Programación en Redes TCP/IP Utilizando Interfase de Sockets

Sergio D. Saade y José L. Borelli

Resumen

En la actualidad, el conjunto de protocolos de comunicación de computadoras de mayor difusión a nivel mundial, lo constituye sin dudas TCP/IP. Sobre él actúan la vasta mayoría de aplicaciones de redes como telnet, ftp, servidores de web, correo electrónico, servidores de chat, etc.
La presente publicación introduce los fundamentos que se utilizan para la construcción de tales aplicaciones, utilizando una de las interfases disponibles de programación: Interfase de Sockets.
La publicación comienza con un resumen de la suite de protocolos TCP/IP, para luego definir lo que se entiende como un "socket". Se introducen los dos tipos principales de "sockets" y su uso en aplicaciones distribuidas.
Se incluyen las principales llamadas al sistema ("system calls") utilizadas para la programación en red.
El artículo concluye con un ejemplo de un programa que demuestra el uso de los "system calls" introducidos y de la estructura de la programación de sistemas distribuidos en redes TCP/IP.

Palabras Claves: Redes de Computadoras – Sistemas Operativos - Protocolos TCP/IP – Programación Sockets.

Abstract

At the present time the most popular computer communication protocol suite worldwide is without any doubt TCP/IP. All familiar network applications, such as telnet, ftp, web servers, email, etc. are developed above it.
This publication introduces the foundation for building such applications using one of the available programming interfaces known as socket interface.
This publication starts introducing the TCP/IP protocol suite briefly in order to define the term socket. Then, the two main sockets types are defined, as well as their use in distributed applications.
The most important system calls used in network programming are shown.
The publication finishes with a network program, which demonstrates the use of the system calls introduced and the programming structure in distributed application in TCP/IP networks.

Keywords: Computer Networks – Operating Systems – TCP/IP Protocols – Sockets programming.

Introducción

La suite de protocolos TCP/IP es el conjunto de protocolos más utilizado hoy en día para la comunicación entre computadoras que conforman una red.
Se puede sintetizar a los protocolos TCP/IP, por una estructura *jerárquica*, como la mostrada en la figura 1[1].

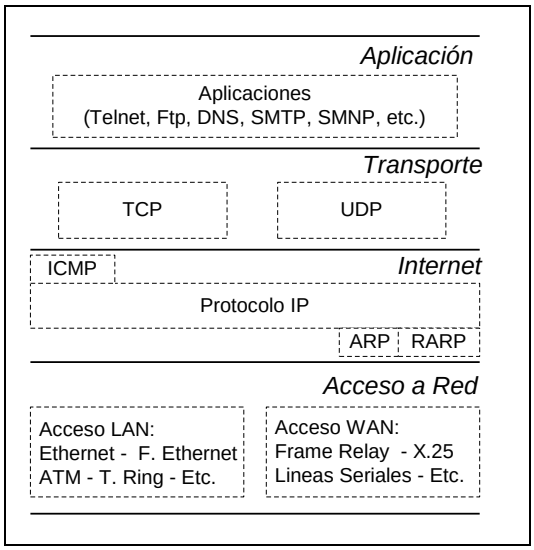

Figura 1: Estructura de Protocolos TCP/IP

La arquitectura esta formada por cuatro capas perfectamente distinguibles:

- *Capa de Acceso a la Red:* esta capa, en realidad no forma parte del modelo TCP/IP, puesto que incluye el conjunto de tecnologías y protocolos de acceso a redes LAN, MAN y WAN.
- *Capa de Internet:* cuya función principal es la entrega ("delivery") de paquetes de datos desde un computador origen a un computador destino. El protocolo más importante dentro de esta capa es **IP** (**I**nternet **P**rotocol), cuya función es la de

[1] La referencia bibliográfica [1] contiene un excelente tratado de las Suite de Protocolos TCP/IP.

establecer un encaminamiento o ruteo de tales tramas de datos.

- *Capa de Transporte:* cuya función es ofrecer transferencia de datos extremo-extremo (léase computador origen a computador destino), independiente del camino o ruta que siguen los datos (resuelto por IP). En esta capa se distinguen dos protocolos: **TCP** (**T**ransport **C**ontrol **P**rotocol) y **UDP** (**U**ser **D**atagram **P**rotocol). El primero es un protocolo confiable y orientado a conexión, mientras que el segundo es no-confiable y no-orientado a conexión.
- *Capa de Aplicación:* en esta capa es donde se encuentran las aplicaciones de red, tales como aplicaciones para transferencias de archivos (ftp, tftp, rcp y otras), login remoto (rlogin, telnet, ssh y otras), servidores webs, correo electrónico, etc.

