devuelto se usará en llamadas posteriores a funciones de la interfaz. El parámetro "af" determina que familia de direcciones o conectores vamos a emplear.

Las principales familias son:

AF_UNÍX: Comunica procesos que se ejecutan en una misma máquina.

AF_INET: Son los protocolos de internet. Utiliza algunos como TCP o UDP.

El parámetro "type" indica la semántica de la comunicación para el conector y puede tomar los valores:

SOCK_STREAM: Orientado a la conexión. Es un circuito virtual.

SOCK_DGRAM: Protocolo de tipo datagrama.

SOCK_RAW: Sólo puede ser utilizado por usuarios con permisos de superusuario, ya que facilita el acceso directo a los protocolos internos de la red

SOCK_SEQPACKET Y SOCK_RDM: Protocolos no orientados a conexión que proporcionan un envío fiable y secuencial de datagramas. El segundo aún no esta implementado. Sí para un conector hubiese más de un protocolo se especificaría mediante el argumento "protocolo".

En resumen, l**os sockets se crean llamando a la función socket(), esta función retorna un descritpor de socket, que es tipo int.**

Si hubo algún error, socket() retorna -1 y la variable global errno se establece con un valor que indica el error que se produjo (ver man 3 perror).

sockfd = socket (int dominio, int tipo, int protocolo);

sockfd	Es el descriptor de socket devuelto. Luego se utilizará para conectarse, recibir conexiones, enviar y recibir datos, etc.
dominio	Dominio donde se realiza la conexión. Para este tutor siempre será AF_INET.
tipo	Podrá ser SOCK_STREAM o SOCK_DGRAM o SOCK_RAW.
protocolo	0 (cero, selecciona el protocolo más apropiado).

Veamos un ejemplo:

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

Si se especifica al protocolo como cero, el sistema selecciona el protocolo apropiado de uno de los protocolos disponibles, dependiendo del tipo de socket requerido.

Para más información: man 2 socket

Se puede especificar el número de protocolo o también seleccionar el protocolo por su nombre utilizando la función getprotobyname(), que retorna una estructura tipo protent con los siguientes datos:

char *p_name; /* Nombre oficial del protocolo */

char **p_aliases; /* lista de alias. */

int p_proto; /* número de protocolo */

Esta función lee el archivo /etc/protocols para completar la estructura.

```
pp=getprotobyname("tcp");

s=socket(AF_INET, SOCK_STREAM, pp->p_proto);
```

Para más información : man 3 getprotoent

NOMBRE DE UN CONECTOR (BIND)

bind () se utiliza para atar el socket a un puerto y una dirección IP, es decir la dirección IP y número de puerto del host local por donde escuchará, al especificar una IP del host local le estamos diciendo por cual interfaz física escuchará (el sistema puede tener varias interfaces ethernet, ppp, etc).

Es necesario llamar a bind() cuando se está programando un servidor.

Cuando se está programando un cliente no se utiliza esta función, el kernel le asignará al socket la dirección IP y número de puerto disponible al llamar a ciertas funciones, por ejemplo cuando llamamos a connect() para conectarnos con un sistema remoto.

En el servidor es necesario llamar a bind() debido a que el número de puerto debe ser conocido (y no aleatorio, como lo es en los clientes) para que los clientes puedan conectarse. Por ejemplo si estamos programando un servidor de paginas web debemos llamar a bind() para asignarle al socket el puerto 80.

```
int bind(int sockfd, struct sockaddr *my_addr, int addrlen)
```

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

my_addr : Es un puntero a una estuctura sockaddr que contiene la IP del host local y el número de puerto que se va a asignar al socket. (Mas abajo se detalla).

addrlen : Debe ser establecido al tamaño de la estuctura sockaddr. sizeof(struct sockaddr).

Ejemplo :

...

```
struct sockaddr_in sin;

...

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

NOTA: Es importante conocer como está formada la estructura sockaddr_in y como establecemos su valor.

```
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 primer estructura, sockaddr, almacena la dirección de protocolo para muchos

tipos de protocolos.

sa_family puede ser AF_INET, AF_UNIX u otros dominios, para nuestro tutorial solo será AF_INET.

sa_data contiene la dirección IP y número de puerto asignado al socket.

Se creó la esctuctura sockaddr_in para el caso de internet, para poder referenciar los elementos de forma más fácil.

Los punteros a la estructura sockaddr_in deben ser precedidos con un cast tipo *struct sockaddr antes de pasarlos como parámetros a funciones.

