SOCKETS

Interfaz de Programación de Aplicaciones para la Familia de Protocolos TCP/IP

INTRODUCCIÓN

Debido a su ubicuidad, la familia de protocolos TCP/IP se considera como un estándar de facto, sin embargo no ocurre lo mismo con las interfaces de programación disponibles para desarrollar aplicaciones en red.

En general, cada tipo de sistema operativo provee una API de comunicación TCP/IP. Tales interfaces presentan aspectos comunes derivados principalmente de la interfaz de comunicación de los sistemas operativos tipo Unix (Berkeley Sockets).

Por ejemplo, Microsoft Windows incluye una interfaz de programación denominada Windows Sockets 2 basada en el paradigma popularizado por los Berkeley Sockets. Las diferencias en su arquitectura en relación a éstos últimos son reflejo del Modelo de Arquitectura de Sistema Abierto de Windows (WOSA, Windows Open System Architecture). Por su parte, Java también ofrece una interfaz de programación TCP/IP que incluye la orientación a objetos y portabilidad que le caracteriza.

Este documento sólo presenta información relacionada con los Berkeley Sockets.

BERKELEY SOCKETS

El subsistema de entrada/salida de Unix, en general, sigue el paradigma denominado *abrir – leer – escribir – cerrar*. Sin embargo, el manejo de entrada/salida de red involucra más detalles y opciones. Por ejemplo, entre los los aspectos a considerar se encuentran:

- La relación cliente/servidor es asimétrica. La aplicación requiere saber cuál será su rol (cliente o servidor) antes de inciar la comunicación.
- La entrada/salida de red puede trabajar orientada a conexión o sin conexión.
- En un entorno de red los nombres son más importantes que en el manejo de archivos o dispositivos.
- Para especificar una conexión de red se requieren más parámetros que para abrir un archivo o dispositivo:

```
{protocolo, dirección_local, proceso_local,
 dirección_remota, proceso_remoto }
```

- La interfaz de red debe soportar múltiples protocolos.

En vista de estás necesidades, cuando se añadieron los protocolos de red a BSD Unix se decidió que, como éstos eran más complejos que los dispositivos convencionales de E/S, su interacción con los procesos del usuario debía ser más compleja. En particular, la interfaz de protocolo debía permitir a los programadores crear un código de servidor que esperara las conexiones pasivamente, así como también, un código cliente realizara activamente las conexiones. Además, debía soportar que los programadores de aplicación envíasen datagramas especificando la dirección de destino junto con cada datagrama en lugar de destinos enlazados en el momento en que llamaban a `open`. Para manejar todos estos casos, los diseñadores eligieron abandonar el paradigma tradicional de *abrir – leer – escribir – cerrar* y adicionar algunas llamadas nuevas al sistema operativo, así como también una nueva biblioteca de funciones.

La adición de protocolos de red a UNIX incrementó sustancialmente la complejidad de la interfaz de E/S, pues los diseñadores intentaron construir un mecanismo general para incluir otros protocolos diferentes a TCP/IP. Como

consecuencia, la aplicación no puede proporcionar una dirección de 32 bits y esperar a que el sistema operativo la interprete de manera correcta. La aplicación debe especificar explícitamente que el número de 32 bits representa una dirección IP.

El subsistema de E/S de red de UNIX, principlamente se centra en una abstracción conocida como sócalo (`socket`). Pensamos al socket como una generalización del mecanismo de acceso a archivos de UNIX que proporciona un punto final para la comunicación. Similar al acceso a archivos, los programas de aplicación requieren que el sistema operativo cree un socket cuando se necesita. El sistema devuelve un `short integer` que utiliza el programa de aplicación para hacer referencia al socket creado. La diferencia principal entre los descriptores de archivos y los descriptores de sockets es que el sistema operativo enlaza un descriptor de archivo a un archivo o dispositivo específico cuando se llama a `open`, pero se pueden crear sockets sin enlazarlos a direcciones de destino específicas. La aplicación puede elegir especificar una dirección de destino cada vez que utiliza el socket (es decir, cuando se envían datagramas empleando UDP) o elegir enlazar la dirección destino a un socket y evadir su especificación repetidamente (es decir, cuando se hace una conexión TCP).

La figura 1 ilustra la secuencia de llamadas del sistema que se suceden en una transferencia orientada a conexión (se inicia la ejecución del servidor y posteriormente un cliente establece una conexión).

La comunicación entre un cliente y un servidor empleando un protocolo no orientado a conexión es diferente. La figura 2 muestra la sequencia de llamadas que se suceden en este caso.

En este último caso, el cliente no establece una conexión con el servidor, sino que le envía datagramas mediante la llamada del sistema `sendto`, la cual requiere como parámetro la dirección destino (servidor). Por su parte, el servidor sólo tiene que invocar una llamada `recvfrom` que espera hasta que arriva data de algún cliente. `recvfrom` devuelve la dirección del proceso cliente junto con el datagrama de forma tal que el servidor puede enviar su respuesta al proceso correcto.

