**Introducción a la programación de sockets en C**

Based on Beej’s Guide to Networking Programming

Cavasin Nicolas

ncavasin97@gmail.com

Version 1.0

16 de Septiembre, 2018.

**Contenidos:**

[Historia del socket: 4](#_Toc525121066)

[¿Qué es un socket? 4](#_Toc525121067)

[Tipos de sockets: 4](#_Toc525121068)

[Byte order 5](#_Toc525121069)

[Estructuras involucradas 6](#_Toc525121070)

[1. socket descriptor: 6](#_Toc525121071)

[2. addrinfo: 6](#_Toc525121072)

[3. sockaddr: 6](#_Toc525121073)

[4. sockaddr\_in y sockaddr\_in6: 6](#_Toc525121074)

[5. in\_addr y in\_addr6: 7](#_Toc525121075)

[6. sockaddr\_storage: 7](#_Toc525121076)

[Manejo de direcciones IP 8](#_Toc525121077)

[1. inet\_pton(): 8](#_Toc525121078)

[2. inet\_ntop(): 8](#_Toc525121079)

[Llamadas al sistema 9](#_Toc525121080)

[1. getaddrinfo(): 9](#_Toc525121081)

[2. socket(): 11](#_Toc525121082)

[3. bind(): 11](#_Toc525121083)

[4. connect(): 12](#_Toc525121084)

[5. listen(): 13](#_Toc525121085)

[6. accept(): 13](#_Toc525121086)

[7. send(): 14](#_Toc525121087)

[8. recv(): 15](#_Toc525121088)

[9. close(): 15](#_Toc525121089)

[10. shutdown(): 15](#_Toc525121090)

[Bloqueo 16](#_Toc525121091)

[Portabilidad entre IPv4 e IPv6 17](#_Toc525121092)

[Ejemplo de servidor TCP 18](#_Toc525121093)

[Ejemplo de cliente TCP 20](#_Toc525121094)

[Referencias: 22](#_Toc525121095)

*Página intencionalmente en blanco*

# Historia del socket:

1. Los sockets fueron desarrollados a principios de los ’80 por la Universidad de Berkeley.
2. Son la base de la comunicación del Stack TCP/IP.
3. Desarrollados sobre Unix.
4. Basados en el modelo cliente-servidor.

# ¿Qué es un socket?

1. Los sockets permiten la comunicación entre procesos remotos, es decir, entre procesos que residen en diferentes computadoras.
2. Un socket, al igual que todo en Unix, *es un archivo*.
3. Sin embargo, y ahí reside la magia, el archivo puede ser una red de conexión (TCP/IP), un FIFO, un pipe, una terminal, etc.
4. Al ser un archivo, se pueden crear, escribir, leer y cerrar.

# Tipos de sockets:

Dependiendo de la finalidad existen tres tipos de socket:

1. Stream sockets:
   1. Orientados a la conexión ---> TCP.
   2. Telnet, HTTP.
2. Datagram sockets:
   1. No orientados a la conexión ---> UDP.
   2. TFTP, DHCP, streaming audio/video, juegos multijugador.
3. Raw sockets.

# Byte order

Existen dos formas de almacenar los bytes en las computadoras:

1. Big-Endian: el número HEX b34f se almacena en dos bytes secuenciales respetando su orden, primero b3 y luego 4f.
2. Little-Endian: el número HEX b34f se almacena también en dos bytes secuenciales pero en orden inverso, es decir, primero 4f y luego b3.

El modo de almacenamiento utilizado por cada computadora es referenciado como **Host Byte Order** y puede ser cualquiera de los dependiendo del fabricante del procesador.

Sin embargo, para la comunicación interprocesos sobre la red se ha tomado como default la utilización Big-Endian la cual es referenciada como **Network Byte Order**.

Por esto, cada vez que se desee enviar datos a través de la red o se reciban datos de ella, se debe asumir que el **Host Byte Order** no es el mismo que el **Network Byte Order** e invocar a una función para que realice la conversión en caso de ser necesario.

Existen dos tipos de números que se pueden convertir, short (2 bytes) y long (4 bytes) incluyendo las variaciones sin signo.

Las funciones a utilizar son:

1. htons() ---> **H**ost **TO** **N**etwork **S**hort.
2. htonl() ---> **H**ost **TO N**etwork **L**ong.
3. ntohs() ---> **N**etwork **TO** **H**ost **S**hort.
4. ntohl() ---> **N**etwork **TO** **H**ost **L**ong.

