# Capitulo VIII: Desarrollo de API en Linux

## 8.1 Introducción

Los servidores y clientes dentro de una red TCP/IP necesitan utilizar los **sockets** para conectarse, enviar y recibir mensajes independientemente de la situación.

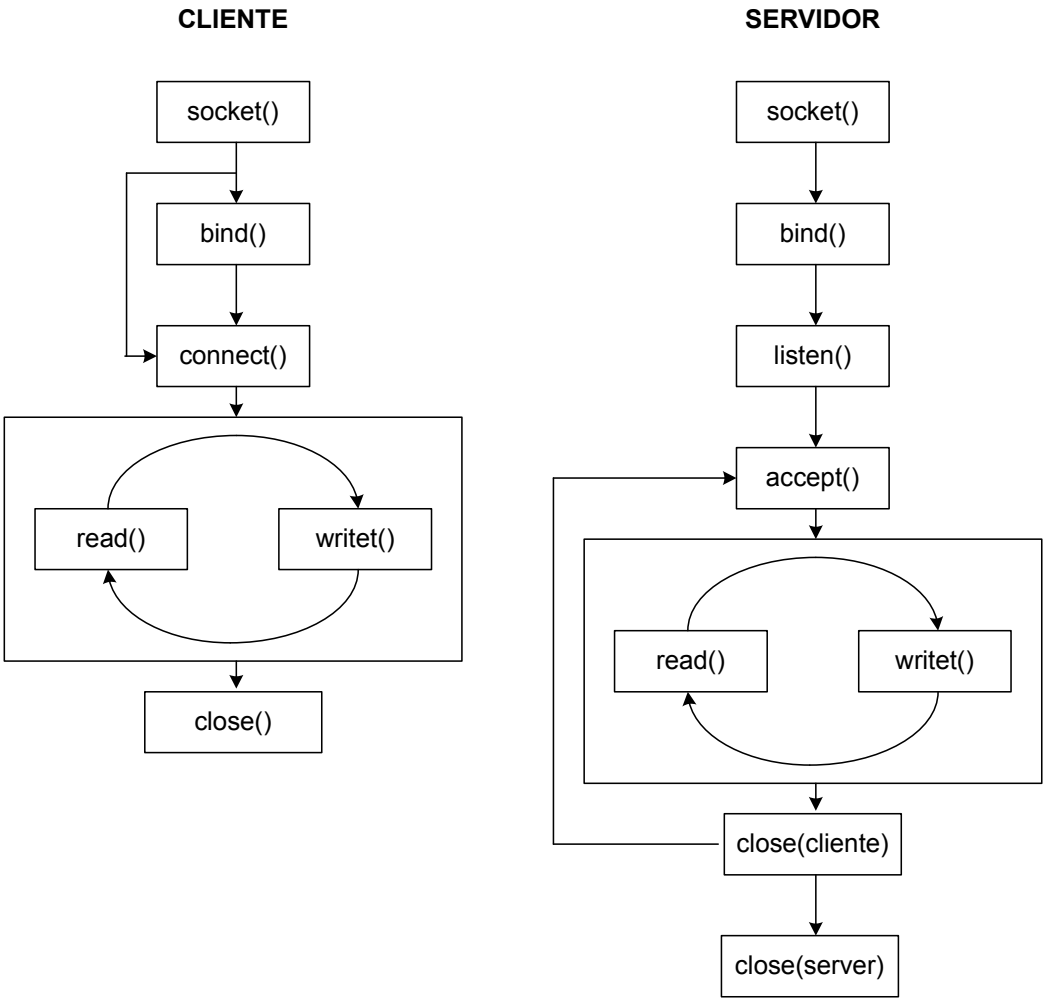


**Figura 8.1: interacción entre el cliente y el servidor**

El programa cliente puede seguir algunos pasos para comunicarse con un servidor. Se pueden cumplir todos los pasos o saltearse alguno de ellos, lo cual el sistema operativo permite salvar y realizar esta abstracción.

