CREACIÓN DE SOCKETS INTERNET EN LENGUAJE C

Diferentes tipos de sockets en Internet

Definición de socket: Un socket es cada uno de los extremos del canal de comunicación, es decir, es una manera de "hablar" con otro proceso. En el caso de los sockets internet, ese otro proceso se encuentra en otra computadora. Para ser más preciso, es una manera de hablar con otras computadoras usando **Descriptores de Archivo** estándar de Unix (en inglés llamados **File Descriptors**). En Unix, todas las acciones de entrada y salida son desempeñadas escribiendo o leyendo en uno de estos descriptores de archivo, los cuales son simplemente un número entero, asociado a un fichero abierto que puede ser una conexión de red, una terminal, o cualquier otro recurso.

Ahora bien, sobre los diferentes tipos de sockets en Internet, hay muchos tipos, pero aquí se mencionarán 2 de ellos: Sockets TCP (o Sockets de Flujo) SOCK_STREAM y Sockets UDP de Datagramas (SOCK_DGRAM).

Sockets de Flujo: Están libres de errores. Si, por ejemplo, enviáramos por el socket de flujo tres objetos "A, B, C", llegarán al destino en el mismo orden -- "A, B, C". Estos sockets usan TCP y es este protocolo el que asegura el orden de los objetos durante la transmisión.

Sockets de Datagramas: Utilizan UDP ("User Datagram Protocol"), y no necesitan de una conexión accesible como los Sockets de Flujo -- se construirá un paquete de datos con información sobre su destino y se enviará afuera, sin necesidad de una conexión.

Estructuras

Las estructuras son usadas en la programación de sockets para almacenar información sobre direcciones. La primera de ellas es **struct sockaddr**, la cual contiene información del socket.

```
struct sockaddr
{
   unsigned short sa_family;  /* Familia de la dirección */
   char sa_data[14];          /* 14 bytes de la dirección del protocolo */
};
```

Pero, existe otra estructura, **struct sockaddr_in**, la cual nos ayuda a hacer referencia a los elementos del socket.

```
struct sockaddr_in
{
   short int sin_family;        /* Familia de la Dirección             */
   unsigned short int sin_port; /* Puerto                              */
   struct in_addr sin_addr;     /* Dirección de Internet               */
   unsigned char sin_zero[8];   /* Del mismo tamaño que struct sockaddr */
};
```

Nota. sin_zero puede ser configurada con ceros usando las funciones memset() o bzero().

La siguiente estructura no es muy usada, se llama **struct in_addr**:

```
struct in_addr
{
   unsigned   long s_addr;
};
```

En su lugar, es más usada esta estructura llamada **struct hostent**, con la cual obtenemos información del nodo remoto

Aquí se puede ver su definición:

```
struct hostent
{
   char *h_name;                  /* El nombre oficial del nodo.              */
   char **h_aliases;              /* Lista de Alias.                          */
   int h_addrtype;                /* Tipo de dirección del nodo.              */
   int h_length;                  /* Longitud de la dirección.                */
   char **h_addr_list;            /* Lista de direcciones del nombre del      */
                                   /* servidor.                                */
   #define h_addr  h_addr_list[0] /* Dirección, para la compatibilidad con    */
                                   /* anteriores.                              */
};
```

Esta estructura está definida en el archivo netdb.h.

Conversiones

Existen dos tipos de ordenamiento de bytes: "*bytes más significativos"*, y "*bytes menos significativos"*. A estas conversiones se les llama "Ordenación de Bytes para Redes", algunas máquinas utilizan este tipo de ordenación para guardar sus datos internamente.

