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

¿Qué es un socket? 4

Tipos de sockets: 4

Byte order 5

Estructuras involucradas 6

1. socket descriptor: 6
2. addrinfo: 6
3. sockaddr: 6
4. sockaddr_in y sockaddr_in6: 6
5. in_addr y in_addr6: 7
6. sockaddr_storage: 7

Manejo de direcciones IP 8

1. inet_pton(): 8
2. inet_ntop(): 8

Llamadas al sistema 9

1. getaddrinfo(): 9
2. socket(): 11
3. bind(): 11
4. connect(): 12
5. listen(): 13
6. accept(): 13
7. send(): 14
8. recv(): 15
9. close(): 15
10. shutdown(): 15

Bloqueo 16

Portabilidad entre IPv4 e IPv6 17

Ejemplo de servidor TCP 18

Ejemplo de cliente TCP 20

Referencias: 22

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:
    a. Orientados a la conexión ---> TCP.
    b. Telnet, HTTP.
2. Datagram sockets:
    a. No orientados a la conexión ---> UDP.
    b. 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:**
   i. Es un *int* regular.

```
int socket_desc;
```

2. **addrinfo:**
   i. Utilizada para preparar las estructuras de dirección del socket.
   ii. Primero le inicializaremos algunos valores y luego se invoca a **getaddrinfo()** para que la complete.
   iii. 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
};
```

3. **sockaddr:**
   i. 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
};
```

   ii. sa_family ---> AF_INET (IPv4) o AF_INET6 (IPv6).
   iii. sa_data ---> contiene la combinación IP destino + puerto del socket.

4. **sockaddr_in** y **sockaddr_in6:**
   i. 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.
   ii. 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.
   iii. 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
};
```

iv. sin_family <==> sa_family de sockaddr.
v. sin_port debe estar en **Network Byte Order** usando **htons()** (pues es un short sin signo).
vi. sin_zero es para padding y debe ser inicializado con **memset()**.

5. **in_addr** y **in_addr6:**
   i. Mantenida solo por motivos históricos.
   ii. 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
};
```

   iii. 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;
```

6. **sockaddr_storage:**
   i. Es otra estructura paralela.
   ii. Diseñada para almacenar tanto estructuras IPv4 como IPv6.
   iii. 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];
};
```

   iv. 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():**
   i. **P**resentation **TO N**etwork.
   ii. 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
```

   iii. Se la invoca indicando:
      a. Familia de dirección.
      b. Puntero char de string a convertir.
      c. Dir. de memoria donde se almacenara la IP (ip_dest)
   iv. La IP es convertida a un struct in_addr y copiada en ip_dest.
   v. ip_dest debe tener el mismo tamaño que el struct in_addr (4 bytes).
   vi. Retorna -1 en error, 0 si la dirección tiene mal formato.

2. **inet_ntop():**
   i. **N**etwork **TO P**resentation.
   ii. 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);
```

   iii. Se la invoca indicando:
      a. Familia de dirección.
      b. Dir. de memoria donde se almacena la IP (ip_dest).
      c. Puntero al string que almacenara el resultado.
      d. Tamaño máximo de ese string.
   iv. 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():**
   i. Se encarga de resolver queries DNS y además llena los structs necesarios con el resultado de la resolución DNS.
   ii. 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
```

   iii. Esta función recibe tres parámetros:
      a. node = hostname o IP.
      b. service = número de puerto o nombre de servicio (“tcp”).
      c. hints = puntero al struct addrinfo previamente inicializado por nosotros.
      d. res = puntero que contendrá una lista enlazada con los resultados de la funcion.
   iv. Devuelve un puntero a una lista enlazada de structs addrinfo. Los structs addrinfo contienen structs sockaddr cuyo contenido podrá ser utilizado más adelante.
   v. Retorna <> 0 si hubo error y este se puede imprimir usando **gai_strerror()**.
   vi. Una vez que terminamos de usar la lista de structs addrinfo debemos liberar la memoria que esta ocupa usando **freeaddrinfo(**<primer_nodo>**)**.
   vii. 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
```

viii. 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;
}
```

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

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

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

    iii. Los argumentos pueden indicarse de dos maneras:
        a. Harcodeados:
            - domain = PF_INET o PF_INET6.
            - type = SOCK_STREAM o SOCK_DGRAM.
            - protocol = valor del protocolo a utilizar (valores en /etc/services)
        b. Usando **getadrrinfo()**.
    iv. Devuelve un socket descriptor que puede ser utilizado en otras llamadas al sistema.
    v. Retorna -1 en **errno** si hubo algún error.
    vi. 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);
```