Notas sobre sockaddr_in:

- sin_family sera AF_INET
- sin_port (número de puerto) y sin_addr (dirección IP) deberán estar en network byte order, osea habrá que usar htons().
- sin_family no debe convertirse a network byte order porque es solo usado por el kernel y no es enviado por la red.
- sin_zero se utiliza para rellenar la estructura a la longitud de sockaddr, debe estar inicializada a cero con la función bzero(). Ver la página del manual

OTRAS FUNCIONES UTILES

- **inet_addr()**

Convierte una dirección IP en notación números y puntos, en un unsigned long, retorna la dirección en network byte order. Retorna -1 si hubo error.

Ejemplo:

```
struct sockaddr_in ina;

....

ina.sin_addr.s_addr=inet_addr("192.168.1.1");
```

- i**net_ntoa()**

Realiza la conversión inversa, convierte una dirección IP en unsigned long en network byte order, a un string en números y puntos.

Ejemplo:

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

inet_ntoa() retorna un puntero a un array de caracteres, que se encuentra almacenado estáticamente dentro de inet_ntoa(). Cada vez que se llama a inet_ntoa() se sobreescribe la última dirección IP.

Ejemplo:

```
char *addr1, *addr2;

addr1=inet_ntoa(ina1.sin_addr)       /* Supongamos que vale 200.41.32.127 */

addr2=inet_ntoa(ina2.sin_addr)      /* Supongamos que vale 132.241.5.10 */

printf("direccion 1: \n", addr1);

printf("direccion 2: \n", addr2);
```

Esto imprimirá:

```
 direccion 1: 132.241.5.10

 direccion 2: 132.241.5.10
```

Se ve que la primer dirección IP es sobreescrita en la segunda llamada a inet_ntoa() y se pierde su valor.

Para que esto no suceda, luego de la primera llamada a inet_ntoa() se debe usar strcpy() para guardar la primera dirección IP y luego llamar por segunda vez a inet_ntoa(), de esta manera no se pierde la primera dirección IP.

Ver : man 3 inet

Asignación de valores a una variable tipo sockaddr_in

Debemos asignarle valores a una variable tipo sockaddr_in antes de llamar a las función bind().

Veamos un ejemplo:

...

..

struct sockaddr_in my_addr;

.....

my_addr.sin_family = AF_INET;

my_addr.sin port = htons (3490); // Numero de puerto por donde escuchara el servidor.
my_addr.sin_addr.s_addr = inet_addr ("132.241.5.10"); // IP de la interface por donde escuchara el servidor.

bzero (&(my_addr.sin_zero), 8); // Relleno con ceros.

Notas :

- Si asignamos el valor cero a sin_port, el sistema nos dará automáticamente un puerto disponible.

```
my_addr.sin_port=0;
```

- Podemos automatizar la asignación de la IP, si ponemos el valor INADDR_ANY a s_addr, el sistema le asignará la dirección IP local. Recordar que el programa puede ejecutarse en distintas PC's con distintas IP's

```
my_addr.sin_addr.s_addr = htonl (INADDR_ANY);
```

- Las variables **my_addr.sin_por**t y **my_addr.sin_addr.s_addr** deben estar en network byte order, son valores que viajan por la red, pero **my_addr.sin_family** no porque solo es utilizado por el kernel para saber que tipo de dirección contiene la estructura.

DISPONIBILIDAD PARA RECIBIR PETICIONES DE SERVICIO (LISTEN)

El servidor indica que esta disponible para recibir peticiones con la llamada listen. El tipo de conector a de ser SOCK_STREAM y esta llamada suele ejecutarse en el proceso servidor tras socket y bind.

Se llama desde el servidor, habilita el socket para que pueda recibir conexiones. Solo se aplica a sockets tipo SOCK_STREAM.

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

sockfd : Es el descriptor de socket devuelto por la función socket() que será utilizado para recibir conexiones.

backlog : Es el número máximo de conexiones en la cola de entrada de conexiones. Las conexiones entrantes quedan en estado de espera en esta cola hasta que se aceptan (accept ()).

En los servidores interactivos mientras se esta atendiendo a un cliente pueden llegar peticiones de otros, por lo que es importante la cola de conexiones que habilita el listen. Si esta llamada funciona correctamente emite el valor 0 y en caso contrario el -1.

PETICIÓN DE CONEXIÓN (CONNECT)

La llamada connect es necesaria para establecer una conexión.

Para conectores SOCK_DGRAM: Connect especifica la dirección del conector remoto pero no se conecta con él. Además solo se podrán recibir mensajes procedentes de la dirección especificada.