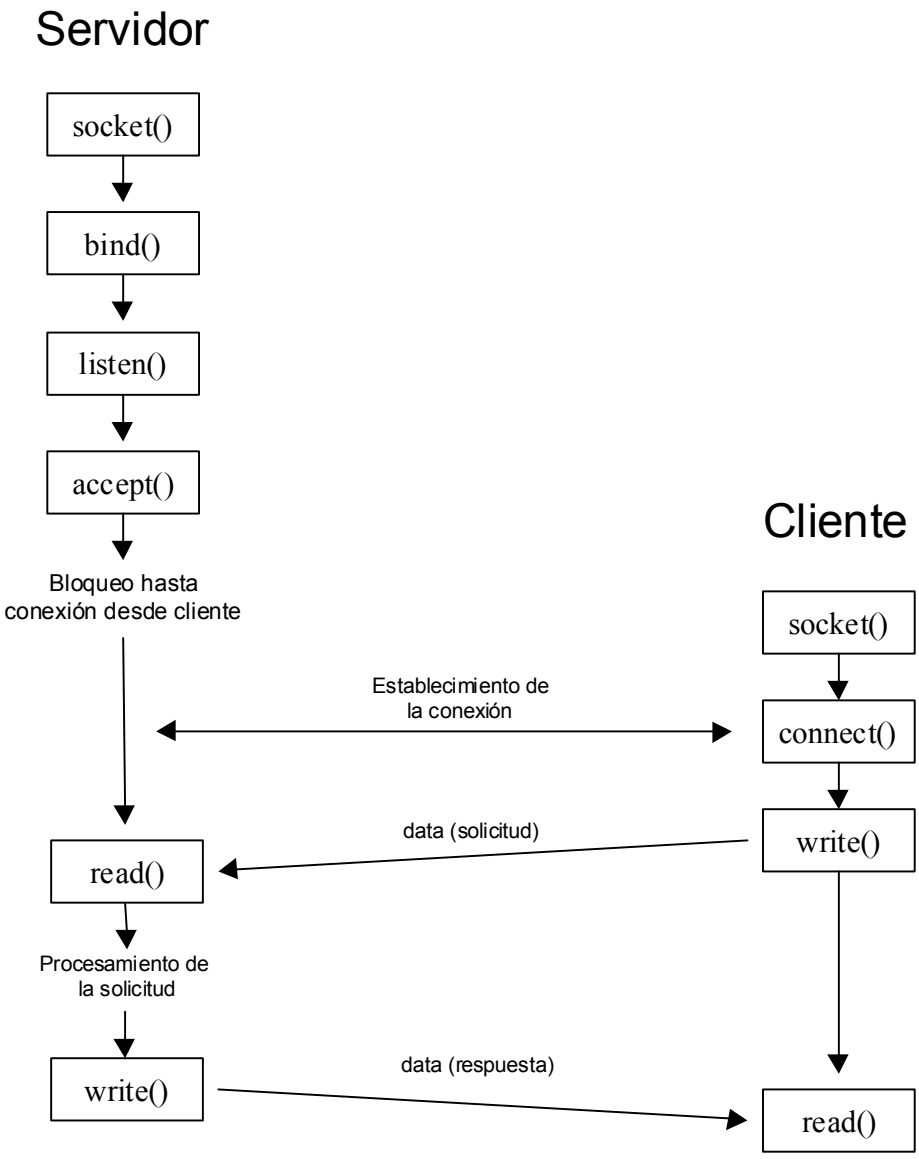

Figura 1. Llamadas al sistema para un protocolo orientado a conexión.

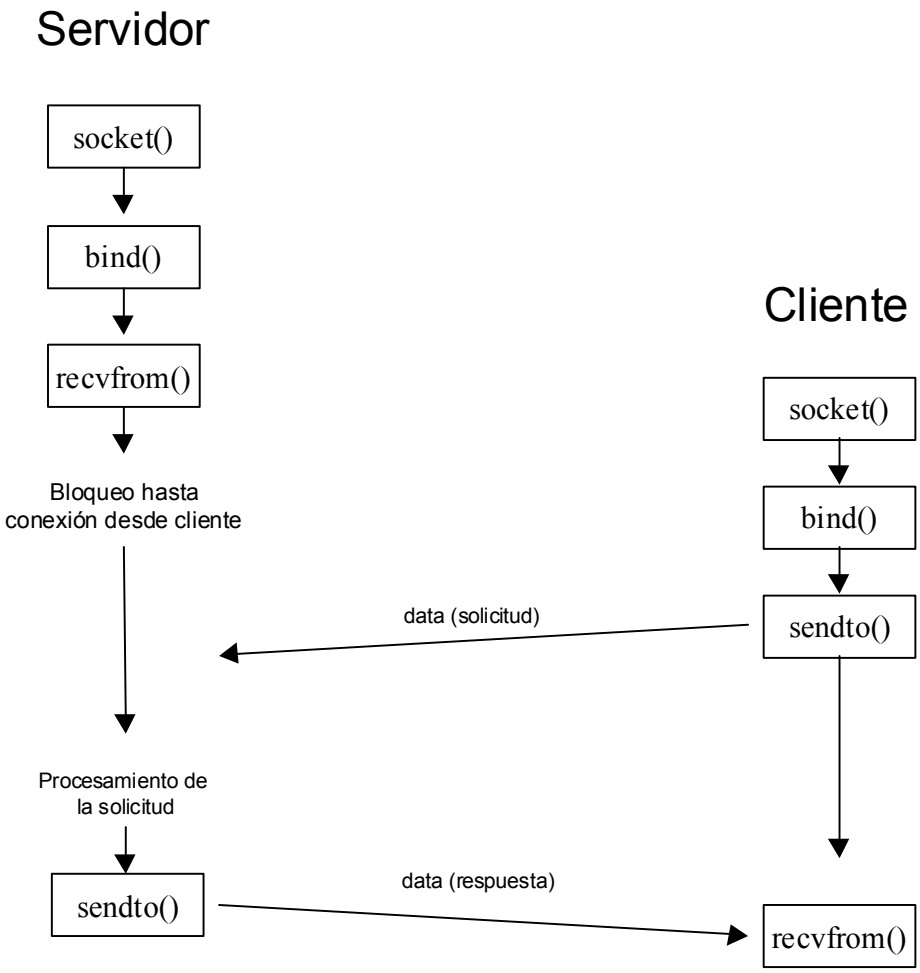

Figura 2. Llamadas al sistema para un protocolo no orientado a conexión.

Protocolos del Dominio UNIX

Los sistemas operativos tipo Unix proveen soporte a los llamados protocolos del dominio Unix. Diferente a los sockets de red, los sockets del dominio Unix sólo pueden utilizarse para comunicar procesos que se ejecutan en el mismo sistema Unix. Ellos constituyen una forma de comunicación interproceso, pero su implementación ha sido realizada dentro del sistema de red de forma similar a otros verdaderos protocolos de comunicación.