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

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

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

    iv. sockfd = socket descriptor obtenido con **socket()**.
    v. *my_addr = puntero a un struct sockaddr que contiene mi IP + puerto.
    vi. addrlen = es la longitud de *my_addr expresada en bytes.
    vii. Retorna -1 en **errno** si hay error.
    viii. 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);
```

ix. AI_PASSIVE = indica al kernel que realice bind con la IP que está utilizando el host.
x. 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.
xi. 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);
}
```

4. **connect():**
   i. Esta llamada permite conectarse a un host remoto.
   ii. Interfaz de la función:

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

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

iii. sockfd = socket descriptor obtenido con **socket()**.
iv. *serv_addr = IP + puerto destino.
v. addrlen = longitud de serv_addr expresada en bytes.
vi. Todos estos valores son obtenidos de la función **getaddrinfo()**.
vii. Retorna -1 en **errno** si hay error.
viii. 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);
```

5. **listen():**
   i. 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.
   ii. Para poder escuchar, primero se debe hacer **bind()** para conocer el puerto en el que se ejecuta el servicio.
   iii. Interfaz de la función:

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

   iv. sockfd = socket descriptor obtenido con **socket()**.
   v. backlog = número de conexiones pendientes de **accept()** que pueden ser encolados. Normalmente es almacenado como un macro.
   vi. Retorna -1 en **errno** si hay error.

6. **accept():**
   i. Es la segunda parte del proceso que permite que un host remoto se conecte con nosotros.
   ii. 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()**.
   iii. Interfaz de la función:

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

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

   iv. sockfd = socket descriptor obtenido con **listen()**.
   v. 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).
   vi. addrlen = puntero a un integer que almacena **sizeof(**struct sockaddr_storage**)**.
   vii. Devuelve un nuevo socket descriptor listo para realizar **read()** y **write()**.

viii. Retorna -1 en **errno** si hay error.
ix. 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!
}
```

7. send():

i. Permite enviar datos hacia un socket descriptor.
ii. Interfaz de la función:

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

iii. sockfd = socket descriptor al cual se le quiere enviar datos (puede ser obtenido por **socket()** o por **accept()**)
iv. msg = es un puntero a los datos que se desean enviar.
v. len = longitud de los datos a enviar expresada en bytes.
vi. flags = debe ser seteado a 0 (ver **send() man page** para + info).
vii. 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.
viii. Retorna -1 en **errno** si hay error.
ix. Ejemplo de utilización:

```
char *msg = "Beej was here!";
int len, bytes_sent;

len = strlen(msg);
bytes_sent = send(sockfd, msg, len, 0);
```

8. **recv():**
   i. Permite leer información desde un socket descriptor.
   ii. Interfaz de la función:

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

   iii. sockfd = socket descriptor sobre el cual se va a leer.
   iv. buf = puntero al buffer donde se almacenara lo leído.
   v. len = tamaño máximo de buf expresado en bytes.
   vi. flags = seteado en 0 (ver **recv() man page** para + info).
   vii. Devuelve la cantidad de bytes leidas, es decir, cantidad de bytes copiadas a buf.
   viii. Retorna -1 en **errno** si hay error.
   ix. Si retorna 0 es porque el otro extremo del socket ha cerrado la conexión y esta es la única forma de indicarlo.

9. **close():**
   i. Cierra la conexión de un socket descriptor.
   ii. Impide continuar leyendo o escribiendo del socket descriptor cerrado.
   iii. Ejemplo de utilización:

```
close(sockfd);
```

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

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

   iv. sockfd = socket descriptor a cerrar.
   v. how = método de cierre:
      a. 0 ---> deshabilita futuros **recv()**.
      b. 1 ---> deshabilita futuros **send()**.
      c. 2 ---> deshabilita futuros **recv()** y **send()** (similar a close).
   vi. Retorna -1 en **errno** si hay error o 0 si tuvo éxito.
   vii. 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
```

6. 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.

7. Reemplazar el struct in_addr por el in_addr6.

8. Usar **inet_pton()** e **inet_ntop()**.

9. Reemplazar **gethostbyname()** por **getaddrinfo()**.

10. Reemplazar **gethostbyaddr()** por **getnameinfo()**.

11. 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.