La unidad fundamental de toda la programación de red en Linux es el **socket**. De la misma forma que la E/S de archivos permite establecer conexiones con el sistema de archivo, los sockets constituyen un medio que el programa utiliza para direccionar, enviar y recibir mensajes. A

continuación se enumeraran los pasos que el cliente toma para conectarse con el servidor.

1. **Crear un Socket**: se selecciona de **diversos dominios** de red (por ejemplo, de Internet) y clases de sockets(**como flujo**).

2. **Configurar las opciones del sockets**(**opcional**). Se dispone de muchas opciones que afectan al comportamiento del sockets.

3. **Asociar una dirección IP a un puerto** (**opcional**). Si no se especifica, el TCP/IP asume se conectara con cualquier dirección IP y asume un puerto opcional.

4. **Conectarse con el servidor (opcional).** Se extiende y establece un canal bidireccional entre el programa local y otro programa de la red. Si se omite el programa utiliza una comunicación dirigida y sin conexión.

5. **Cerrar la conexión parcialmente** (**opcional**). Se restringe el canal de envió o de recepción.

6. **Enviar o recibir mensajes** (**opcional**). Una razón para prescindir de cualquier E/S podría incluir la comprobación de disponibilidad del servidor.

7. **Cerrar la conexión**. Por supuesto este paso es importante ya que los programas no cierran las conexiones terminadas, los programas que consumen CPU pueden agotar los descriptores de archivo y las llamadas del sistema (system call).

La programación de sockets se diferencia de la aplicación o herramienta de trabajo normal, en que trabajamos con programas y sistemas que funcionan concurrentemente. Esto implica que necesita conocer las sincronizaciones, la temporización y la administración de recursos. Los sockets enlazan tareas asíncronas con un único canal bidireccional. Esto podría conducir a problemas como el interbloqueo y la inanición.

## 8.2 Escucha del servidor: el algoritmo básico del cliente

La conexión ***Cliente-Socket*** mas simple es la que abre una conexión a un servidor, envía una consulta y acepta la respuesta. Algunos de los servicios estándar incluso no esperan consultas. Un ejemplo es el servicio de "***time of day***" hallado en el puerto 13. Desafortunadamente, muchas distribuciones de Linux no tienen ese servicio abierto sin que se revise el archivo **/etc/inetd.conf**. Si tiene acceso a una maquina puede interactuar y conectarse a ese puerto.

Un algoritmo básico de un cliente TCP debe cumplir con los siguientes pasos.

- **Paso 1**: Crear un Sockets.
- **Paso 2**: Crear una dirección de destino para el servidor.
- **Paso 3**: Conectar con el servidor.
- **Paso 4**: Leer y visualizar cualquier mensaje.

- **Paso 5**: Cerrar la conexión.

### *Paso 1. Crear un Sockets.*

La única herramienta que hace eficaz al receptor de mensajes y comienza el proceso completo de envió y recepción de mensajes de otra computadora es la llamada del sistema **socket()**.

Al igual que la llamada del sistema **open()** crea un descriptor para acceder a los archivos y dispositivos en nuestro sistema, **socket()** crea un descriptor para acceder a las computadoras de la red.

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

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

**Tabla 8.2.1: parámetros de la función socket()**

| Parámetro | Valor | Descripción |
|---|---|---|
| **domain** | **PF_INET** | Protocolo de Internet IPv4, pila TCP/IP. |
| | **PF_LOCAL** | Canales con nombres locales al estilo BSD. |
| | **PF_IPX** | Protocolos Novell. |
| | **PF_INET6** | Protocolo de Internet versión 6. Pila TCP/IP. |
| **Type** | **SOCK_STREAM** | Fiable, flujo de datos secuencial(flujo de bytes)(**TCP**). |
| | **SOCK_RDM** | Fiable, datos en paquetes (no implementado en muchos sistemas). |
| | **SOCK_DGRAM** | No fiable, datos en paquetes (**datagramas**)¨(**UDP**). |
| | **SOCK_RAW** | No fiable, datos en paquetes de bajo nivel. |
| **Protocol** | **0** | Soportado por la mayoría de las conexiones. |