Básicamente lo que se quiere hacer es invocar **hton*x*()** para convertir los datos a **Network Byte Order** antes de enviarlos hacia la red e invocar **ntoh*x*()** para convertir los datos a **Host Byte Order** luego de recibirlos desde la red. De esta forma se logra una independencia del “Endianismo” utilizado por cada proceso que se comunica sobre Internet.

# Estructuras involucradas

A continuación se presentan todas las estructuras utilizadas por los sockets:

1. **socket descriptor:**
2. Es un *int* regular.

int socket\_desc;

1. **addrinfo:**
   * 1. Utilizada para preparar las estructuras de dirección del socket.
     2. Primero le inicializaremos algunos valores y luego se invoca a **getaddrinfo()** para que la complete.
     3. Devuelve un puntero a una lista enlazada de structs addrinfo que contiene valores que serán utilizados más adelante.

struct addrinfo {

int ai\_flags; // AI\_PASSIVE, AI\_CANONNAME, etc.

int ai\_family; // AF\_INET, AF\_INET6, AF\_UNSPEC

int ai\_socktype; // SOCK\_STREAM, SOCK\_DGRAM

int ai\_protocol; // use 0 for "any"

size\_t ai\_addrlen; // size of ai\_addr in bytes

struct sockaddr \*ai\_addr; // struct sockaddr\_in or \_in6

char \*ai\_canonname; // full canonical hostname

struct addrinfo \*ai\_next; // linked list, next node

};

1. **sockaddr:**
2. Almacena información de la dirección del socket.

struct sockaddr {

unsigned short sa\_family; // address family, AF\_xxx

char sa\_data[14]; // 14 bytes of protocol address

};

1. sa\_family ---> AF\_INET (IPv4) o AF\_INET6 (IPv6).
2. sa\_data ---> contiene la combinación IP destino + puerto del socket.
3. **sockaddr\_in** y **sockaddr\_in6:**
4. Para referenciar + fácil los componentes de la dirección del socket almacenados en sockaddr (puerto e IP), se crearon dos estructuras paralelas a sockaddr. Una para IPv4 y otra para IPv6.
5. Un puntero al struct sockaddr\_in puede ser casteado para que apunte a un struct sockaddr y vice-versa gracias a que ambos tienen el mismo tamaño en bytes.
6. Por ende, a pesar de que **connect()** acepte solo un struct sockaddr\*, se puede utilizar un sockaddr\_in y castearlo antes de invocar dicha función.

struct sockaddr\_in {

short int sin\_family; // Address family, AF\_INET

unsigned short int sin\_port; // Port number

struct in\_addr sin\_addr; // Internet address

unsigned char sin\_zero[8]; // Padding to struct sockaddr

};

struct sockaddr\_in6 {

u\_int16\_t sin6\_family; // address family, AF\_INET6

u\_int16\_t sin6\_port; // port number, Network Byte Order

u\_int32\_t sin6\_flowinfo; // IPv6 flow information

struct in6\_addr sin6\_addr; // IPv6 address

u\_int32\_t sin6\_scope\_id; // Scope ID

};

1. sin\_family <==> sa\_family de sockaddr.
2. sin\_port debe estar en **Network Byte Order** usando **htons()** (pues es un short sin signo).
3. sin\_zero es para padding y debe ser inicializado con **memset()**.
4. **in\_addr** y **in\_addr6:**
5. Mantenida solo por motivos históricos.
6. Es una unión que permite acceder a la direccion IPv4 almacenada en **Network Byte Order**.

// Internet address (a structure for historical reasons)

struct in\_addr {

uint32\_t s\_addr; // that's a 32-bit int (4 bytes)

};

// Internet address (a structure for historical reasons)

struct in6\_addr {

unsigned char s6\_addr[16]; // IPv6 address

};

1. Ejemplo de acceso en IPv4:

//variable ina as a struct sockaddr\_in:

struct sockaddr\_in ina;

//accesing the value of the IP:

IPv4 = ina.sin\_addr.s\_addr;

1. **sockaddr\_storage:**
2. Es otra estructura paralela.
3. Diseñada para almacenar tanto estructuras IPv4 como IPv6.
4. Utilizada para aceptar conexiones (ya que no se sabe si van a ser IPv4 o IPv6) y luego castear hacia el tipo de sockaddr\_in correspondiente.

struct sockaddr\_storage {

sa\_family\_t ss\_family; // address family

// all this is padding, implementation specific, ignore it:

char \_\_ss\_pad1[\_SS\_PAD1SIZE];

int64\_t \_\_ss\_align;

char \_\_ss\_pad2[\_SS\_PAD2SIZE];

};

1. ss\_family ---> indica el tipo de familia (AF\_INET o AF\_INET6) y debe ser utilizado para determinar el tipo de casteo.