Los sockets del dominio Unix presentan interfaces de programación orientada a conexión y no orientada a conexión. Ambas pueden ser consideradas como confiables puesto que existen dentro del kernel, sin embargo se recomienda el uso de la interfaz orientada a conexión.

Los protocolos del dominio Unix presentan una característica no provista por otra familia de protocolos y mecanismos de comunicación interproceso: la habilidad de pasar los derechos de acceso de un proceso a otro. Además, no requieren de encapsulamiento de los mensajes.

El espacio de nombres empleado por los protocolos del dominio Unix incluye rutas de nombre de archivos (pathnames). Un ejemplo de asociación podría ser:

```
{unixstr, 0, /tmp/log.01528, 0, /dev/logfile}
```

El protocolo es `unixstr` (Unix Stream) y es orientado a conexión. Los procesos local y remoto de este ejemplo son `/tmp/log.01528` y `/dev/logfile`, respectivamente. Las direcciones remota y local tienen valor cero. Una asociación empleando el protocolo no orientado a conexión del dominio Unix es similar, excepto que emplea el término `unixdg` (Unix Datagram) en lugar de `unixstr`.

Las implementaciones tradicionales de estos sockets crean un "archivo" con el nombre y ruta especificada, sin embargo estas entradas al sistema de archivos no son verdaderos "archivos". Por ejemplo, no se pueden abrir con la llamada `open`. Estos archivos son del tipo `S_IFSOCK` tal como lo reporta las llamadas `stat` o `fstat`.

Direcciones de Sockets

La gran mayoría de las llamadas del sistema relacionadas con el manejo de sockets tienen como argumento un puntero a una estructura correspondiente a la dirección. Esta estructura está definida en `<sys/socket.h>`:

```
struct sockaddr {
   u_short sa_family;/* familia de dirección: AF_xxx */
   char sa_data[14]; /* dirección */
};
```

El contenido de los 14 bytes de dirección son interpretados de acuerdo al tipo de dirección (familia). Para la familia Internet, están definidas las siguientes estructuras en <netinet/in.h>:

```
struct in_addr {
   u_long s_addr; /* netid/hostid(32 bits) */
};

struct sockaddr_in {
   short sin_family; /* AF_INET */
   u_short sin_port; /* número de puerto(16 bits) */
   struct in_addr sin_addr; /* netid/hostid(32 bits) */
   char sin_zero[8];  /* sin uso */
};
```

Llamadas Elementales del Sistema

`socket()`

Esta llamada del sistema permite crear un socket especificando el tipo de protocolo de comunicación deseado (TCP, UDP, etc.)

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

int socket(int familia, int tipo, int protocolo);
```

La familias de direcciones (AF_XXXX, Address Family) más utilizadas son:

- AF_UNIX: Protocolos del Dominio Unix
- AF_INET: Protocolos Internet

También pueden utilizarse términos equivalentes como: PF_INET, PF_UNIX (PF _XXX, Protocol Family).

El tipo de socket puede ser alguno de los siguientes:

- SOCK_STREAM

- SOCK_DGRAM
- SOCK_RAW
- SOCK_SEQPACKET
- SOCK_RDM

No todas las combinaciones de familia y tipo son válidas. La tabla 1 muestra las combinaciones válidas junto con el protocolo realmente seleccionado.

	AF_UNIX	AF_INET
SOCK_STREAM	Si	TCP
SOCK_DGRAM	Si	UDP
SOCK_RAW		IP

Tabla 1. Protocolos correspondientes a la familia y tipo de socket.

El argumento protocolo típicamente se le asigna el valor de cero en la mayoría de las aplicaciones. Sin embargo, existen aplicaciones especializadas que lo especifican. Las combinaciones válidas para la familia de protocolos internet se muestra en la tabla 2.

Familia	Tipo	Protocolo	Protocolo Real
AF_INET	SOCK_DGRAM	IPPROTO_UDP	UDP
AF_INET	SOCK_STREAM	IPPROTO_TCP	TCP
AF_INET	SOCK_RAW	IPPROTO_ICMP	ICMP
AF_INET	SOCK_RAW	IPPROTO_RAW	(bajo nivel)

Tabla 2. Combinaciones de familia, tipo y protocolo.

Las constantes IPPROTO_XXX están definidas en el archivo `<netinet/in.h>`.

La llamada del sistema `socket` devuelve un entero corto, similar a un descriptor de archivo, denominado descriptor de socket. Para ello únicamente se ha especificado la familia de protocolo y el tipo de socket. Para utilizar realmente al socket creado deben ser especificados la dirección y el proceso local, así como la dirección y el proceso remoto.

`socketpair()`

Esta llamada sólo está disponible para el dominio Unix.

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

int socketpair(int familia, int tipo,
               int protocolo, int sockvec[2]);
```

Esta llamada tiene un argumento adicional a la llamada `socket`, se trata de `sockvec`. `socketpair()` crea dos sockets de manera simultánea y coloca dos descriptores de sockets en los dos elementos del arreglo `sockvect`. Esta llamada es similar a la llamada `pipe`. Se debe entender que `soketpair()` no es significativa cuando se aplica a la familia de portocolos TCP/IP, se incluye con el propósito de realizar una descripción completa de la interfaz.

Ya que esta llamada del sistema está limitada al dominio Unix, existen sólo dos versiones posibles:

```
int rc, sockfd[2];

int socketpair(AF_UNIX, SOCK_STREAM, 0,sockfd);

ó

int socketpair(AF_UNIX, SOCK_DGRAM, 0,sockfd);
```

`bind()`

Inicialmente un socket se crea sin ninguna asociación con direcciones locales o de destino. Para los protocolos TCP/IP, esto significa que no se ha asignado un puerto local, ni se han especificado puerto y dirección IP destino. En muchos casos los programas de aplicación no se preocupan por las direcciones locales que utilizan, sin embargo los procesos del servidor que operan en un puerto bien conocido deben ser capaces de especificar dicho puerto para el

sistema. Una vez que se ha creado un socket, el servidor utiliza la llamada `bind()` para establecer una dirección local. La sintaxis de la llamada es la siguiente:

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

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

El segundo argumento es un puntero a la dirección correspondiente al protocolo empleado y el tercer argumento es el tamaño de dicha estructura.