Las aplicaciones utilizan una de varias interfases disponibles para acceder a los protocolos de capa de transporte, a saber: interfase de "sockets", interfase **TLI** ("**T**ransport **L**ayer **I**nterface"), Interfase NetBIOS u otras. Dentro de estas interfases (o API's[2]), la de mayor uso en la actualidad, la constituye la de Sockets, que será analizada en el presente artículo.

Puertos y Sockets

La interfase de "sockets" fue creada en 1981 para la versión Berkeley de UNIX (BSD), de allí que en la bibliografía se la referencia comúnmente como "Berkeley Sockets". Antes de introducir las principales funciones utilizada por dicha interfase, se definirá lo que es un "socket"[3].

La forma en que una aplicación se identifica con el protocolo de transporte (y viceversa) es a través de un número denominado **puerto** (cuyo tamaño es de 16 Bytes). Un puerto identifica en forma unívoca a la aplicación, vale decir que dentro de un computador, no pueden existir más de una aplicación con el mismo número de puerto. La figura 2, muestra algunas aplicaciones típicas, con los números de puertos que utiliza y el correspondiente protocolo de transporte.

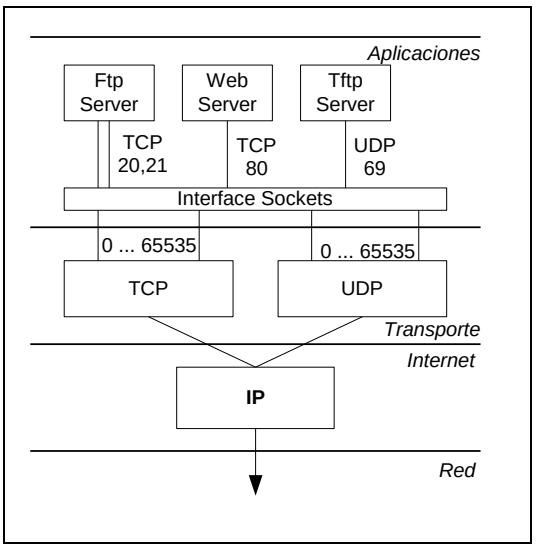

Figura 2: Número de Puertos en Aplicaciones (Servidoras)

Las aplicaciones de red bajo TCP/IP, siguen normalmente el modelo Cliente/Servidor, en donde se tiene por una parte una aplicación servidora que atiende y resuelve las solicitudes de (hipotéticamente) diversas aplicaciones clientes. Por ejemplo, se puede tener una aplicación para la transferencia de archivos, como ftp. Esta aplicación consta de una parte servidora - el servidor ftp – y una parte cliente – el cliente ftp. Este último solicita al primero el servicio de transferencia de un cierto archivo, para lo cual el servidor contesta con el envío del mismo.

Las aplicaciones servidoras, siempre atienden o escuchan en puertos que son universalmente conocidos[4] ("well known ports"), mientras que las aplicaciones clientes, lo hacen sobre puertos que son creados dinámicamente[5] ("dinamically allocated ports").

En la figura 2, las aplicaciones que se muestran son servidoras, por lo tanto, los puertos correspondientes son puertos universalmente conocidos.

Con un puerto se identifica una aplicación dentro de un computador en forma precisa. Con un "socket", es posible identificar a una aplicación dentro de toda la red. Eso se consigue especificando además de la aplicación, la dirección IP del computador donde reside (que es única). Se tiene así una terna:

(Dirección IP, Protocolo de Transporte, Puerto)

[2] API: Application Program Interface.

[3] El término "socket" podría traducirse al español, como tomacorrientes. Es decir, un "socket", vendría a ser el punto de contacto entre un artefacto eléctrico y la red eléctrica.

[4] Ocupan el rango de números que se encuentra entre 0 a 1023.

[5] Ocupan el rango de números que se encuentra entre 1024 a 65.535 (2^{16}-1).

que define en forma unívoca a una aplicación dentro de toda la Internet. Esta terna se denomina un **"socket"**.
En particular, si tenemos una comunicación entre dos aplicaciones cooperativas, se puede representar la asociación entre ambas aplicaciones a través de una quíntupla:

(Protocolo, Dirección IP Local, Puerto Local, Dirección IP Remota, Puerto Remoto)

La figura 3, muestra esta quíntupla, la cual crea un canal de comunicación **virtual** entre las dos aplicaciones distribuidas