# Manejo de direcciones IP

Existen dos funciones que facilitan la manipulación de direcciones IP:

1. **inet\_pton():**
2. **P**resentation **TO** **N**etwork.
3. Convierte un string IP con notación punto-decimal a binario para poder ser almacenado un struct in\_addr o in\_addr6 dependiendo del tipo de familia especificado en su invocación.

struct sockaddr\_in sa; // IPv4

struct sockaddr\_in6 sa6; // IPv6

//no error-checking is done for simplificaton

inet\_pton(AF\_INET, "10.12.110.57", &(sa.sin\_addr)); // IPv4

inet\_pton(AF\_INET6, "2001:db8:63b3:1::3490", &(sa6.sin6\_addr)); // IPv6

1. Se la invoca indicando:
   1. Familia de dirección.
   2. Puntero char de string a convertir.
   3. Dir. de memoria donde se almacenara la IP (ip\_dest)
2. La IP es convertida a un struct in\_addr y copiada en ip\_dest.
3. ip\_dest debe tener el mismo tamaño que el struct in\_addr (4 bytes).
4. Retorna -1 en error, 0 si la dirección tiene mal formato.
5. **inet\_ntop():**
6. **N**etwork **TO P**resentation.
7. Convierte una IP binaria almacenada con formato struc in\_addr o in\_addr6 a un string con notación punto-decimal o hexa-dos-puntos dependiendo el tipo de familia especificado en su invocación.

// IPv4:

char ip4[INET\_ADDRSTRLEN]; // space to hold the IPv4 string

struct sockaddr\_in sa // pretend this is loaded with something

inet\_ntop(AF\_INET, &(sa.sin\_addr), ip4, INET\_ADDRSTRLEN);

printf("The IPv4 address is: %s\n", ip4);

// IPv6:

char ip6[INET6\_ADDRSTRLEN];// space to hold the IPv6 string

struct sockaddr\_in6 sa6; // pretend this is loaded with something

inet\_ntop(AF\_INET6, &(sa6.sin6\_addr), ip6, INET6\_ADDRSTRLEN);

printf("The address is: %s\n", ip6);

1. Se la invoca indicando:
   1. Familia de dirección.
   2. Dir. de memoria donde se almacena la IP (ip\_dest).
   3. Puntero al string que almacenara el resultado.
   4. Tamaño máximo de ese string.
2. INET\_ADDRSTRLEN y INET6\_ADDRSTRLEN son macros que almacenan el tamaño máximo del string utilizado para almacenar la dirección IP.

# Llamadas al sistema

A continuación se explicara como invocar ciertas funciones que permiten acceder a la funcionalidad de la red del núcleo Unix.

1. **getaddrinfo():**
2. Se encarga de resolver queries DNS y además llena los structs necesarios con el resultado de la resolución DNS.
3. A continuación se muestra la interfaz de la función:

#include <sys/types.h>

#include <sys/socket.h>

#include <netdb.h>

int getaddrinfo(const char \*node, // e.g. "www.example.com" or IP

const char \*service, // e.g. "http" or port number

const struct addrinfo \*hints,

struct addrinfo \*\*res); //resulting linked-list of structs addrinfo

1. Esta función recibe tres parámetros:
   1. node = hostname o IP.
   2. service = número de puerto o nombre de servicio (“tcp”).
   3. hints = puntero al struct addrinfo previamente inicializado por nosotros.
   4. res = puntero que contendrá una lista enlazada con los resultados de la funcion.
2. Devuelve un puntero a una lista enlazada de structs addrinfo. Los structs addrinfo contienen structs sockaddr cuyo contenido podrá ser utilizado más adelante.
3. Retorna <> 0 si hubo error y este se puede imprimir usando **gai\_strerror()**.
4. Una vez que terminamos de usar la lista de structs addrinfo debemos liberar la memoria que esta ocupa usando **freeaddrinfo(**<primer\_nodo>**)**.
5. Ejemplo de utilización:

int status;

struct addrinfo hints;

struct addrinfo \*servinfo; // will point to the results

memset(&hints, 0, sizeof hints); // make sure the struct is empty

hints.ai\_family = AF\_UNSPEC; // don't care IPv4 or IPv6

hints.ai\_socktype = SOCK\_STREAM; // TCP stream sockets

hints.ai\_flags = AI\_PASSIVE; // fill in my IP for me

if ((status = getaddrinfo(NULL, "3490", &hints, &servinfo)) != 0) {

fprintf(stderr, "getaddrinfo error: %s\n", gai\_strerror(status));

exit(1);

}

// servinfo now points to a linked list of 1 or more struct addrinfos

// ... do everything until you don't need servinfo anymore ....

freeaddrinfo(servinfo); // free the linked-list

1. Ejemplo de recorrido de lista enlazada de structs addrinfo resultante:

//showip.c -- show IP addresses for a host given on the command line

#include <stdio.h>

#include <string.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netdb.h>

#include <arpa/inet.h>

#include <netinet/in.h>

int main(int argc, char \*argv[])

{

struct addrinfo hints, \*res, \*p;

int status;

char ipstr[INET6\_ADDRSTRLEN];

if (argc != 2)

{

fprintf(stderr,"usage: showip hostname\n");

return 1;

}

memset(&hints, 0, sizeof hints);

hints.ai\_family = AF\_UNSPEC; // AF\_INET or AF\_INET6 to force version

hints.ai\_socktype = SOCK\_STREAM;

if ((status = getaddrinfo(argv[1], NULL, &hints, &res)) != 0)

{

fprintf(stderr, "getaddrinfo: %s\n", gai\_strerror(status));

return 2;

}

printf("IP addresses for %s:\n\n", argv[1]);

for(p = res;p != NULL; p = p->ai\_next)

{

void \*addr;

char \*ipver;

// get the pointer to the address itself,

// different fields in IPv4 and IPv6:

if (p->ai\_family == AF\_INET) // IPv4

{

struct sockaddr\_in \*ipv4 = (struct sockaddr\_in \*)p->ai\_addr;

addr = &(ipv4->sin\_addr);

ipver = "IPv4";

}

else // IPv6

{

struct sockaddr\_in6 \*ipv6 = (struct sockaddr\_in6 \*)p->ai\_addr;

addr = &(ipv6->sin6\_addr);

ipver = "IPv6";

}

// convert the IP to a string and print it:

inet\_ntop(p->ai\_family, addr, ipstr, sizeof ipstr);

printf(" %s: %s\n", ipver, ipstr);

}

freeaddrinfo(res); // free the linked list

return 0;

}

1. **socket():**
2. Recibe tres argumentos para indicar el tipo de socket que se desea.
3. Interfaz de la función:

#include <sys/types.h>

#include <sys/socket.h>

int socket(int domain, int type, int protocol);

1. Los argumentos pueden indicarse de dos maneras:
   1. Harcodeados:

* domain = PF\_INET o PF\_INET6.
* type = SOCK\_STREAM o SOCK\_DGRAM.
* protocol = valor del protocolo a utilizar (valores en /etc/services)
  1. Usando **getadrrinfo()**.

1. Devuelve un socket descriptor que puede ser utilizado en otras llamadas al sistema.
2. Retorna -1 en **errno** si hubo algún error.
3. Ejemplo de utilización de **getaddrinfo()**.

int s;

struct addrinfo hints;

struct addrinfo \*res;

// do the lookup

// [pretend we already filled out the "hints" struct]

getaddrinfo("www.example.com", "http", &hints, &res);

// [again, you should do error-checking on getaddrinfo(), and walk

// the "res" linked list looking for valid entries instead of just

// assuming the first one is good (like many of these examples do.)