La llamada `bind()` tiene tres usos:

1. Registrar la dirección bien conocida de los servidores con el sistema. Tanto los servidores orientados a conexión como los no orientados a conexión requieren especificar esta dirección antes de aceptar solicitudes.
2. Un cliente puede registrar una dirección específica para sí mismo.
3. Un cliente no orientado a conexión necesita asegurar que el sistema le asigne una dirección única, de manera tal que el servidor tenga una dirección válida a dónde enviar sus respuestas.

`connect()`

Inicialmente, un socket recientemente creado se encuentra “desconectado”, lo que significa que éste no está asociado con ningún destino externo. La llamada de sistema `connect()` enlaza permanentemente un socket a un destino. Un programa de aplicación debe llamar a `connect` para establecer una conexión antes de que pueda transferir datos a través de un socket de flujo confiable.

Los sockets utilizados con servicios de datagramas sin conexión no necesitan estar conectados antes de ser utilizados.

Un proceso cliente conecta un socket mediante la siguiente llamada de sistema:

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

```
int connect(int sockfd, struct sockaddr *servaddr,
            int addlen);
```

El argumento `sockfd` es un descriptor de socket. El segundo y tercer argumento es un puntero a una estructura de dirección del socket y el tamaño de ésta, respectivamente.

Para la mayoría de los protocolos orientados a conexión, la llamada `connect()` resulta en el establecimiento de una conexión real entre el sistema remoto y un sistema local. Típicamente, son intercambiados mensajes entre los dos sistemas pudiéndose establecer los parámetros de la "conversación" (tamaño del búfer, cantidad de data a intercambiar entre acuses de recibo, etc.).

Si no es posible establecer la conexión, la llamada `connect()` devuelve un código de error.

El cliente no tiene que haberse enlazado a una dirección local (`bind`) antes de llamar a `connect()`. La conexión típicamente causa que sean asignados la dirección y el proceso local.

Un cliente no orientado a conexión también puede utilizar la llamada `connect()`, pero en este caso tienen otro significado. Para un protocolo sin conexión, todo lo que hace `connect()` es almacenar la dirección `servaddr` específicada por el proceso, de forma tal que el sistema sabe a donde enviar cualquier data que el proceso escriba al socket. Únicamente datagramas provenientes de esta dirección serán recibidos por el socket. En este caso, `connect()` regresa inmediatamente y no existe un verdadero intercambio de mensajes entre el sistema local y el remoto.

Una ventaja de emplear la llamada `connect()` cuando se utiliza un protocolo no orientado a conexión es que no es necesario especificar la dirección de destino para cada datagrama. Se pueden emplear las llamadas `read`, `write`, `recv` y `send`.

Existe otra característica importante de los protocolos no orientados a conexión cuando se emplea la llamada `connect()`. Si el protocolo soporta

notificación de dirección inválida, la rutina del protocolo puede informar al proceso del usuario si éste ha enviado un datagrama a una dirección invalida.

`listen()`

La llamada de sistema `listen()` es utilizada por un servidor orientado a conexión para indicar que está preparado para recibir conexiones.

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

Es usualmente ejecutada después de las llamadas `socket()` y `bind()`, e inmediatamente antes de `accept()`. El segundo argumento (`backlog`) especifica cuántas solicitudes de conexión pueden esperar en cola mientras el sistema ejecuta la llamada `accept()`. El valor típico de este argumento es 5, el máximo valor permitido.

En el instante que el servidor crea un proceso hijo mediante la llamada `fork()` y el proceso padre ejecuta de nuevo la llamada `accept()`, es posible que arriven nuevas solicitudes de conexión de otros clientes. Precisamente, `backlog` se refiere a esta cola de solicitudes de conexiones pendientes.

`accept()`

Después de la llamada `listen()`, un servidor orientado a conexión espera una conexión real de algún cliente mediante la ejecución de `accept()`.

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

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

`accept()` toma la primera solicitud de conexión de la cola y crea otro descriptor de socket con la mismas propiedades de `sockfd`. Si no hay solicitud de conexión pendientes, esta llamada se bloquea hasta que alguna arrive.

Los argumentos `peer` y `addrlen` son utilizados para retornar la dirección del proceso par conectado (cliente) y el tamaño de la estructura de dirección. Para la familia de protocolos TCP/IP esta estructura tiene un tamaño de 16 bytes. El valor entero que devuelve la llamada corresponde a un código de error o a un descriptor de socket.

`accept()` automáticamente crea un nuevo descriptor de socket, asumiendo que el servidor es de tipo concurrente lo cual es lo más frecuente. Para este caso se presenta el siguiente ejemplo de código:

```
int sockfd, newsockfd;

if ((sockfd = socket(...)) < 0)
   err_sys("socket error");
if (bind(sockfd, ...) < 0)
   err_sys("bind error");
if (listen(sockfd, 5) < 0)
   err_sys("listen error");

for(; ;)
{
   newsockfd=accept(sockfd, ...);  /* se bloquea */
   if (newsokfd < 0)
      err_sys("accept error");
   if (fork() == 0)
   {
      close(sockfd);  /*  proceso hijo  */
      doit(newsockfd); /* procesamiento de la solicitud */
      exit(0);
   }

   close(newsockfd);  /* proceso padre */
}
```

Cuando una conexión es recibida y aceptada, el proceso realiza una llamada `fork()`, donde el proceso hijo sirve a la conexión y el padre espera por otra solicitud.