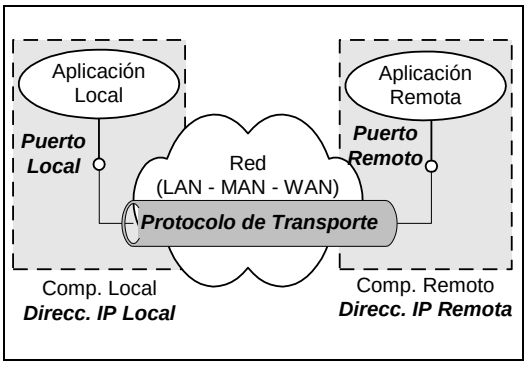

Figura 3: Comunicación entre Aplicaciones Cooperativas Distribuidas

La interfase de "sockets" tiene como un atractivo particular, el hecho que realiza una analogía entre la interfase de acceso al sistema de archivos de Unix y la comunicación entre aplicaciones distribuidas en red.
En efecto, en Unix, un archivo está identificado en cada proceso a través de un número que se denomina descriptor de archivo. La API del sistema de archivo, define funciones como open(), close(), read(), write(), etc. que utilizan a tal descriptor para acceder al archivo y realizar las operaciones correspondientes.
Es posible extrapolar este concepto a un "socket". De esta forma se asociaría un descriptor de "socket" con una aplicación específica en la red. Las funciones serían básicamente las mismas: open(): para abrir un socket o asociar una aplicación específica a un número (descriptor de socket), read(): para recibir o leer un mensaje desde la aplicación (remota), write(): para escribir un mensaje en la aplicación (remota). En las implementaciones actuales de "sockets", si bien el concepto del manejo de aplicaciones distribuidas con una interfase análoga a la del sistema de archivo sigue vigente, las funciones que se utilizan son diferentes[6]. Eso se debe a que en realidad, los parámetros que se deben especificar para una comunicación entre aplicaciones de red son mayores, como por ejemplo, si la aplicación es cliente o servidor, si el protocolo que se usará para transporte es orientado a conexión (TCP) o no-orientado a conexión (UDP), direcciones de red (IP), número de puertos, etc.

Tipos de Sockets

Al especificar el protocolo de transporte en el momento de la creación de un socket, se está determinando uno de los dos tipos básicos de sockets que existen, a saber[7]:

- *Sockets de Flujo ("Stream Sockets"):* cuando se utilizan estos sockets, se tiene una comunicación confiable y libre de errores, en donde los datos se transfieren de un extremo al otro conformando un flujo continuo de bytes ("stream"). Los "sockets" de este tipo son orientados a conexión, es decir, existe una conexión lógica entre los procesos o aplicaciones. Debido a estas características, estos tipos de "sockets" utilizan como protocolo de transporte a TCP.
- *Sockets de Datagrama*[8]: en este caso, la comunicación es potencialmente no-confiable y no es orientada a conexión. Es decir, cada datagrama es independiente del previo y del posterior, por lo que hipotéticamente los datos extremo a extremo, pueden ser recibidos fuera del orden en que fueron enviados (o incluso no recibidos). El protocolo de transporte asociado a estos tipos de "sockets" es UDP.

De acuerdo al tipo de "socket" que se seleccione para la comunicación, se tendrá entonces aplicaciones cliente/servidor orientadas a conexión o no-orientadas a conexión.
La secuencia de "system calls" que sigue una aplicación cliente/servidor en una comunicación orientada a conexión, es la mostrada en la figura 4.

[6] Es de particular interés destacar que la interfase original de sockets utilizaba exactamente las mismas seis "system calls" para el manejo de sockets que para el acceso al sistema de archivos, a saber: open, creat, close, read, write y lseek.

[7] Existen otros tipos de sockets como "raw" y "sequenced packets", este último sin implementación por ningún protocolo hasta la fecha.

[8] La palabra datagrama es utilizada comúnmente en la terminología de redes de computadoras, para denotar un paquete de datos.