Para conectores SOCK_STREAM: Connect intenta contactar con el ordenador remoto con objeto de realizar una conexión entre el conector remoto y el conector local. La llamada permanece bloqueada hasta que la conexión se completa.

Es utilizada por el cliente para conectarse.

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

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

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

addrlen : Debe ser inicializado al tamaño de struct sockaddr (sizeof (struct sockaddr)).

Como en anteriores casos si la llamada se ejecuta correctamente devuelve 0 y si no -1.

ACEPTACIÓN DE UNA CONEXIÓN (ACCEPT)

La llamada accept nos sirve para que los procesos descriptores puedan leer peticiones de servicio.

Se usa con conectores orientados a conexión. Extrae la primera petición de conexión que hay en cola, creada con una llamada previa la listen. Luego crea un nuevo conector con las mismas propiedades que sfd y reserva un nuevo descriptor de fichero (nsfd) para él.

Accept permanece bloqueada hasta que reciba una nueva petición de conexión cuando no la tiene.

La llamada select puede usarse para ver si el conector tiene pendiente alguna petición de conexión.

Se utiliza en el servidor, con un socket habilitado para recibir conexiones (listen()). Esta función retorna un nuevo descriptor de socket al recibir la conexión del cliente en el puerto configurado. La llamada a accept() no retornará hasta que se produce una conexión o es interrumpida por una señal.

int accept (int sockfd, void *addr, int *addrlen)

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

addr : Puntero a una estructura sockadd_in. Aquí se almacenará información de la conexión entrante. Se utiliza para determinar que host está llamando y desde qué número de puerto.

addrlen : Debe ser establecido al tamaño de la estuctura sockaddr. sizeof(struct sockaddr).

accept() no escribirá más de addrlen bytes en addr . Si escribe menos bytes, modifica el valor de addrlen a la cantidad de bytes escritos.

Ejemplo:

```
...

int sockfd, new_sockfd;

struct sockaddr_in  my_addr;

struct sockaddr_in  remote_addr;

int addrlen;

...

// Creo el socket.

sockfd = socket (AF_INET, SOCK_STREAM, 0 );

...

// Se le asigna un número de puerto al socket por donde el servidor escuchará.
```

```
// Antes de llamar a bind()  se debe asignar valores a 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);

...
```

Si la llamada funciona correctamente devolverá un numero entero no negativo que se debe interpretar como un descriptor del conector aceptado, en caso de error devolverá el valor -1.

LECTURA O RECEPCIÓN DE MENSAJES DE UN CONECTOR

Cuando el canal de comunicaciones está iniciado y el servidor y el cliente disponen de un conector con el canal, contamos con 5 llamadas al sistema para leer datos o mensajes de un conector. Estas llamadas son read, readv, recv, recvfrom, y recvmsg.

El funcionamiento de la llamada read tiene el mismo interfaz que para el manejo de ficheros. Para conectores su comportamiento es igual exceptuando que obviamente el descriptor de ficheros en realidad un descriptor de conector.

Las otras cuatro llamadas son variaciones de read que sólo funcionan con conectores.

ESCRITURA O ENVIO DE MENSAJES A UN CONECTOR.

Como ocurría para el read, tenemos 5 llamadas para escribir datos en un conector. Write, writev, send, sedto, sedmsg.

La llamada write se comporta también como cuando se usa con ficheros con la salvedad de que el descriptor de ficheros es en realidad un descriptor de conector.

Writev es una generalización de write y se puede utilizar para fichero y para conector. Las otras tres devuelven el total de bytes escritos en el conector.

En resumen, después de establecer la conexión, se puede comenzar con la transferencia de datos.

send() y recv() son idénticas a write() y read(), excepto que se agrega un parámetro flags.

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

sockfd :	Descriptor socket por donde se enviarán los datos.
msg :	Puntero a los datos a ser enviados.
len :	Longitud de los datos en bytes.
flags :	Leer: man 2 send

send() retorna la cantidad de datos enviados, la cual podrá ser menor que la cantidad de datos que se escribieron en el buffer para enviar . send() enviará la máxima cantidad de datos que pueda manejar y retorna la cantidad de datos enviados, es responsabilidad del programador comparar la cantidad de datos enviados con len y si no se enviaron todos los datos, enviarlos en la próxima llamada a send().

recv (int sockfd, void *buf, int len, unsigned int flags)

sockfd :	Descriptor socket por donde se recibirán los datos.
buf :	Puntero a un buffer donde se almacenarán los datos recibidos.
len :	Longitud del buffer buf.
flags :	Ver : man 2 recv

Si no hay datos a recibir en el socket , la llamada a recv() no retorna (bloquea) hasta que llegan datos, se puede establecer al socket como no bloqueante (ver: man 2 fcntl) de manera que cuando no hay datos para recibir la función retorna -1 y establece la variable errno=EWOULDBLOCK. recv() retorna el número de bytes recibidos.