Considerando un servidor iterativo, se presenta el siguiente ejemplo de código:

```
int sockfd, newsockfd;

if ((sockfd = socket(...)) < 0)
   err_sys("socket error");
if (bind(sockfd, ...) < 0)
   err_sys("bind error");
if (listen(sockfd, 5) < 0)
   err_sys("listen error");

for(; ;)
{
   newsockfd=accept(sockfd, ...);  /* se bloquea */

   if (newsokfd < 0)
      err_sys("accept error");

   doit(newsockfd); /* procesamiento de la solicitud */
   close(newsockfd);
}
```

En este caso el servidor utiliza en descriptor del socket conectado, `newsockfd`. Luego de cerrar la conexión espera por otra conexión empleando el socket original `sockfd`.

`send()`, `sendto()`, `recv()` y `recvfrom()`

Estas llamadas son símilares a las llamadas `read()` y `write()`, excepto por algunos argumentos adicionales.

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

int send(int sockfd,char *buff, int nbytes,
         int flags);

int sendto(int sockfd,char *buff, int nbytes,
           int flags, struct sockaddr *to,
```

```
            int *addrlen);

int recv(int sockfd,char *buff,int nbytes,
         int flags);

int sendto(int sockfd,char *buff, int nbytes,
           int flags, struct sockaddr *from,
           int *addrlen);
```

Los primeros tres argumentos son similares a los argumentos correspondientes a las llamadas `read()` y `write()`. El argumento `sockfd` contiene un descriptor de socket entero. `buff` contiene la dirección de memoria de los datos que se van a enviar o donde se almacenarán los datos recibidos. `nbytes` especifica la cantidad de bytes de estos datos. El valor de argumento flags es cero o el reultado de la operación *OR* de una de las siguientes constantes:

- `MSG_OOB`: datos fuera de banda
- `MSG_PEEK`: obtiene la data disponible sin descartarla hasta el regreso de la función (`recv()` o `recvfrom()`)
- `MSG_DONTROUTE`: descartar enrutamiento (send o sendto)

El argumento de la llamada `sendto()` especifica la dirección destino y el argumento `addrlen`, el tamaño de la misma. La llamada `recvfrom()` devuelve la estructura de dirección correspondiente al proceso que envía la información, al igual que el tamaño de dicha estructura.

Todas estas funciones devuelven como valor la cantidad de bytes que fueron enviados o recibidos. Típicamente la llamada recvfrom es utilizada con protocolo no oreintados a conexión, en este caso el valor devuelto corresponde al tamaño del datagrama recibido.

`close()`

Cuando un programa llama a `fork()`, la copia que se crea hereda el acceso a todos los sockets abiertos, justo como ocurre con los archivos. Sin embargo, cuando el programa llama a `exec()`, la nueva aplicación retiene el acceso a

todos los sockets abiertos. Tanto los procesos padres como los procesos hijos tienen los mismo derechos de acceso a los sockets existentes. Luego, es responsabilidad del programador asegurar que los sokets sean utilizados por ambos procesos de forma adecuada.

Cuando un proceso termina de utilizar un socket, éste llama a:

```
int close(int sockfd);
```

Si el socket que está siendo cerrado está asociado con un protocolo que garantiza la distribución confiable de la información (e.g. TCP), el sistema debe asegurarse que todos los datos o acuses de recibo pendientes sean enviados. Normalmente, después de una llamada `close()` el sistema retorna el control inmediatamente, pero el kernel aun intenta enviar la data que está en cola.

Rutinas de Ordenamiento de Bytes

Las siguientes cuatro funciones manejan las potenciales diferencias de ordenamiento de bytes entre diferentes arquitecturas computacionales y diferentes protocolos de red.

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

u_long htonl(u_long hostlong);
u_short  htons(u_short hostshort);
u_long ntohl(u_long netlong);
u_short  ntohs(u_short netshort);
```

Estás funciones fueron diseñadas para los protocolos Internet. En sistemas que tienen el mismo esquema de ordenamiento que los protocolos Internet, tal como los procesadores de la serie Motoral 68000, estás funciónes son unos macros nulos. Las conversiones hechas por estas funciones son las siguientes:

Htonl	Convierte un entero largo del host a uno de TCP/IP
Htons	Convierte un entero corto del host a uno de TCP/IP
Ntohl	Convierte un entero largo de TCP/IP a uno del host
Ntohs	Convierte un entero corto de TCP/IP a uno del host

Todas estas funciones operan sobre valores enteros sin signo, a pesar de que ellas trabajan también sobre valores con signo. Implicito a estas funciones está que un entero corto ocupa 16 bits y un entero largo 32 bits.

EJEMPLOS

A continuación se muestra el código de un programa servidor el cual recibe un mensaje y lo devuelve al programa cliente. Se presentan dos versiones una no orientada a conexión y otra orientada a conexión