A continuación se muestra un ejemplo para la llamada TCP/IP de generación de flujo, además de los archivos de cabecera más utilizados:

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

void main (void)
{
        int sd=0;

        sd=socket(AF_INET,SOCK_DGRAM,0);

        if (sd!=-1)

        ……continua mas abajo el código.
```

La **llamada devuelve un valor negativo cuando se produce un error** y coloca el código de error en **errno**(la variable global estándar para los errores de biblioteca).

La llamada del sistema **socket() solo crea las colas para el envió y recepción de los datos**, de forma contraria a la llamada del sistema para la apertura de archivos, la cual abre el archivo y lee el primer bloque. Solo cuando el programa ejecuta una llamada a la funciona **bind()**, el sistema operativo se conecta la cola a la red.

Si el programa no realiza explícitamente la llamada **bind()**, el sistema operativo realiza implícitamente la llamada.

### *Paso 2-3: Definir una dirección de destino para el servidor y Conectarse con el servidor.*

Después de la creación del socket se puede emplear la función **connect()** para interactuar con el servidor.

- Es preciso identificar al servidor de destino utilizando una dirección IP.
- La conexión ofrece el canal para los mensajes. Una vez que existe un canal a través del cual se puede intercambiar información.
- Es importante tener un camino de regreso para obtener los mensajes de vuelta. El servidor obtiene la dirección y el puerto de su cliente conectado para poder responder utilizando el camino similar.

  La función **connect()** presenta la siguiente sintaxis:

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

int connect(int sd, struct sockaddr_in *server, int addr_len)
```

  El significado de los argumentos se describe a continuación:

  - **sd**: es el descriptor de socket abierto cuando se definió el socket.
  - **sockaddr_in**: constituye un tipo de estructura para describir determinada información y esta compuesta por:

```
struct sockaddr_in
        {
                sa_family_t             sin_family;
                unsigned short int      sin_port;
                struct in_addr          sin_addr;
                unsigned char           __pad[]
        }
```

- **addr_len**: es la longitud de la estructura

***El tipo de dominio que se estableció en la llamada del sistema socket() debe ser el mismo valor que el primer campo de la familia sockaddr_in***.

En todos los casos, es mejor una iniciación a cero de la instancia entera de la estructura.

| Campo | Descripción | Orden de Bytes | Ejemplo |
|---|---|---|---|
| sin_family | Familia de protocolos | Host, nativo | AF_INET |
| sin_port | Nro de puerto del servidor | Red | 5000 |
| sin_addr | Direccion IP numerica del servidor | Red | 127.0.0.1(localhost) |

A continuación se muestra un ejemplo mostrando la iniciación e invocación de la llamada del sistema **connect():**

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

void main (void)
{
        int sd=0,cx=0;
        struct sockaddr_in bs;

        sd=socket(AF_INET,SOCK_DGRAM,0);

        if (sd!=-1)
        {
                bs.sin_family = AF_INET;
                bs.sin_port = htons (5000);
                bs.sin_addr.s_addr = htonl (127.0.0.1);

                cx=connect(sd,(struct sockaddr_in *)&bs, sizeof(bs));

                printf ("\n Servidor : %d %d \n",cx,sd);
        }
```

Después de que el programa establezca la conexión, el descriptor del socket **sd**, se convierte en un canal de lectura/escritura entre los dos programas.

### *Paso 4. Leer y visualizar cualquier mensaje Obtención de la respuesta del servidor*

El socket se abre y el canal se establece. Ahora, se puede obtener el primer **"hola"** desde el cliente. Algunos servidores inician la conversación como cuando la persona responde al teléfono. Una vez que la conexión esta abierta, se puede utilizar la biblioteca estándar de llamadas de E/S de bajo nivel para la comunicación. Para la lectura de los mensajes es posible

emplear las funciones **read()** y **recv()**, proporcionando la ultima un mejor control de los errores.

### *Llamada a la función read():*

