Introducción a Sockets en Linux

Horacio Goetendía Bonilla

6 de Enero de 2003

Índice

1. Conceptos

1.1. Socket

Un *socket* es una manera de hablar con otra computadora usando descriptores de archivo standars de Unix.

Un *socket* es un dispositivo de comunicación bidireccional entre procesos, las cuales pueden ser procesos de una misma computadora o procesos de una computadora con procesos de otras. "*sockets* permite la comunicación entre procesos".

En Unix, todas las acciones de entrada y de salida son realizadas escribiendo o leyendo en uno de estos descriptores de archivos, los cuales son simplemente un número entero asociado a un archivo abierto que puede ser una conexión de red, una terminal, o cualquier otra [1].

Un *socket* en Unix crea un extremo de una comunicación y devuelve un descriptor.

Cualquiera puede obtener una pagina web usando el comando `telnet` porque ambos (`telnet` y el servidor web) usan *sockets* para la comunicación en red.

2. *Sockets* en Unix standar

La creación de un *socket* implica la especificación de tres parámetros: Namespace, estilo de comunicación y el protocolo (`socket(int dominio, int tipo, int protocolo)`)las cuales su explicación nos ayudará a entender los conceptos y usos de los *sockets*.

2.1. Namespace (`int dominio`) *(dominio de comunicaciones)*

Este parámetro especifica como es escrita la dirección del *socket*. Una dirección de socket identifica un final de una conexión de *socket*. Por ejemplo, una dirección de *socket* en un dominio local (*local namespace*) es un archivo ordinario mientras que un dominio de internet (*internet namespace*) esta compuesta por la dirección de internet o la llamada dirección IP.

En síntesis este parámetro selecciona un dominio de comunicaciones es decir selecciona la familia de protocolo que se usará para la comunicación. Estas familias se determinan en `<sys/socket.h>`.

En Unix están definidas constantes para la invocación del dominio de comunicaciones, en la programación de *sockets* estas comienzan con `PF_` [2].

A continuación se lista los dominios de comunicaciones:

[1] En Unix todo es un archivo

[2] PF = Protocol Family

Nombre	Propósito	Pág.Man.
PF_UNIX,PF_LOCAL	Comunicación local	unix(7)
PF_INET	Protocolos de Internet IPv4	ip(7)
PF_INET6	Protocolos de Internet IPv6	
PF_IPX	Protocolos IPX - Novell	
PF_NETLINK	Dispositivo de la interfaz de usuario del núcleo	netlink(7)
PF_X25	Protocolo ITU-T X.25 / ISO-8208	x25(7)
PF_AX25	Protocolo AX.25 de radio para aficionados	
PF_ATMPVC	Acceso directo a PVCs ATM	
PF_APPLETALK	Appletalk	ddp(7)
PF_PACKET	Interfaz de paquetes de bajo nivel	packet(7)

Cuadro 1: Constantes sobre dominios de comunicaciones

2.1.1. Localnamespace *(dominio Local)*

Las constantes que son más usadas en esta parte son:
PF_UNIX ó PF_LOCAL las cuales fueron detalladas en la tabla anterior.

2.1.2. Internetnamespace *(dominio de internet o dirección IP)*

Las constantes que son más usadas son las PF_INET la cual esta detallada en la tabla anterior.

2.2. Estilo de comunicación `(int tipo)`

El estilo de comunicación es el parámetro para especificar como el *socket* va a controlar la data a ser transmitida y el número de pares de comunicación. En síntesis el estilo de comunicación determina el manejo de los paquetes a ser transmitidos y como estos son direccionados del emisor al receptor. En este argumento se especifica la semántica de la comunicación.

En Unix están definidas constantes para la invocación del estilo de comunicaciones, en la programación de *sockets* estas comienzan con `SOCK_` [3].

A continuación se lista los estilos de comunicación:

SOCK_STREAM Proporciona flujos de bytes basados en una conexión bidireccional secuenciada, confiable. Se puede admitir un mecanismo de transmisión de datos fuera-de-banda.

SOCK_DGRAM Admite datagramas (mensajes no confiables, sin conexión, de una longitud máxima fija).

[3] SOCK = Socket

SOCK_SEQPACKET Proporciona un camino de transmisión de datos basado en conexión bidireccional secuenciado, confiable, para datagramas de longitud máxima fija; se requiere un consumidor para leer un paquete entero con cada llamada al sistema de lectura.

SOCK_RAW Proporciona acceso directo a los protocolos de red.

SOCK_RDM Proporciona una capa de datagramas fiables que no garantiza el orden.

SOCK_PACKET Obsoleto y no debería utilizarse en programas nuevos. Vea packet(7).

Algunos tipos de conectores pueden no ser implementados por todas las familias de protocolos. Por ejemplo, SOCK_SEQPACKET no está implementado para AF_INET.