Ejemplo 1

```
/*
      NAME: UDP.ECHO.SERVER
      SYNOPSIS: UDPServer
      DESCRIPTION: Este programa crea un socket del dominio inet no
      orientado a conexión (UDP) el cual recibe un mansaje de un proceso
      cliente y lo retorna a éste.
*/

#include <stdio.h>
#include <errno.h>
#include <sys/time.h>

#include <sys/param.h>
#include <sys/socket.h>
#include <sys/file.h>

#include <netinet/in_systm.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <netdb.h>

#define MAXHOSTNAME 80

void reusePort(int sock);
```

```
main( argc, argv )
int argc;
char *argv[];
{
   int   sd;
   struct   sockaddr_in server;
   struct  hostent *hp, *gethostbyname();
   struct  servent *sp;
   struct sockaddr_in from;
   int fromlen;
   int length;
   char buf[512];
   int rc;
   char ThisHost[80];

   sp = getservbyname("echo", "udp");

/* get EchoServer Host information, NAME and INET ADDRESS */

   gethostname(ThisHost, MAXHOSTNAME);

/* OR strcpy(ThisHost,"localhost"); */

   printf("----EchoServer running at host NAME: %s\n", ThisHost);
   if  ( (hp = gethostbyname(ThisHost)) == NULL ) {
      fprintf(stderr, "Can't find host %s\n", argv[1]);
      exit(-1);
   }
   bcopy ( hp->h_addr, &(server.sin_addr), hp->h_length);
   printf("    (EchoServer INET ADDRESS is: %s )\n",
                  inet_ntoa(server.sin_addr));


/* Construct name of socket to send to. */
   server.sin_family = hp->h_addrtype;
   /*OR server.sin_family = AF_INET; */


   server.sin_addr.s_addr = htonl(INADDR_ANY);

    server.sin_port = htons(0);
   /*OR    server.sin_port = sp->s_port; */
    /* server.sin_port = htons(atoi(argv[1]));  */

/* Create socket on which to send  and receive */

   /* sd = socket (PF_INET,SOCK_DGRAM,0); */
   sd = socket (hp->h_addrtype,SOCK_DGRAM,0);

   if (sd<0) {
      perror("opening datagram socket");
      exit(-1);
   }
reusePort(sd);
   if ( bind( sd, &server, sizeof(server) ) ) {
      close(sd);
```

```
      perror("binding name to datagram socket");
      exit(-1);
   }

   length = sizeof(server);
   if ( getsockname (sd,&server,&length) ) {
      perror("getting socket name");
      exit(0);
   }

   printf("Server Port is: %d\n", ntohs(server.sin_port));

/*  get data from  clients and send it back */
   for(;;){
      fromlen = sizeof(from);
      printf("\n...server is waiting...\n");
      if ((rc=recvfrom(sd, buf, sizeof(buf), 0, &from, &fromlen)) < 0)
         perror("receiving datagram  message");
      if (rc > 0){
         buf[rc]=NULL;
         printf("Received: %s\n", buf);
         printf("From %s:%d\n", inet_ntoa(from.sin_addr),
             ntohs(from.sin_port));
         if ((hp = gethostbyaddr(&from.sin_addr.s_addr,
             sizeof(from.sin_addr.s_addr),AF_INET)) == NULL)
            fprintf(stderr, "Can't find host %s\n",
inet_ntoa(from.sin_addr));
         else
            printf("(Name is : %s)\n", hp->h_name);

         if (sendto(sd, buf, rc, 0, &from, sizeof(from)) <0 )
            perror("sending datagram message");
      }
   }
}
void reusePort(int s)
{
  int one=1;

  if ( setsockopt(s,SOL_SOCKET,SO_REUSEADDR,(char *) &one,sizeof(one)) ==
-1 )
  {
    printf("error in setsockopt,SO_REUSEPORT \n");
    exit(-1);
  }
}
```

```
/*
      NAME: UDP.ECHO.CLIENT
      SYNOPSIS: UDPClient hostid portnumber
      DESCRIPTION: El programa crea un socket del dominio inet no
      orientado a conexión (UDP), toma los mensajes escritos por el
      usuario y los envía al servidor de "echo" ejecutandose en el
      hostid. Luego, espera la respuesta del servidor de "echo", y lo
      muestra al usuario, con un mensaje indicando su hubo un error
      durante la transmisión.

*/


#include <stdio.h>
#include <errno.h>
#include <sys/time.h>

#include <sys/param.h>
#include <sys/socket.h>
#include <sys/file.h>

#include <netinet/in_systm.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <netdb.h>
#define MAXHOSTNAME 80
#define BUFSIZE 1024
char buf[BUFSIZE];
char rbuf[BUFSIZE];

main( argc, argv )
int argc;
char *argv[];
{
   int   sd;
   struct   sockaddr_in server;
   struct  hostent *hp, *gethostbyname();
   struct  servent *sp;
   struct sockaddr_in from;
   struct sockaddr_in addr;
   int fromlen;
   int length;
   int rc, cc;
   char ThisHost[80];

   sp = getservbyname("echo", "udp");
/* get EchoClient Host information, NAME and INET ADDRESS */

   gethostname(ThisHost, MAXHOSTNAME);

/* OR strcpy(ThisHost,"localhost"); */


   printf("----EchoCleint running at host NAME: %s\n", ThisHost);
   if  ( (hp = gethostbyname(ThisHost)) == NULL ) {
```

```
      fprintf(stderr, "Can't find host %s\n", argv[1]);
      exit(-1);
   }
   bcopy ( hp->h_addr, &(server.sin_addr), hp->h_length);
   printf("    (EchoCleint INET ADDRESS is: %s )\n",
inet_ntoa(server.sin_addr));

/* get EchoServer Host information, NAME and INET ADDRESS */

   if  ( (hp = gethostbyname(argv[1])) == NULL ) {
      addr.sin_addr.s_addr = inet_addr(argv[1]);
      if ((hp = gethostbyaddr(&addr.sin_addr.s_addr,
          sizeof(addr.sin_addr.s_addr),AF_INET)) == NULL) {
         fprintf(stderr, "Can't find host %s\n", argv[1]);
         exit(-1);
      }
   }
   printf("----EchoServer running at host NAME: %s\n", hp->h_name);
   bcopy ( hp->h_addr, &(server.sin_addr), hp->h_length);
   printf("    (EchoServer INET ADDRESS is: %s )\n",
inet_ntoa(server.sin_addr));

/* Construct name of socket to send to. */
   server.sin_family = hp->h_addrtype;
   /* server.sin_family = AF_INET; */

   server.sin_port = htons(atoi(argv[2]));
   /*OR    server.sin_port = sp->s_port; */

   /*   Create socket on which to send  and receive */

   sd = socket (hp->h_addrtype,SOCK_DGRAM,0);
   /*OR  sd = socket (PF_INET,SOCK_DGRAM,0); */

   if (sd<0) {
      perror("opening datagram socket");
      exit(-1);
   }
/* get data from USER, send it SERVER,
      receive it from SERVER, display it back to USER  */
   for(;;) {
      printf("\nType anything followed by RETURN, or type CTRL-D to
exit\n");
      cleanup(buf);
      cleanup(rbuf);
      rc=read(0,buf, sizeof(buf));
      if (rc == 0) break;

      if (sendto(sd, buf, rc, 0, &server, sizeof(server)) <0 )
         perror("sending datagram message");
      fromlen= sizeof(from);
      if ((cc=recvfrom(sd, rbuf, sizeof(rbuf), 0, &from, &fromlen)) < 0)
         perror("receiving echo");
      if (cc == rc)
         if (strcmp(buf, rbuf) == 0){
            printf("Echo is good\n");
            printf("  Received: %s", rbuf);
```

```
            printf("  from %s:%d\n", inet_ntoa(from.sin_addr),
                ntohs(from.sin_port));
            if ((hp = gethostbyaddr(&from.sin_addr.s_addr,
                sizeof(from.sin_addr.s_addr),AF_INET)) == NULL)
               fprintf(stderr, "Can't find host %s\n",
                   inet_ntoa(from.sin_addr));
            else
               printf("  (Name is : %s)\n", hp->h_name);
         }
         else
            printf("Echo bad -- strings unequal: %s\n", rbuf);
      else
         printf("Echo bad -- lengths unequal: %s\n", rbuf);
   }
   printf ("EOF... exit\n");
   close(sd);
   exit (0);
}
cleanup(buf)
char *buf;
{
   int i;
   for(i=0; i<BUFSIZE; i++) buf[i]=NULL;
}
```