```
#include<unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

Probablemente se encuentre familiarizado con esta llamada. Aparte de su capacidad especial para utilizar el descriptor de socket(sd) en lugar del descriptor de archivos(fd), todo lo demás es lo mismo que la lectura de un archivo. Puede utilizarlo de la siguiente manera:

```
#include <stdio.h>
#include <sys/socket.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include<unistd.h>

void main (void)
{
        int sd=0,cx=0, cant_leidos=0, cant_tx=0,
        struct sockaddr_in server, cliente;
        char recibo[255], envio[255]="Hola...";


        sd=socket(AF_INET,SOCK_STREAM,0);

        if (sd!=-1)
        {
                server.sin_family = AF_INET;
                server.sin_port = htons (5000);
                server.sin_addr.s_addr = htonl (127.0.0.1);

                cx=connect(sd,(struct sockaddr_in *)&server, sizeof(server));

                printf ("\n Servidor : %d %d \n",cx,sd);
        }

        while (1)
        {
        // Proceso de Escritura //

        write (sd,  envio, strlen (envio));

        cant_tx= sizeof(envio);

                if (cant_tx>0)
                {
                        printf ("\n enviados? : %d \n",tx);
                        printf ("\n mensaje? : %s \n",envio);

                        // Proceso de Lectura //
                        cant_tx=0;
                        cant_leidos=0;

                while (cant_leidos=0)
```

```
                    {
                        read(sd, recibo, MAXBUF);

                        cant_leidos= sizeof(recibo);

                        if (cant_leidos > 0)
                            {
                            printf ("\n bytes leidos: %d \n",cant_leidos);
                            printf ("\n mensaje recibido : %s \n", recibo);
                            }

                    }


                }
            }
            close (sd);
        }
```

La seguridad y fiabilidad son de extrema importancia cuando se crean los programas de red. Cuando se escriben programas hay que asegurarse que los buffers no se puedan desbordar y que todos los valores devueltos se comprueben.

La llamada del sistema **read()** no ofrece un control especial sobre la forma en que utiliza el **socket**. Es por ello que se ofrece otra llamada a la función **recvfrom()**.

### *Llamada a la función recvfrom():*

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

int recvfrom(int sd, void *buf, int len, unsigned int flags);
```

La llamada **recvfrom()** presenta a diferencia de la llamada **read()** señalizaciones que dan un mayor control sobre el que obtener, incluso ofrece control de flujo. Estos valores pueden agruparse con el operador O (FLAG 1 | FLAG 2 | …)

Cuando se **trabaja normalmente el parámetro de FLAG se establece en cero**. Las señalizaciones mas empleadas son:

- **MSG_OOB**: se procesan los datos fuera de banda.
- **MSG_PEEK**: se lee de forma no destructiva
- **MSG_WAITALL**: se utiliza para que no devuelva el mensaje hasta que el buffer suministrado este completo
- **MSG_DONTWAIT**: se utiliza para que el mensaje no se bloquee si la cola esta vacía.

La llamada del sistema **recv()** es mas flexible que **read()**, permitiendo el uso de distintas señalizaciones para modificar su comportamiento. Para

realizar una lectura normal desde un canal de socket(equivalente a read()), emplee el siguiente código:

**int bytes_read;**

**bytes_read = recv(sd, buffer, MAXBUF, 0),**

Para leer de forma no destructiva los datos fuera de banda del socket, haga lo siguiente:

**bytes_read = recv(sd, buffer, MAXBUF, MSG_PEEK),**

Para leer de forma no destructiva los datos fuera de banda del socket, haga lo siguiente:

**bytes_read = recv(sd, buffer, MAXBUF, MSG_OOB | MSG_PEEK);**

En el primer ejemplo se obtiene información del servidor proporcionando un buffer, un tamaño y sin señalización. En el segundo ejemplo muestra la información. Existe un flujo intencionad aquí: ¿Qué ocurre si el servidor envía mas información que la que pueda aceptar el buffer?. No es un error crítico, nada fallara. El algoritmo simplemente puede perder los datos que no se lean. Se puede emplear esta funcion de la siguiente forma.