- **sendto() y recvfrom()**

Funciones para realizar transferencia de datos sobre sockets datagram.

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

sockfd : Descriptor socket por donde se enviarán los datos.

msg : Puntero a los datos a ser enviados.

len : Longitud de los datos en bytes.

flags : Leer : man 2 sendto

to : Puntero a una estructura sockaddr que contiene la dirección IP y número de puerto destino.

tolen : Debe ser inicializado al tamaño de struct sockaddr (sizeof (struct sockaddr)).

sendto() retorna el número de bytes enviados, el cual puede ser menor que len, igual que en send().

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

sockfd : Descriptor socket por donde se recibirán los datos.

buf: Puntero a un buffer donde se almacenarán los datos recibidos.

len: Longitud del buffer buf.

flags: Ver la página del manual de recv().

from: Puntero a una estructura sockaddr que contiene la dirección IP y número de puerto del host origen de los datos.

fromlen: Debe ser inicializado al tamaño de struct sockaddr (sizeof (struct sockaddr)).

Si no hay datos a recibir en el socket , la llamada a recvfrom() no retorna (bloquea) hasta que llegan datos, se puede establecer al socket como no bloqueante (ver: man 2 fcntl) de manera que cuando no hay datos para recibir la función retorna -1 y establece la variable errno=EWOULDBLOCK. Más adelante se hablará de esto.

recvfrom() retorna el numero de bytes recibidos, igual que recv().

- **close () y shutdown ().**

Finalizan la conexión del descriptor de socket.

close (sockfd)

Después de utilizar close, el socket queda desabilitado para realizar lecturas o escrituras.

shutdown (sockfd, int how)

Permite deshabilitar la comunicación en una determinada dirección o en ambas direcciones.

how: Especifica en qué dirección se deshabilita la comunicación. 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 close ().

- **getpeername(), gethostname(), gethostbyname().**

int getpeername (int sockfd, struct sockaddr *addr, int *addrlen)

Nos dice quién está conectado en el otro extremo de un socket stream.

sockfd:	Descriptor del socket stream conectado.
addr:	Puntero a una estructura sockaddr que almacenará la dirección del otro extremo de la conexión.
addrlen :	Puntero a un int que contiene la longuitud de sockaddr (sizeof (sockaddr)).

int gethostname (char *hostname, size_t size)

Retorna el nombre del sistema donde esta ejecutándose el programa.

El nombre puede ser utilizado por gethostbyname() para determinar la direccion IP del host local.

hostname: Puntero a un array de caracteres que contendrá el nombre del host.

size: Longitud en bytes del array hostname.

gethostname () retorna cero cuando se ejecuta con éxito.

struct hostent *gethostbyname (const char *name)

Se utiliza para convertir un nombre de un host a su dirección IP, osea utiliza el servidor de nombres.

Ejemplo: Supongamos que estamos programando un cliente telnet.

$ telnet algun.sitio.com

La aplicación cliente primero debe trasladarar el nombre del sitio (algun.sitio.com) a conectarse a su dirección IP, para luego poder realizar todos los pasos de conexión anteriormente descritos.

Retorna un puntero a una estructura hostent, que está 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]
```

Descripción de los campos de la estructura hostent:

h_name : Nombre oficial del host.

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 el host.
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 número de error, en vez de la variable errno (ver man herror).

BLOQUEO

Suponga que se crea un socket y se conecta a un servidor a un servidor, sabemos que el socket nos provee una comunicación bidireccional, podemos enviar y recibir datos simultáneamente.

Llamamos a la función recv() para recibir datos, pero el servidor en ese momento no tiene nada para enviarnos, entonces la función recv() no retorna. Justo en ese momento queremos enviar datos hacia el servidor, pero no podemos porque la función recv() no retornó y nos bloqueó el programa.

Debemos encontrar alguna forma para que la función recv() retorne aunque el servidor no envíe nada.

Esto se realiza estableciendo al socket como no bloqueante.

Cuando creamos un socket, se establece como bloqueante, al llamar a ciertas funciones como accept(), recv() , recvfrom(), etc, se bloquea el programa.
Para establecer al socket como no bloqueante utilizamos la función fcntl() de la siguiente manera :