Dentro de las estructuras, existen dos tipos de dato que vamos a manejar: short y long. Las siguientes funciones se encargan de hacer este tipo de conversiones:

- htons() -> host to network small -> "Nodo a variable corta de Red" (decimal a binario corto)
- htonl() -> host to network long -> "Nodo a variable larga de Red" (decimal a binario largo)
- ntohs() -> network to host small -> "Red a variable corta de Nodo" (binario a decimal corto)
- ntohl() -> network to host long -> "Red a variable larga de Nodo" (binario a decimal largo)

Direcciones IP

En C, existen algunas funciones que nos ayudarán a manipular *direcciones IP*. En esta sección se hablará de las funciones inet_addr() y inet_ntoa().

Por un lado, la función inet_addr() convierte una dirección IP en un entero largo sin signo (unsigned long int), por ejemplo:

```
dest.sin_addr.s_addr = inet_addr("195.65.36.12");

/*Recordar que esto sería así, siempre que tengamos una estructura "dest" del tipo
sockaddr_in*/
```

Por otro lado, inet_ntoa() convierte a una cadena que contiene una dirección IP en un entero largo. Por ejemplo:

```
(...)

char *ip;

ip=inet_ntoa(dest.sin_addr);

printf("La direccion es: %s\n",ip);

(...)
```

Se debe recordar también que la función inet_addr() devuelve la dirección en formato binario por lo que no necesitaremos llamar a htonl().

Funciones Importantes

En esta sección, (en la cual se nombrarán algunas de las funciones más utilizadas para la programación en C de sockets), se mostrará la sintaxis de la función, las bibliotecas necesarias a incluir para llamarla, y algunos pequeños comentarios.

socket()

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

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

Analicemos los argumentos:

- **domain.** Se podrá establecer como AF_INET (para usar los protocolos ARPA de Internet).
- **type.** Aquí se debe especificar la clase de socket que queremos usar (de Flujos o de Datagramas). Las variables que deben aparecer son SOCK_STREAM o SOCK_DGRAM según se use sockets de Flujo o de Datagramas, respectivamente.
- **protocol.** Aquí, simplemente se puede establecer el protocolo a 0.

La función socket(…) nos devuelve un descriptor de socket, el cual podremos usar luego para llamadas al sistema. Si nos devuelve *-1*, se ha producido un error (esto puede resultar útil para rutinas de verificación de errores).

---

bind()

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

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

Analicemos los argumentos:

- **fd.** Es el descriptor de archivo del socket servidor, devuelto por la llamada a socket().
- **my_addr.** es un puntero a una estructura sockaddr

- **addrlen.** contiene la longitud de la estructura sockaddr a la cual apunta el puntero my_addr.

En el tema anterior, se vio que en los sockets UNIX (o locales), la llamada bind() asocia el socket a un archivo local. En los sockets TCP, la llamada bind() asocia el socket a un puerto local (el puerto es local al servidor).

Por otro lado, podremos hacer que nuestra dirección IP y puerto sean elegidos automáticamente:

```
server.sin_port = 0;            /* bind() elegirá un puerto aleatoriamente */
server.sin_addr.s_addr = INADDR_ANY; /* asigna IP del servidor automáticamente */
```

Un aspecto importante sobre los puertos y la llamada bind() es que todos los puertos menores de 1024 están reservados. Se podrá establecer un puerto, siempre que esté entre 1024 y 65535 (y siempre que no estén siendo usados por otros programas).

---

connect() → Esta función es del cliente

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

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

Analicemos los argumentos:

- **fd.** Deberá configurarse como el file descriptor del socket, el cual fue devuelto por la llamada a socket().
- **serv_addr.** Es un puntero a la estructura sockaddr la cual contiene la dirección IP destino y el puerto.
- **addrlen.** Análogamente de lo que pasaba con bind(), este argumento deberá establecerse como sizeof(struct sockaddr).

La función connect() se usa para conectarse a un puerto definido en una dirección IP. Devolverá -1 si ocurre algún error.

---

listen()

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

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

Veamos los argumentos de listen():

- **fd.** Es el descriptor del socket, el cual fue devuelto por la llamada a socket()
- **backlog.** Es el número de conexiones permitidas.