// See the section on client/server for real examples.]

s = socket(res->ai\_family, res->ai\_socktype, res->ai\_protocol);

1. **bind():**
2. Esta función se encarga de asignar un puerto efímero al socket descriptor creado con **socket()**.
3. Solo es utilizado cuando se va a realizar **listen()**, pues nos interesa saber el puerto local que será usado.
4. Interfaz de la función:

#include <sys/types.h>

#include <sys/socket.h>

int bind(int sockfd, struct sockaddr \*my\_addr, int addrlen);

1. sockfd = socket descriptor obtenido con **socket()**.
2. \*my\_addr = puntero a un struct sockaddr que contiene mi IP + puerto.
3. addrlen = es la longitud de \*my\_addr expresada en bytes.
4. Retorna -1 en **errno** si hay error.
5. Ejemplo de utilización:

struct addrinfo hints, \*res;

int sockfd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);

hints.ai\_family = AF\_UNSPEC; // use IPv4 or IPv6, whichever

hints.ai\_socktype = SOCK\_STREAM; // TCP Socket

hints.ai\_flags = AI\_PASSIVE; // fill in my IP for me

getaddrinfo(NULL, "3490", &hints, &res);

// make a socket:

sockfd = socket(res->ai\_family, res->ai\_socktype, res->ai\_protocol);

// bind it to the port we passed in to getaddrinfo():

bind(sockfd, res->ai\_addr, res->ai\_addrlen);

1. AI\_PASSIVE = indica al kernel que realice bind con la IP que está utilizando el host.
2. En algunos casos cuando el servidor se reinicia y se vuelve a ejecutar **bind()** la llamada falla. Esto sucede por el tiempo 2MSL que protege a los sockets de reencarnaciones.
3. A continuación se muestra un ejemplo que fuerza la reutilización de ese puerto:

int yes=1;

// lose the pesky "Address already in use" error message

if (setsockopt(listener,SOL\_SOCKET,SO\_REUSEADDR,&yes,sizeof yes) == -1)

{

perror("setsockopt");

exit(1);

}

1. **connect():**
2. Esta llamada permite conectarse a un host remoto.
3. Interfaz de la función:

#include <sys/types.h>

#include <sys/socket.h>

int connect(int sockfd, struct sockaddr \*serv\_addr, int addrlen);

1. sockfd = socket descriptor obtenido con **socket()**.
2. \*serv\_addr = IP + puerto destino.
3. addrlen = longitud de serv\_addr expresada en bytes.
4. Todos estos valores son obtenidos de la función **getaddrinfo()**.
5. Retorna -1 en **errno** si hay error.
6. Ejemplo de utilización. Observar que no se invocó a **bind()** pues no es necesario conocer el número de puerto asignado:

struct addrinfo hints;

struct addrinfo \*res;

int sockfd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);

hints.ai\_family = AF\_UNSPEC;

hints.ai\_socktype = SOCK\_STREAM;

getaddrinfo("www.example.com", "3490", &hints, &res);

// make a socket:

sockfd = socket(res->ai\_family, res->ai\_socktype, res->ai\_protocol);

// connect!

connect(sockfd, res->ai\_addr, res->ai\_addrlen);

1. **listen():**
2. Esta llamada prepara buffers de memoria para permitir que un host remoto se conecte con nosotros, proceso que consiste de dos pasos: primero escuchar **listen()** y luego aceptar **accept()** conexiones entrantes.
3. Para poder escuchar, primero se debe hacer **bind()** para conocer el puerto en el que se ejecuta el servicio.
4. Interfaz de la función:

int listen(int sockfd, int backlog);

1. sockfd = socket descriptor obtenido con **socket()**.
2. backlog = número de conexiones pendientes de **accept()** que pueden ser encolados. Normalmente es almacenado como un macro.
3. Retorna -1 en **errno** si hay error.
4. **accept():**
5. Es la segunda parte del proceso que permite que un host remoto se conecte con nosotros.
6. Luego de invocar a **listen()** sobre un determinado puerto, cualquier host remoto puede realizar **connect()** sobre dicho puerto. Su solicitud de conexión será encolada hasta que usemos **accept()**.
7. Interfaz de la función:

#include <sys/types.h>

#include <sys/socket.h>

int accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen);

1. sockfd = socket descriptor obtenido con **listen()**.
2. addr = puntero a un sockaddr\_storage que luego será casteado a un sockaddr\_in o sockaddr\_in6 dependiendo del tipo de familia de conexión (AFI\_INET o AF\_INET6).
3. addrlen = puntero a un integer que almacena **sizeof(**struct sockaddr\_storage**)**.
4. Devuelve un nuevo socket descriptor listo para realizar **read()** y **write()**.
5. Retorna -1 en **errno** si hay error.
6. Ejemplo de utilización:

#include <string.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#define MYPORT "3490" // the port users will be connecting to

#define BACKLOG 10 // how many pending connections queue will hold

int main(void)

{

struct sockaddr\_storage their\_addr;

socklen\_t addr\_size;

struct addrinfo hints, \*res;

int sockfd, new\_fd;

// !! don't forget your error checking for these calls !!