```
#include <stdio.h>
#include <sys/socket.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include<unistd.h>

void main (void)
{
        int sd=0,cx=0, cant_leidos=0, cant_tx=0,
        struct sockaddr_in server, cliente;
        char recibo[255], envio[255]="Hola...";


        sd=socket(AF_INET,SOCK_DGRAM,0);

        if (sd!=-1)
        {
                server.sin_family = AF_INET;
                server.sin_port = htons (5000);
                server.sin_addr.s_addr = htonl (127.0.0.1);

                cx=connect(sd,(struct sockaddr_in *)&server, sizeof(server));

                printf ("\n Servidor : %d %d \n",cx,sd);
        }

        while (1)
        {
        // Proceso de Escritura //
```

```
sendto(sd, envio, strlen(envio)+1,0, (struct sockaddr_in *)&server, sizeof(server));

cant_tx= sizeof(envio);

        if (cant_tx>0)
        {
                printf ("\n enviados? : %d \n",tx);
                printf ("\n mensaje? : %s \n",envio);

                // Proceso de Lectura //
                cant_tx=0;
                cant_leidos=0;

        while (cant_leidos=0)
                {
                        recvfrom (sd,recibo, sizeof(recibo) ,0,(struct sockaddr_in
                        *)&cliente, sizeof(cliente));

                        cant_leidos= sizeof(recibo);

                        if (cant_leidos > 0)
                                {
                                printf ("\n bytes leidos: %d \n",cant_leidos);
                                printf ("\n mensaje recibido : %s \n", recibo);
                                }

                }


        }
        }
        close (sd);
}
```

### *Paso 5: cierre de la conexión*

Una vez que se obtuvo a información que necesitaba del servidor y que todo se realizo bien, se debe cerrar la conexión.