2.2.1. Estilo de comunicación tipo conexión *(sockets de flujo)*

Este estilo *SOCK_STREAM* usan TCP/IP. Si en este tipo de *socket* enviáramos 3 datos por ejemplo A, B y C estos llegarán al destino en el mismo orden A, B y C. Garantiza la llegada de todos los paquetes del emisor al receptor en el orden en que fueron enviados, si los paquetes se pierden se alteran o desordenan el receptor inmediatamente pide una retransmisión.

Una analogía de este estilo es la comunicación tipo teléfono en donde la dirección del emisor y receptor son establecidas al inicio de la comunicación, es decir cuando la conexión es establecida [4]. Algo más directo es que podríamos asumiríamos que es una conexión tipo tubería [5].

2.2.2. Estilo de comunicación tipo datagrama *(sockets de datagramas)*

Este estilo *SOCK_DGRAM* usan UDP (User Datagram Protocol) es decir que admite datagramas. Este estilo no necesita una conexión accesible como los sockets de flujo, este estilo construye un paquete de datos con información sobre su destino y se lo enviará afuera sin la necesidad de una conexión. No se garantiza el orden de llegada, los paquetes en el trayecto pueden alterarse, desordenarse o perderse solo se garantiza el mejor esfuerzo en la transmisión.

[4] Un conector de flujo debe estar en un estado conectado antes de que cualquier dato pueda ser enviado o recibido en él.

[5] Se crea una conexión con otro conector mediante la llamada `connect(2)` que secciones adelante se explicará

Protocolo	Número	Alias	Descripción
ip	0	IP	internet protocol, pseudo protocol number
icmp	1	ICMP	internet control message protocol
igmp	2	IGMP	internet group management protocol
ggp	3	GGP	gateway-gateway protocol
ipencap	4	IP-ENCAP	IP encapsulated in IP (officially IP)
st	5	ST	ST datagram mode
tcp	6	TCP	transmission control protocol

Cuadro 2: Extracto de equivalentes numéricos de los protocolos manejados

2.3. Protocolo (`int protocolo`)

El protocolo especifica un protocolo particular para ser usado con el conector. Normalmente sólo existe un protocolo que admita un tipo particular de conector dentro de una familia de protocolos dada. Sin embargo, es posible que puedan existir varios protocolos, en cuyo caso un protocolo particular puede especificarse de esta manera. El número de protocolo a emplear es específico al "dominio de comunicación"en el que la comunicación va a tener lugar; vea `protocols(5)`. Consulte `getprotoent(3)` para ver cómo asociar una cadenas con el nombre de un protocolo a un número de protocolo.

En síntesis cada protocolo es valido para cualquier tipo de dominio de comunicaciones *namespace* pero hay uno en especial que es el mejor cuando se usa cierto dominio de comunicaciones*namespace*. Para este argumento no existe constantes, su uso es directo, es decir se usa equivalentes numéricos para asociar con el protocolo correspondiente [6] *Ver Cuadro 2*

3. Conociendo el modelo de comunicación

La creación y destrucción de *sockets* se realiza con las funciones `socket` y `close` correspondientemente, cuando se crea un *socket* se especifica:

```
int socket (int dominio, int tipo, int protocolo)
```

los cuales ya vimos en la sección anterior.

Ejemplo:

```
socket (PF_LOCAL,SOCK_STREAM,0)
```

El 0 en el tercer argumento usualmente es el protocolo correcto.

Si la creación del socket es realizada sin problema alguno devuelve un descriptor de archivo para el *socket*, este descriptor de archivo es que es un número entero declarado con anterioridad en el programa.

[6] Una lista completa de los equivalentes numéricos de los protocolos al final.

Ejemplo:

```
FDesciptor=socket (PF_LOCAL,SOCK_STREAM,0)
```

Se puede leer o escribir al *socket* usando las funciones `read` o `write` respectivamente.

Con la función `close` destruyes el *socket* [7].

Ejemplo:

```
close (FDescriptor)
```

Identifiquemos los agentes que intervienen en una comunicación usando *sockets*:

3.1. El Cliente

El cliente es aquel que establece la conexión de un *socket* local a un *socket* servidor con la función `connect`, especificando un puntero a la estructura de dirección del servidor de socket como segundo argumento de la función `connect`, esperando que el servidor acepte la conexión.

Como tercer argumento especificamos la longitud, en bytes, del puntero a la estructura de direcciones que fué especificado como segundo argumento[8]

Como primer argumento tenemos el descriptor de archivos, resultado de la creación del *socket*.

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

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

3.1.1. Primer argumento (`int sockfd`)

El primer argumento es el descriptor de archivos, resultado de la creación del *socket*.

3.1.2. Segundo argumento (`struct sockaddr *serv_addr`)

El segundo argumento es un puntero a la estructura de dirección del servidor de *sockets* la cual contiene la dirección IP destino y el puerto.

La estructura `sockaddr` esta definida en la cabecera `<sys/socket.h>` El programa cliente se muestra en el cuadro 3.