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);

hints.ai\_family = AF\_UNSPEC; // use IPv4 or IPv6, whichever

hints.ai\_socktype = SOCK\_STREAM;

hints.ai\_flags = AI\_PASSIVE; // fill in my IP for me

getaddrinfo(NULL, MYPORT, &hints, &res);

// make a socket, bind it, and listen on it:

sockfd = socket(res->ai\_family, res->ai\_socktype, res->ai\_protocol);

bind(sockfd, res->ai\_addr, res->ai\_addrlen);

listen(sockfd, BACKLOG);

// now accept an incoming connection:

addr\_size = sizeof their\_addr;

new\_fd = accept(sockfd, (struct sockaddr \*)&their\_addr, &addr\_size);

// ready to communicate on socket descriptor new\_fd!

}

1. **send():**
2. Permite enviar datos hacia un socket descriptor.
3. Interfaz de la función:

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

1. sockfd = socket descriptor al cual se le quiere enviar datos (puede ser obtenido por **socket()** o por **accept()**)
2. msg = es un puntero a los datos que se desean enviar.
3. len = longitud de los datos a enviar expresada en bytes.
4. flags = debe ser seteado a 0 (ver **send()** **man page** para + info).
5. Devuelve la cantidad de datos escritos en el socket, la cual a veces puede ser menor a la enviada y queda en el programador el envío de los datos restantes.
6. Retorna -1 en **errno** si hay error.
7. Ejemplo de utilización:

char \*msg = "Beej was here!";

int len, bytes\_sent;

len = strlen(msg);

bytes\_sent = send(sockfd, msg, len, 0);

1. **recv():**
2. Permite leer información desde un socket descriptor.
3. Interfaz de la función:

int recv(int sockfd, void \*buf, int len, int flags);

1. sockfd = socket descriptor sobre el cual se va a leer.
2. buf = puntero al buffer donde se almacenara lo leído.
3. len = tamaño máximo de buf expresado en bytes.
4. flags = seteado en 0 (ver **recv()** **man page** para + info).
5. Devuelve la cantidad de bytes leidas, es decir, cantidad de bytes copiadas a buf.
6. Retorna -1 en **errno** si hay error.
7. Si retorna 0 es porque el otro extremo del socket ha cerrado la conexión y esta es la única forma de indicarlo.
8. **close():**
9. Cierra la conexión de un socket descriptor.
10. Impide continuar leyendo o escribiendo del socket descriptor cerrado.
11. Ejemplo de utilización:

close(sockfd);

1. **shutdown()**:
2. Permite cambiar la usabilidad de un socket descriptor pero no lo cierra.
3. Como los sockets son full-duplex, **shutdown()** permite cerrar un canal de la comunicación o ambos si se desea (como en **close()**).
4. Interfaz de la función:

int shutdown(int sockfd, int how);

1. sockfd = socket descriptor a cerrar.
2. how = método de cierre:
   1. 0 ---> deshabilita futuros **recv()**.
   2. 1 ---> deshabilita futuros **send()**.
   3. 2 ---> deshabilita futuros **recv()** y **send()** (similar a close).
3. Retorna -1 en **errno** si hay error o 0 si tuvo éxito.
4. ES IMPORTANTE RECORDAR INVOCAR A **close()**, PUES **shutdown()** NO CIERRA EL SOCKET DESCRIPTOR SINO QUE CAMBIA SU USABILIDAD.

# Bloqueo

Las siguientes llamadas al sistema son bloqueadoras, es decir, ejecutan **sleep()**:

1. **listen()**.
2. **accept()**.
3. **recv()**.

La razón por la que lo hacen es simplemente porque pueden. Cuando se invoca a **socket()**, el kernel Unix otorga permisos para ejecutar **sleep()** al socket descriptor que devuelve.