```
#include<unistd.h>

int close(int sd)
```

## 8.3 Ordenar bytes de red

Muchos **tipos distintos de computadoras** pueden residir en una red, pero es posible que no utilicen el mismo procesador. Todos los procesadores no almacenan sus números binarios de la misma forma. Las computadoras usan dos tipos de almacenamiento de números binarios básicos: **big endian** y **little endian**. Simplemente los **big endian** se leen de izquierda a derecha y los numero **litle endian** se leen de derecha a izquierda. Por ejemplo:

**Nro en hexadecimal: 0CC557B3**

**Big Endian: 0C C5 57 B3**

**Litle endian: B3 57 C5 0C**

Para que las **computadoras** de una **red heterogénea** se comuniquen eficientemente, tienen que usar binario y deben establecer un **endianness**. Los endiannes de una computadora es el orden de bytes enviados por el host. Los endiannes de una red se denominan orden de bytes de red. ***El orden de bytes debe siempre ser big endian*** en una red de datos.

En el código se rellenan tres campos: **sin_family** (es un campo ordenado de bytes a host), mientras que **sin_port** y **sin_addr** requieren transformación de acuerdo a la siguiente tabla.

| Llamada | Significado | Descripción |
|---|---|---|
| **htons()** | De host a red corto | Convierte 16 bits a big endian. Se emplea en la transmisión. |
| **htonl()** | De host a red largo | Convierte 32 bits a big endian. Se emplea en la transmisión. |
| **ntohs()** | De red a host corto | Convierte 16 bits a formato del host. Se emplea en la recepción. |
| **ntol()** | De red a host largo | Convierte 32 bits a formato del host. Se emplea en la recepción. |

También se convierte la dirección IP ASCII al binario equivalente mediante la utilización de la función **inet_aton()**.

| Llamada | Descripción |
|---|---|
| **inet_aton()** | Transforma la notación punto (X.Y.Z.T) al binario ordenado en red. Devuelve cero si falla y distinto de cero si al dirección es valida. |
| **inet_ntoa()** | Transforma un binario IP ordenado en red a un ASCII en notación decimal. |
| **gethostbyname()** | Pregunta a un servidor de nombres para transformar el nombre a una dirección IP de 32 bits. |
| **getsrvbyname()** | Obtiene el puerto y el protocolo asociado a un servicio del archivo /etc/services. |

## 8.4 Envío de llamadas entre pares

Se puede pensar en el pase de mensajes desde dos puntos de vistas distintos: **continuo y sin interrupción**(TCP) o **paquetes discontinuos de información**(UDP). Haciendo una analogía podemos decir que el flujo de datos continuo es como una transmisión telefónica y el paquete discontinuo es como una carta en un sobre con dirección.

El flujo continuo requiere que se tenga una conexión establecida con el destino. Esto asegura que la información no se perderá durante el intercambio y que esta ordenada cuando llega. Los mensajes discontinuos permiten que se realice una conexión para facilitar simplemente la programación. Sin una conexión, el programa tiene que colocar una dirección en cada mensaje.

Es posible utilizar la llamada del sistema **connect()** en un **socket** UDP. Puede ser atractivo cuando no necesita E/S de alto nivel, ya que UDP ofrece

una ayuda de rendimiento con los mensajes autocontenidos. Sin embargo, la conexión UDP funciona algo diferente a como TCP gestiona las conexiones.

La llamada del sistema **connect()** en una conexión UDP registra simplemente el destino de cualquier mensaje enviado. Puede utilizar read() o write() como una conexión TCP, pero no tendrá garantías de fiabilidad y ordenación.

Normalmente con UDP se utiliza la llamada del sistema **sendto()** o **recvfrom()**. El **servidor que espera** la conexión puede utilizar la misma interfaz de conexión o puede utilizar **sendto()** y **recvfrom()**. Para que el programa cliente se conecte con el equipo servidor, se necesita que este publique su numero de puerto con la llamada **bind().**

El número de puerto de destino es una interfaz de puerto acordado entre los dos programas. El servidor que escucha activa el puerto a través de la llamada **bind()** para solicitar DEST_PORT. Cuando el mensaje llega el destino puede emitir una llamada del sistema **connect()** el mismo.

## 8.5 Sockets sin conexión

La ***interfaz sin conexión no realiza la llamada del sistema connect()***, pero no se puede llamar a send() o recv() sin tener una conexión. De echo el sistema operativo ofrece dos llamadas del sistema de bajo nivel que incluyen la dirección de destino: **recvfrom()** y **sendto()**.

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

int sendto (int sd, char *envio, int long_msg, int opciones, struct sockaddr *addr, int addr_len);

int recvfrom (int sd, char *recibo, int maxsize, int opciones, struct sockaddr *addr, int *addr_len);
```

La llamada de la función sendto() añade la dirección del socket del destino. Al enviar un mensaje, rellena la estructura **addr** y llama a **sendto()**.

El ultimo parámetro tiene un tipo distinto. Es un puntero a un entero. Los dos últimos parámetros de **sendto()** son el destino. Los dos últimos parámetros de **recvform()** son la dirección del origen. Ya que la familia de estructura **sockaddr** pueden ser de distintos tamaños, puede recoger posiblemente un mensaje del origen que es distinto del tipo de socket (por omisión AF_INET).

Puesto que recvfrom() puede cambiar el parámetro addr_len, necesita establecer el valor cada vez que realice la llamada del sistema. De otra manera el valor en addr_len puede disminuir potencialmente con cada llamada del sistema.