[7] La función `close` cierra un descriptor de fichero

[8] El formato de La dirección del *socket* varía acorde el namespace o dominio del *socket*

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
/* netbd.h es necesitada por la estructura hostent */
#define PORT 3550 /* El Puerto Abierto del nodo remoto */
#define MAXDATASIZE 100 /* El número máximo de datos en bytes */
int main(int argc, char *argv[])
{
      int fd, numbytes; /* archivos descriptores */
      char buf[MAXDATASIZE];
      /* en donde es almacenará el texto recibido */
      struct hostent *he;
      /* estructura que recibirá información sobre el nodo remoto */
      struct sockaddr_in server;
      /* información sobre la dirección del servidor */

      if (argc !=2)
      { /* esto es porque nuestro programa sólo necesitará un argumento, (la IP) */
      printf(Üsage: %s <IP Address>",argv[0]); exit(-1);
      }
      if ((he=gethostbyname(argv[1]))==NULL)
      { /* llamada a gethostbyname() */
      printf("gethostbyname() error"); exit(-1);
      }
      if ((fd=socket(AF_INET, SOCK_STREAM, 0))==-1)
      { /* llamada a socket() */
      printf("socket() error"); exit(-1);
      }
      server.sin_family = AF_INET; server.sin_port = htons(PORT);
      /* htons() es necesaria nuevamente*/
      server.sin_addr = *((struct in_addr *)he->h_addr);
      /*he->h_addr pasa la información de *he a "h_addr"*/
      bzero(&(server.sin_zero),8);
      if(connect(fd, (struct sockaddr *)&server, sizeof(struct sockaddr))==-1)
      { /* llamada a connect() */
      printf(çonnect() error"); exit(-1);
      }
      if ((numbytes=recv(fd,buf,MAXDATASIZE,0)) == -1)
      { /* llamada a recv() */
      printf(ïecv() error"); exit(1);
      }
      buf[numbytes]= '0';
      printf("Server Message: %s",buf);
      /* muestra el mensaje de bienvenida del servidor */
      close(fd); /* cerramos fd */
}
```

Cuadro 3: Programa Servidor

3.2. El Servidor

El ciclo de vida de un servidor consiste en la creación de un estilo de conexión del *socket* enlazando una dirección al *socket*. En el cuadro 4 se muestra el programa servidor.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define PORT 3550 /* El puerto que será abierto */
#define BACKLOG 2 /* El número de conexiones permitidas */
main()
{
     int fd, fd2;
    /* los archivos descriptores */
     struct sockaddr_in server;
    /* para la información de la dirección del servidor */
     struct sockaddr_in client;

    /* para la información de la dirección del cliente */
     int sin_size; /* A continuación la llamada a socket() */

     if ((fd=socket(AF_INET, SOCK_STREAM, 0)) == -1 )
        { printf("error en socket()"); exit(-1); }
      server.sin_family = AF_INET; server.sin_port = htons(PORT);
      server.sin_addr.s_addr = INADDR_ANY;

      /* INADDR_ANY coloca nuestra dirección IP automáticamente */
      bzero(&(server.sin_zero),8);

      /* escribimos ceros en el resto de la estructura */
      /* A continuación la llamada a bind() */
      if(bind(fd,(struct sockaddr*)&server, sizeof(struct sockaddr))==-1)
       {
       printf("error en bind() ");
        exit(-1);
        }
      if(listen(fd,BACKLOG) == -1)
       { /* llamada a listen() */ printf("error en listen()"); exit(-1);}
      while(1)
      {
       sin_size=sizeof(struct sockaddr_in);
       /* A continuación la llamada a accept() */
       if ((fd2 = accept(fd,(struct sockaddr *)&client, &sin_size))==-1)
       {printf("error en accept()"); exit(-1);}
       printf("You got a connection from%s", inet_ntoa(client.sin_addr) );
       /* que mostrará la IP del cliente */
       send(fd2,"Bienvenido a mi servidor.",25,0);
       /* que enviará el mensaje de bienvenida al cliente */
       close(fd2); /* cierra a fd2 */
      }
}
```

Cuadro 4: Programa Servidor

Referencias

[1] Lawrence Besaw : *Berkeley UNIX System Call and Interprocess Comunication: BSD Socket References*

BSD, 1987

[2] José Miguel Alonso : *TCP/IP en UNIX Programación de aplicaciones distribuidas*

RAMA, 2000

[3] Sandra Loosemore : *The GNU C Library Reference Manual*

Free Software Foundation, 1999

[4] Mark Mitchell, Jeffrey Oldham, Alex Samuel : *Advanced Linux Programming*

New Riders, 2001

[5] Kay A. Robbins, Steven Robbins : *Unix Programación Práctica*

Prentice Hall, Primera Edición

[6] Kurt Wall : *Programación en Linux con ejemplos*

Prentice Hall, 2000

[7] Vic Metcalfe: *Programming Unix Sockets in C FAQ*

Frequently Asked Questions, 1998

[8] Jim Frost: *BSD Sockets a quick and dirty primer*

Frequently Asked Questions, 1990