Si queremos evitar que el socket a utilizar sea bloqueante se debe utilizar la función **fcntl()** de la siguiente:

#include <unistd.h>

#include <fcntl.h>

sockfd = socket(PF\_INET, SOCK\_STREAM, 0);

fcntl(sockfd, F\_SETFL, O\_NONBLOCK);

Al no permitir que un socket bloquee, se lo puede consultar (*polling*) para obtener información. Si se intenta leer un socket no bloqueante y no tiene datos devolverá -1 y **errno** será seteado con EAGAIN o EWOULDBLOCK. Sin embargo, es una mala idea realizar *polling* ya que se gastaran ciclos del procesador esperando que el socket reciba datos.

Una solución más elegante para verificar si hay datos esperando a ser leídos en el socket es utilizar la función **select()**.

# Portabilidad entre IPv4 e IPv6

Los siguientes puntos son recomendaciones para mantener un código IP-agnostico y lograr una mayor portabilidad:

1. Utilizar **getaddrinfo()** para obtener toda la info del struct sockaddr en vez de especificarla a mano.
2. Eliminar todo lo hardcodeado y sacarlo a funciones auxiliares.
3. Cambiar AF\_INET por AF\_INET6.
4. Cambiar PF\_INET por PF\_INET6.
5. Reemplazar la asignación a los struct in\_addr e in\_addr6 por INADDR\_ANY

struct sockaddr\_in sa;

struct sockaddr\_in6 sa6;

sa.sin\_addr.s\_addr = INADDR\_ANY; // use my IPv4 address

sa6.sin6\_addr = in6addr\_any; // use my IPv6 address

1. Reemplazar el struct sockaddr\_in por el sockaddr\_in6 y recordar que no hay padding y por ende no hay que realizar **memset()** del campo sin\_zero.
2. Reemplazar el struct in\_addr por el in\_addr6.
3. Usar **inet\_pton()** e **inet\_ntop()**.
4. Reemplazar **gethostbyname()** por **getaddrinfo()**.
5. Reemplazar **gethostbyaddr()** por **getnameinfo()**.
6. Reemplaza INADDR\_BROADCAST por IPv6 multicast.

# Ejemplo de servidor TCP

//server.c -- a stream socket server demo

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <errno.h>

#include <string.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <netdb.h>

#include <arpa/inet.h>

#include <sys/wait.h>

#include <signal.h>

#define PORT "3490" // the port users will be connecting to

#define BACKLOG 10 // how many pending connections queue will hold

void sigchld\_handler(int s)

{

// waitpid() might overwrite errno, so we save and restore it:

int saved\_errno = errno;

while(waitpid(-1, NULL, WNOHANG) > 0);

errno = saved\_errno;

}

// get sockaddr, IPv4 or IPv6:

void \*get\_in\_addr(struct sockaddr \*sa)

{

if (sa->sa\_family == AF\_INET)

{

return &(((struct sockaddr\_in\*)sa)->sin\_addr);

}

return &(((struct sockaddr\_in6\*)sa)->sin6\_addr);

}

int main(void)

{

int sockfd, new\_fd; // listen on sock\_fd, new connection on new\_fd

struct addrinfo hints, \*servinfo, \*p;

struct sockaddr\_storage their\_addr; // connector's address information

socklen\_t sin\_size;

struct sigaction sa;

int yes=1;

char s[INET6\_ADDRSTRLEN];

int rv;

memset(&hints, 0, sizeof hints);

hints.ai\_family = AF\_UNSPEC;

hints.ai\_socktype = SOCK\_STREAM;

hints.ai\_flags = AI\_PASSIVE; // use my IP

if ((rv = getaddrinfo(NULL, PORT, &hints, &servinfo)) != 0)

{

fprintf(stderr, "getaddrinfo: %s\n", gai\_strerror(rv));

return 1;

}

// loop through all the results and bind to the first we can

for(p = servinfo; p != NULL; p = p->ai\_next)

{

if ((sockfd = socket(p->ai\_family, p->ai\_socktype,

p->ai\_protocol)) == -1) {

perror("server: socket");

continue;

}

if (setsockopt(sockfd, SOL\_SOCKET, SO\_REUSEADDR, &yes, sizeof(int)) == -1) {

perror("setsockopt");

exit(1);

}

if (bind(sockfd, p->ai\_addr, p->ai\_addrlen) == -1)

{

close(sockfd);

perror("server: bind");

continue;