Si utiliza la llamada del sistema **recvfrom()**, debe suministrar un valor **addr** y **addr_len**, no puede establecerlos a NULL(0). Puesto que UDP es sin conexión, necesita conocer el origen de la consulta. La forma más fácil de hacer eso es guardar intacto las variable tipo **sockaddr recibida del**

**cliente,** mientras se procesa la consulta. Por ejemplo, al recibir información de un cliente particular se pueden mostrar estos datos:

```
recvfrom (sd,recibo, sizeof(recibo) ,0,(struct sockaddr_in *)&cliente, sizeof(cliente));

printf ("\n Familia de este Cliente: %s \n", cliente.sin_family);
printf ("\n Puerto de este Cliente: %s \n", ntohs(cliente.sin_port));
printf ("\n Dirección IP de este Cliente: %s \n", ntohl(cliente.sin_port));
```

## 8.6 Asociación del puerto al socket

Puede requerir un numero de puerto especifico desde el sistema operativo utilizando la llamada del sistema **bind()**.

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

int bind (int sd, struct sockaddr *addr, int addr_size);
```

Ejemplo uso de la función en una porción de código de un programa servidor.

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

void main (void)
{
        int sdsrv, txsrv=0,rxsrv=0,bd=0, ac=0, tx=0, longrx=0;
        struct sockaddr_in srv, bs;
        char recibo[255];
        char envio[255]="Mundo...";

        sdsrv=socket(AF_INET,SOCK_DGRAM,0);

        if (sdsrv!=-1)
        {
                srv.sin_family = AF_INET;
                srv.sin_port = htons (5000);
                srv.sin_addr.s_addr = htonl (INADDR_ANY);

                bd=bind (sdsrv,(struct sockaddr_in *)&srv, sizeof(srv));
```

**……continua con el programa**

## 8.7 Como enviar el mensaje

El emisor es el primero en enviar la información. El receptor debe tener una localización y un numero de puerto conocido, establecido con la llamada del sistema **bind()**. El emisor no necesita establecer su número de puerto, porque cada mensaje incluye la dirección de retorno.

El emisor realiza la inicialización y a menudo solo una petición de mensaje. Puede colocar este código en un bucle para lanzar mensajes de una

parte a otra (cliente y servidor). Es una porcion de codigo que normalmente ve del lado del servidor.

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

void main (void)
{
        int sdsrv,txsrv=0,rxsrv=0,bd=0, ac=0, tx=0, longrx=0;
        struct sockaddr_in srv, bs;
        char recibo[255];
        char envio[255]="Mundo...";

        sdsrv=socket(AF_INET,SOCK_DGRAM,0);

        if (sdsrv!=-1)
        {
                srv.sin_family = AF_INET;
                srv.sin_port = htons (5000);
                srv.sin_addr.s_addr = htonl (INADDR_ANY);

                bd=bind (sdsrv,(struct sockaddr_in *)&srv, sizeof(srv));

                printf ("\n Servidor : %d \n",bd);
        }

                listen (sdsrv,5);

        while (1)
        {

                ac=accept (sdsrv,(struct sockaddr_in *)&bs, sizeof(bs));

                printf ("\n Accept : %d \n",ac);

                if (ac==-1)
                {
                while (rxsrv==0)
                {

                        rxsrv=recvfrom (sdsrv,recibo, sizeof(recibo),0,(struct sockaddr_in *)&bs,
                        sizeof(bs));

                        printf ("\n mensaje recibido? : %s \n",recibo);

                        longrx=strlen(recibo);

                        printf ("\n caracteres recibidos? : %d \n",longrx);

                        printf ("\n Que devuelve recvfrom ? : %d \n",rxsrv);

                        if (rxsrv>0)
                        {
                        tx=sendto(sdsrv,envio,sizeof(envio)+1,0,(struct sockaddr_in *)&bs,
                        sizeof(bs));
                        }
                        rxsrv=0;
```