Ejemplo 2

```
/*
      NAME: TCPServer
      SYNOPSIS: TCPServer
      DESCRIPTION: Este programa crea un socket del dominio inet
      orientado a conexión (TCP), espera conexiones de los clientes TCP,
      acepta las conexiones, crea un proceso hijo y atiende la solicitud
      recibiendo un mansaje del proceso cliente y retornándolo a éste.
*/


#include <stdio.h>
#include <errno.h>
#include <sys/time.h>

#include <sys/param.h>
#include <sys/socket.h>
#include <sys/file.h>

#include <netinet/in_systm.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <netdb.h>

#define MAXHOSTNAME 80
void reusePort(int sock);

main( argc, argv )
int argc;
char *argv[];
{
  int   sd, psd;
  struct   sockaddr_in server;
  struct  hostent *hp, *gethostbyname();
  struct  servent *sp;
  struct sockaddr_in from;
  int fromlen;
  int length;
  char ThisHost[80];

  sp = getservbyname("echo", "tcp");
  /* get EchoServer Host information, NAME and INET ADDRESS */

  gethostname(ThisHost, MAXHOSTNAME);

  /* OR strcpy(ThisHost,"localhost"); */


  printf("----TCP/Server running at host NAME: %s\n", ThisHost);
  if  ( (hp = gethostbyname(ThisHost)) == NULL ) {
    fprintf(stderr, "Can't find host %s\n", argv[1]);
    exit(-1);
```

```
  }
  bcopy ( hp->h_addr, &(server.sin_addr), hp->h_length);
  printf("    (TCP/Server INET ADDRESS is: %s )\n",
inet_ntoa(server.sin_addr));


  /* Construct name of socket to send to. */
  server.sin_family = hp->h_addrtype;
  /*OR server.sin_family = AF_INET; */


  server.sin_addr.s_addr = htonl(INADDR_ANY);

  /* server.sin_port = htons((u_short) 0);  i*/
  /*OR    server.sin_port = sp->s_port; */
  server.sin_port = htons( (u_short) atoi(argv[1]));

  /*   Create socket on which to send  and receive */

  /* sd = socket (PF_INET,SOCK_DGRAM,0); */
  sd = socket (hp->h_addrtype,SOCK_STREAM,0);

  if (sd<0) {
    perror("opening stream socket");
    exit(-1);
  }

reusePort(sd);

  if ( bind( sd, &server, sizeof(server) ) ) {
    close(sd);
    perror("binding name to stream socket");
    exit(-1);
  }

  length = sizeof(server);
  if ( getsockname (sd,&server,&length) ) {
    perror("getting socket name");
    exit(0);
  }

  printf("Server Port is: %d\n", ntohs(server.sin_port));
  listen(sd,4);
  fromlen = sizeof(from);
  for(;;){
    psd  = accept(sd, &from, &fromlen);
    if ( fork() == 0) {
      close (sd);
      EchoServe(psd, from);
    }
  }
}

EchoServe(psd, from)
int psd;
struct sockaddr_in from;
{
```

```
  char buf[512];
  int rc;
  struct  hostent *hp, *gethostbyname();

  printf("Serving %s:%d\n", inet_ntoa(from.sin_addr),
      ntohs(from.sin_port));
  if ((hp = gethostbyaddr(&from.sin_addr.s_addr,
      sizeof(from.sin_addr.s_addr),AF_INET)) == NULL)
    fprintf(stderr, "Can't find host %s\n", inet_ntoa(from.sin_addr));
  else
    printf("(Name is : %s)\n", hp->h_name);


  /*      get data from  clients and send it back */
  for(;;){
    printf("\n...server is waiting...\n");
    if( (rc=read(psd, buf, sizeof(buf))) < 0)
      perror("receiving stream  message");
    if (rc > 0){
      buf[rc]=NULL;
      printf("Received: %s\n", buf);
      printf("From TCP/Client: %s:%d\n", inet_ntoa(from.sin_addr),
          ntohs(from.sin_port));
      printf("(Name is : %s)\n", hp->h_name);
      if (send(psd, buf, rc, 0) <0 )
        perror("sending stream message");
    }
    else {
      printf("TCP/Client: %s:%d\n", inet_ntoa(from.sin_addr),
          ntohs(from.sin_port));
      printf("(Name is : %s)\n", hp->h_name);
      printf("Disconnected..\n");
      close (psd);
      exit(0);
    }
  }
}
void reusePort(int s)
{
  int one=1;

  if ( setsockopt(s,SOL_SOCKET,SO_REUSEADDR,(char *) &one,sizeof(one)) ==
-1 )
  {
    printf("error in setsockopt,SO_REUSEPORT \n");
    exit(-1);
  }
}
```

```
/*
      NAME: TCPClient
      SYNOPSIS: TCPClient hostid portnumber
      DESCRIPTION: El programa crea un socket del dominio inet orientado
      a conexión (TCP), toma los mensajes escritos por el usuario y los
      envía al servidor de "echo" ejecutandose en el hostid. Luego,
      espera la respuesta del servidor de "echo", y lo muestra al