}

break;

}

freeaddrinfo(servinfo); // all done with this structure

if (p == NULL)

{

fprintf(stderr, "server: failed to bind\n");

exit(1);

}

if (listen(sockfd, BACKLOG) == -1)

{

perror("listen");

exit(1);

}

sa.sa\_handler = sigchld\_handler; // reap all dead processes

sigemptyset(&sa.sa\_mask);

sa.sa\_flags = SA\_RESTART;

if (sigaction(SIGCHLD, &sa, NULL) == -1)

{

perror("sigaction");

exit(1);

}

printf("server: waiting for connections...\n");

while(1) // main accept() loop

{

sin\_size = sizeof their\_addr;

new\_fd = accept(sockfd, (struct sockaddr \*)&their\_addr, &sin\_size);

if (new\_fd == -1)

{

perror("accept");

continue;

}

inet\_ntop(their\_addr.ss\_family,

get\_in\_addr((struct sockaddr \*)&their\_addr),

s, sizeof s);

printf("server: got connection from %s\n", s);

if (!fork()) // this is the child process

{

close(sockfd); // child doesn't need the listener

if (send(new\_fd, "Hello, world!", 13, 0) == -1)

perror("send");

close(new\_fd);

exit(0);

}

close(new\_fd); // parent doesn't need this

}

return 0;

}

# Ejemplo de cliente TCP

//client.c -- a stream socket client demo

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <errno.h>

#include <string.h>

#include <netdb.h>

#include <sys/types.h>

#include <netinet/in.h>

#include <sys/socket.h>

#include <arpa/inet.h>

#define PORT "3490" // the port client will be connecting to

#define MAXDATASIZE 100 // max number of bytes we can get at once

// get sockaddr, IPv4 or IPv6:

void \*get\_in\_addr(struct sockaddr \*sa)

{

if (sa->sa\_family == AF\_INET)

{

return &(((struct sockaddr\_in\*)sa)->sin\_addr);

}

return &(((struct sockaddr\_in6\*)sa)->sin6\_addr);

}

int main(int argc, char \*argv[])

{

int sockfd, numbytes;

char buf[MAXDATASIZE];

struct addrinfo hints, \*servinfo, \*p;

int rv;

char s[INET6\_ADDRSTRLEN];

if (argc != 2)

{

fprintf(stderr,"usage: client hostname\n");

exit(1);

}

memset(&hints, 0, sizeof hints);

hints.ai\_family = AF\_UNSPEC;

hints.ai\_socktype = SOCK\_STREAM;

if ((rv = getaddrinfo(argv[1], PORT, &hints, &servinfo)) != 0)

{

fprintf(stderr, "getaddrinfo: %s\n", gai\_strerror(rv));

return 1;

}

// loop through all the results and connect to the first we can

for(p = servinfo; p != NULL; p = p->ai\_next) {

if ((sockfd = socket(p->ai\_family, p->ai\_socktype, p->ai\_protocol)) == -1)

{

perror("client: socket");

continue;

}

if (connect(sockfd, p->ai\_addr, p->ai\_addrlen) == -1)

{

close(sockfd);

perror("client: connect");

continue;

}

break;

}

if (p == NULL)

{

fprintf(stderr, "client: failed to connect\n");

return 2;

}

inet\_ntop(p->ai\_family, get\_in\_addr((struct sockaddr \*)p->ai\_addr), s, sizeof s);

printf("client: connecting to %s\n", s);

freeaddrinfo(servinfo); // all done with this structure

if ((numbytes = recv(sockfd, buf, MAXDATASIZE-1, 0)) == -1)

{

perror("recv");

exit(1);

}

buf[numbytes] = '\0';

printf("client: received '%s'\n",buf);

close(sockfd);

return 0;

}

# Referencias:

[1] – Brian Hall – “Beej’s Guide to Network Programming – Using Internet Sockets – v3.0.21”. [Online] Disponible: <http://beej.us/guide/bgnet/pdf/bgnet_A4_2.pdf>. Accedido 15 de Septiembre 2018.

[2] – Juan Carlos Romero – “Modelo cliente-servidor Sockets”. [Online] Disponible: <http://www.grch.com.ar/docs/bdd/apuntes/unidad.ii/sockets/Practica%20Sockets.ppt>. Accedido 15 de Septiembre 2018.