La función listen() se usa si se están esperando conexiones entrantes, lo cual significa, si se quiere, que alguien pueda conectarse a nuestra máquina.

Después de llamar a listen(), se deberá llamar a accept(), para así aceptar las conexiones entrantes. La secuencia resumida de llamadas al sistema es:

1. socket()
2. bind()
3. listen()
4. accept()

Como todas las funciones descritas arriba, listen() devolverá -1 en caso de error.

---

accept()

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

int accept(int fd, void *addr, int *addrlen);
```

Veamos los argumentos de la función:

- **fd. Es el descriptor de archivo del socket servidor**, que fue devuelto por la llamada a socket().
- **addr.** Es un puntero a una estructura sockaddr_in en la que se pueda determinar qué nodo nos está contactando y desde qué puerto (es decir, un cliente y por lo tanto **aquí va la estructura socket cliente del lado del servidor**).
- **addrlen.** Es la longitud de la estructura a la que apunta el argumento addr, por lo que conviene establecerlo como sizeof(struct sockaddr_in), antes de que su dirección sea pasada a accept().

Cuando alguien intenta conectarse a nuestra computadora, se debe usar accept() para conseguir la conexión. Es muy fácil de entender con la siguiente frase: "alguien sólo podrá conectarse a nuestra máquina, si nosotros aceptamos".

A continuación, un ejemplo del uso de accept() para obtener la conexión, ya que esta llamada es un poco diferente de las demás.

```
(...)

sin_size = sizeof(cliente);
/* En la siguiente linea se llama a accept() */
if ((fd2 = accept(fd1,(struct sockaddr *)&client,&sin_size))==-1){
printf("accept() error\n");
exit(-1);
}

(...)
```

En este punto, lo que devuelve accept() se asignará a la variable fd2 que es el descriptor de archivo del socket cliente que está del lado del servidor.

---

send()

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

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

Y sobre los argumentos de esta llamada:

- **fd.** Es el descriptor de archivo del socket cliente, con el cual se desea enviar datos.
- **msg.** Es un puntero apuntando al dato que se quiere enviar.
- **len.** es la longitud del dato que se quiere enviar (en bytes).
- **flags.** deberá ser establecido a 0.

Al igual que todas las demás llamadas, send() devuelve -1 en caso de error, o el número de bytes enviados en caso de éxito.

El propósito de la función send() es enviar datos usando sockets de flujo es decir, sockets TCP. Si se desea enviar datos usando sockets no conectados de datagramas debe usarse la llamada sendto().

---

recv()

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

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

Veamos los argumentos:

- **fd.** Es el descriptor del socket por el cual se leerán datos.
- **buf.** Es el búfer en el cual se guardará la información a recibir.
- **len.** Es la longitud máxima que podrá tener el búffer.
- **flags.** Por ahora, se deberá establecer como 0.

Análogamente a send(), recv() devuelve el número de bytes leídos en el búfer, o -1 si se produjo un error.

Al igual de lo que se dijo para send(), esta función es usada con datos en sockets de flujo o sockets conectados de datagramas. Si se deseara enviar, o en este caso, recibir datos usando sockets *desconectados* de Datagramas, se debe usar la llamada recvfrom().

---

close()

```
#include <unistd.h>

close(fd);
```

La función close() es usada para cerrar la conexión de nuestro descriptor de socket. Si llamamos a close() no se podrá escribir o leer usando ese socket, y si alguien trata de hacerlo recibirá un mensaje de error.

---

shutdown()

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

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

Veamos los argumentos:

- **fd.** Es el descriptor de archivo del socket al que queremos aplicar esta llamada.
- **how.** Sólo se podrá establecer uno de estos valores:
    - **0** Prohibido recibir.
    - **1** Prohibido enviar.
    - **2** Prohibido recibir y enviar.

Es lo mismo llamar a close() que establecer un shutdown() con valor 2. Devolverá 0 si todo ocurre bien, o -1 en caso de error.