```
                }
        }
}
return (0);
}
```

## 8.8 Recepción del mensaje

El receptor debe enlazarse con el número de puerto especificado por el emisor. Después de la creación del **socket()** se debe publicar el número de puerto acordado, para que exista comunicación entre ambos pares.

En este extracto, el receptor solicita un puerto y luego espera la llegada de los mensajes. Si obtiene un mensaje, procesa la petición y responde con el resultado.

El emisor transmite solamente un mensaje. El receptor, por otra parte, actúa como un servidor, recibiendo, procesando y respondiendo todos los mensajes que llegan. Esto presenta un problema fundamental entre estos dos algoritmos: la asignación de espacio suficiente para el mensaje entero.

Si el programa requiere un mensaje de longitud variable, puede que necesite crear un buffer grande. El mensaje UDP mas grande esta sobre 64 KB, este es finalmente su tamaño mas grande.

## 8.9 Ejemplo código fuente del cliente

```
#include <stdio.h>
#include <sys/socket.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include<unistd.h>

void main (void)
{
        int sd=0,cx=0, cant_leidos=0, cant_tx=0,
        struct sockaddr_in server, cliente;
        char recibo[255], envio[255]="Hola...";


        sd=socket(AF_INET,SOCK_DGRAM,0);

        if (sd!=-1)
        {
                server.sin_family = AF_INET;
                server.sin_port = htons (5000);
                server.sin_addr.s_addr = htonl (127.0.0.1);

                cx=connect(sd,(struct sockaddr_in *)&server, sizeof(server));

                printf ("\n Servidor : %d %d \n",cx,sd);
        }
```

```
            while (1)
            {
            // Proceso de Escritura //

            sendto(sd, envio, strlen(envio)+1,0, (struct sockaddr_in *)&server,
sizeof(server));

            cant_tx= sizeof(envio);

                  if (cant_tx>0)
                  {
                        printf ("\n enviados? : %d \n",tx);
                        printf ("\n mensaje? : %s \n",envio);

                        // Proceso de Lectura //
                        cant_tx=0;
                        cant_leidos=0;

                  while (cant_leidos=0)
                        {
                              recvfrom (sd,recibo, sizeof(recibo) ,0,(struct sockaddr_in
                              *)&cliente, sizeof(cliente));

                              cant_leidos= sizeof(recibo);

                              if (cant_leidos > 0)
                                    {
                                    printf ("\n bytes leidos: %d \n",cant_leidos);
                                    printf ("\n mensaje recibido : %s \n", recibo);
                                    }

                        }


                  }
            }
            close (sd);
      }
```

## 8.10 Ejemplo código fuente del servidor

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

      void main (void)
      {
            int sdsrv,txsrv=0,rxsrv=0,bd=0, ac=0, tx=0, longrx=0;
            struct sockaddr_in srv, bs;
            char recibo[255];
```

```
        char envio[255]="Mundo...";

        sdsrv=socket(AF_INET,SOCK_DGRAM,0);

        if (sdsrv!=-1)
        {
                srv.sin_family = AF_INET;
                srv.sin_port = htons (5000);
                srv.sin_addr.s_addr = htonl (INADDR_ANY);

                bd=bind (sdsrv,(struct sockaddr_in *)&srv, sizeof(srv));

                printf ("\n Servidor : %d \n",bd);
        }

                listen (sdsrv,5);

        while (1)
        {

                ac=accept (sdsrv,(struct sockaddr_in *)&bs, sizeof(bs));

                printf ("\n Accept : %d \n",ac);

                if (ac==-1)
                {
                while (rxsrv==0)
                {

                        rxsrv=recvfrom (sdsrv,recibo, sizeof(recibo),0,(struct sockaddr_in
*)&bs, sizeof(bs));

                        printf ("\n mensaje recibido? : %s \n",recibo);

                        longrx=strlen(recibo);

                        printf ("\n caracteres recibidos? : %d \n",longrx);


                        printf ("\n Que devuelve recvfrom ? : %d \n",rxsrv);

                        if (rxsrv>0)
                        {
                        tx=sendto(sdsrv,envio,sizeof(envio)+1,0,(struct sockaddr_in
*)&bs, sizeof(bs));
                        }

                        rxsrv=0;

                }
        }
}
return (0);
```

```
}
```