```
int fcntl ( int sockfd, int cmd, long arg);
```

sockfd	Descriptor de socket sobre el cual se va a realizar alguna operación.
cmd	Determina el comando que se va a aplicar, para nuestro caso usaremos el comando F_SETFL, el cual establece los flag del

descriptor al valor especificado en **arg**.

arg	Argumentos que necesita el comando, para establecer el socket como no bloqueante sera O_NONBLOCK.

Si se produce un error, retorna -1 y se establece errno especificando el error.

Leer la página del manual de fcntl().

Ejemplo:

```
#include

#include

...

sockfd=socket(AF_INET, SOCK_STREAM, 0);

fcntl (sockfd, F_SETFL, O_NONBLOCK);

...
```

Una vez establecido el socket como no bloqueante, se puede llamar a la funciones bloqueantes como recv() para recibir datos, si no hay datos disponibles recv() devuelve -1 y establece errno=EWOULDBLOCK.

Se puede ir consultando (polling) el socket para saber si hay datos disponibles, pero esta no es una solución muy buena, se consume tiempo de CPU consultando al socket si tiene datos, existe una solución mas elegante.

FUNCION SELECT

Nos permite monitorear un conjunto de descriptores de sockets y nos avisa cuales tienen datos para leer, cuáles están listos para escribir, y cuáles producieron excepciones.

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

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

Monitorea 3 conjuntos distintos de sockets, aquellos agregados al conjunto readfds los monitorea para ver si hay caracteres disponibles para leer, al

conjunto writefds los monitorea para ver si están listos para ser escritos y al conjunto exceptfds los monitorea para ver si se producen excepciones. Los conjuntos de descriptores son tipo fd_set.

Se proveen 4 macros para manejar los conjuntos de descriptores :

FD_ZERO (fd_set *set)	limpia un conjunto de descriptores.
FD_SET (int fd, fd_set *set)	agrega fd a un conjunto.
FD_CLR (int fd, fd_set *set)	borra fd de un conjunto.
FD_ISSET (int fd, fd_set *set)	Verifica si fd está dentro de un conjunto. Se utiliza luego del retorno de select() para verificar cual descriptor cambio su estado.

Aquellos conjuntos que no tienen descriptores, se especifican con NULL.

n	Es el número de descriptor más alto de cualquiera de los 3 conjuntos, más uno.
timeout	Select puede retornar por dos causas, se produce algún cambio en un descriptor o paso más del tiempo especificado en timeout sin producirse cambios. Si se establece timeout a cero se retorna inmediatamente, si establecemos timeout a NULL se monitorea hasta que se produce algún cambio en los conjuntos, osea puede bloquear. Cuando retorna, timeout indica el tiempo remanente.

Cuando retorna select(), modifica los conjuntos de descriptores reflejando cuál de los descriptores está listo para leer, cuáles para escribir y cuáles causaron excepciones.

Si se produce un error, retorna -1 y se establece errno con el número de error.

Veamos la estructura timeval :

```
struct timeval

{

int tv_sec;             /*  segundos  */

int tv_usec;            /*  micro segundos  */

};
```

Ejemplo :
Monitoreamos la entrada estandard (descriptor 0)

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

#define STDIN  0         /* descriptor para la entrada estandard */

main()

{

      struct timeval timeout;

      fd_set readfds;

      timeout.tv_sec = 2;

      timeout.tv_usec = 500000 ;

      FD_ZERO ( &readfds );

      FD_SET ( STDIN, &readfds );

      select ( STDIN+1, &readfds, NULL, NULL, &timeout );

      if ( FD_ISSET ( STDIN, &readfds ) )

            printf (" Se oprimió una tecla\n");

      else

            printf(" se venció el tiempo\n");

}
```

CIERRE DEL CANAL (CLOSE Y SHUTDOWN)

Para desconectar un proceso de un conector podemos utilizar close. Esta llamada cierra el conector en ambos sentidos.

Finalizan la conexión del descriptor de socket.

```
close ( sockfd)
```

Después de utilizar close, el socket queda desabilitado para realizar lecturas o escrituras.

```
shutdown (sockfd, int how)
```

Permite deshabilitar la comunicación en una determinada dirección o en ambas direcciones.

how : Especifica en qué dirección se deshabilita la comunicación. 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 close ().

BIBLIOGRAFIA

Excelente tutorial (leerlo si hay dudas):

Http://www.beej.us/guide/bgnet/output/html/multipage/index.html

Otro buen tutorial:

http://www.arrakis.es/~dmrq/beej/intro.html

De donde ha sido sacado este documento:

UNIX PROGRAMACION AVANZADA, EDITORIAL RA-MA

http://www.starlinux.net/staticpages/index.php?page=20020720164837437