      usuario, con un mensaje indicando su hubo un error durante la
      transmisión.

*/


#include <stdio.h>
#include <errno.h>
#include <sys/time.h>

#include <sys/param.h>
#include <sys/socket.h>
#include <sys/file.h>

#include <netinet/in_systm.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <netdb.h>
#define MAXHOSTNAME 80
#define BUFSIZE 1024
char buf[BUFSIZE];
char rbuf[BUFSIZE];

main( argc, argv )
int argc;
char *argv[];
{
  int   sd;
  struct   sockaddr_in server;
  struct  hostent *hp, *gethostbyname();
  struct  servent *sp;
  struct sockaddr_in from;
  struct sockaddr_in addr;
  int fromlen;
  int length;
  int rc, cc;
  char ThisHost[80];

  sp = getservbyname("echo", "tcp");
  /* get TCPClient Host information, NAME and INET ADDRESS */

  gethostname(ThisHost, MAXHOSTNAME);

  /* OR strcpy(ThisHost,"localhost"); */


  printf("----TCP/Cleint running at host NAME: %s\n", ThisHost);
  if  ( (hp = gethostbyname(ThisHost)) == NULL ) {
```

```
    fprintf(stderr, "Can't find host %s\n", argv[1]);
    exit(-1);
  }
  bcopy ( hp->h_addr, &(server.sin_addr), hp->h_length);
  printf("    (TCP/Cleint INET ADDRESS is: %s )\n",
inet_ntoa(server.sin_addr));

  /* get TCP/Server Host information, NAME and INET ADDRESS */

  if  ( (hp = gethostbyname(argv[1])) == NULL ) {
    addr.sin_addr.s_addr = inet_addr(argv[1]);
    if ((hp = gethostbyaddr(&addr.sin_addr.s_addr,
        sizeof(addr.sin_addr.s_addr),AF_INET)) == NULL) {
      fprintf(stderr, "Can't find host %s\n", argv[1]);
      exit(-1);
    }
  }
  printf("----TCP/Server running at host NAME: %s\n", hp->h_name);
  bcopy ( hp->h_addr, &(server.sin_addr), hp->h_length);
  printf("    (TCP/Server INET ADDRESS is: %s )\n",
inet_ntoa(server.sin_addr));

  /* Construct name of socket to send to. */
  server.sin_family = hp->h_addrtype;
  /* server.sin_family = AF_INET; */

  server.sin_port = htons(atoi(argv[2]));
  /*OR    server.sin_port = sp->s_port; */

  /*   Create socket on which to send  and receive */

  sd = socket (hp->h_addrtype,SOCK_STREAM,0);
  /*OR  sd = socket (PF_INET,SOCK_STREAM,0); */

  if (sd<0) {
    perror("opening stream socket");
    exit(-1);
  }
  /* Connect to TCP/SERVER */
  if ( connect(sd, &server, sizeof(server)) < 0 ) {
    close(sd);
    perror("connecting stream socket");
    exit(0);
  }
  fromlen = sizeof(from);
  if (getpeername(sd,&from,&fromlen)<0){
    perror("could't get peername\n");
    exit(1);
  }
  printf("Connected to TCP/Server:");
  printf("%s:%d\n", inet_ntoa(from.sin_addr),
      ntohs(from.sin_port));
  if ((hp = gethostbyaddr(&from.sin_addr.s_addr,
      sizeof(from.sin_addr.s_addr),AF_INET)) == NULL)
    fprintf(stderr, "Can't find host %s\n", inet_ntoa(from.sin_addr));
  else
    printf("(Name is : %s)\n", hp->h_name);
```

```
  /*     get data from USER, send it SERVER,
      receive it from SERVER, display it back to USER  */

  for(;;) {
    printf("\nType anything followed by RETURN, or type CTRL-D to
exit\n");
    cleanup(buf);
    cleanup(rbuf);
    rc=read(0,buf, sizeof(buf));
    if (rc == 0) break;
    if (send(sd, buf, rc, 0) <0 )
      perror("sending stream message");
    if ((cc=recv(sd, rbuf, sizeof(rbuf), 0)) < 0)
      perror("receiving echo");
    if (cc == rc)
      if (strcmp(buf, rbuf) == 0){
        printf("Echo is good\n");
        printf("  Received: %s", rbuf);
      }
      else
        printf("Echo bad -- strings unequal: %s\n", rbuf);
    else
      printf("Echo bad -- lengths unequal: %s\n", rbuf);
  }
  printf ("EOF... exit\n");
  close(sd);
  exit (0);
}
cleanup(buf)
char *buf;
{
  int i;
  for(i=0; i<BUFSIZE; i++) buf[i]=NULL;
}
```

CONCLUSIONES

En este documento se presentaron los fundamentos para el desarrollo de aplicaciones cliente/servidor empleando para ello la interfaz de programación de aplicaciones de comunicación provista por los sistemas tipo Unix (Berkeley Sockets).

REFERENCIAS

1. Stevens, W. Richard. “Unix Network Programming”. Prentice-Hall, Inc. 1990.
2. Comer, D. E. “Internetworking with TCP/IP: Principles, Protocols, and Architecture”. Prentice-Hall, Englewood Cliffs, N.J. 1988.