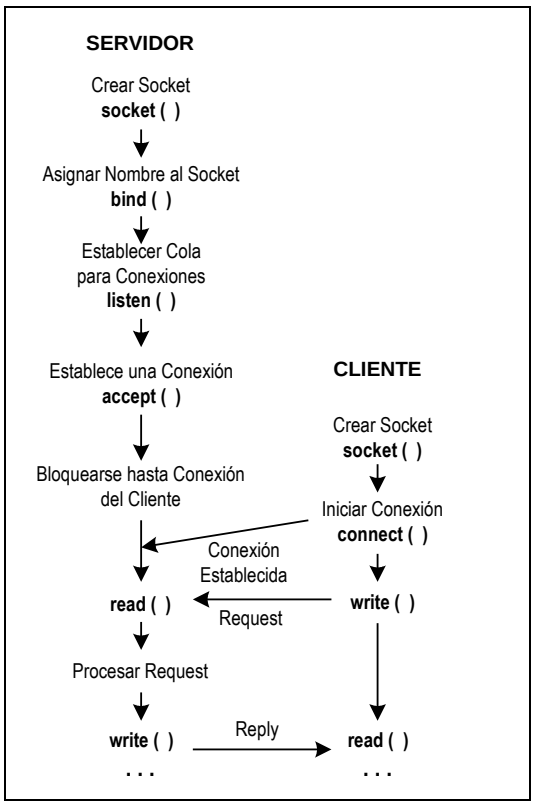

Figura 4: Comunicación Cliente/Servidor – Protocolo Orientado a Conexión

La figura 5, muestra el mismo diagrama para un protocolo no-orientado a conexión.

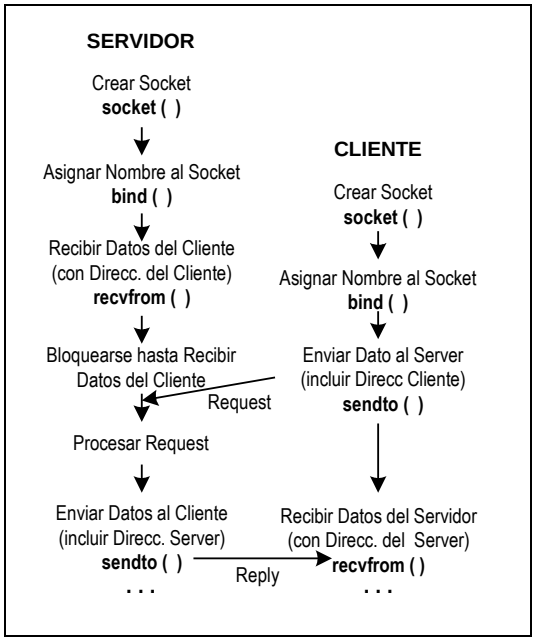

Figura 5: Comunicación Cliente/Servidor – Protocolo No-Orientado a Conexión

Se observa solo la similitud en un par de llamadas al sistema (socket() y bind()). De hecho, la naturaleza de las aplicaciones es radicalmente diferente.

Aplicaciones Orientadas a Conexión

La secuencia que sigue un par de aplicaciones cliente/servidor, mostrada en la figura 4, comenzando con la aplicación servidora: es básicamente la siguiente:

- Se crea un "socket" (socket()). Este "socket" no es más que una asociación entre la aplicación y el protocolo de transporte, ya que en esta instancia, no se asocia a dicho socket con una dirección IP, ni con un número de puerto.
- El "socket" es asociado a una dirección de red y a un número de puerto (que en este caso será un puerto bien conocido, puesto que es una aplicación servidora).
- Se crea una cola para las futuras solicitudes de conexiones remotas (listen())
- A partir de ese momento, la aplicación servidora está preparada para aceptar conexiones de aplicaciones clientes (accept()). Por lo tanto se auto bloquea esperando tales conexiones remotas.
- El cliente por su parte, crea su "socket" y se conecta al servidor (connect()).
- A partir de ese momento comienza el intercambio de datos, con sucesivas lecturas (read()) y escrituras (write()).

Aplicaciones No-Orientadas a Conexión

La secuencia de eventos entre una aplicación cliente/servidor no orientada a conexión, como muestra la figura 5, tiene algunos elementos en común, como por ejemplo, la creación de sockets. Sin embargo, tanto el servidor como el cliente, asocian a dicho socket con una dirección (a través de bind()). Se observa que el cliente no establece una conexión con el servidor. En su lugar, tanto el servidor como el cliente reciben y envían datagramas desde y hacia direcciones específicas (recvfrom() y sendto()).

Llamadas al Sistema de Sockets

Se detallará a continuación, en forma resumida, las principales llamadas al sistema ("system calls") de la interfase de "sockets".

socket

Sintaxis:

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

Para cualquier I/O de red, un proceso debe llamar a esta "system call". Con socket () se crea una nueva instancia de un "socket", actuando este como un punto de contacto para el intercambio de datos.

El “system call socket” devuelve un entero, similar a un descriptor de archivos. Por analogía se lo denomina sockfd[9].
Se debe observar que para obtener este descriptor, lo único que se proveyó fue en realidad la familia de protocolos (que en este trabajo siempre es TCP/IP) y el tipo de “socket[10]”. Por lo que para la asociación completa entre ambas aplicaciones mostrada en la figura 3, solo se especificó uno de los elementos de la quíntupla: el protocolo. Para que este descriptor pueda ser utilizado realmente es necesario especificar los 4 elementos faltantes de dicha quíntupla.

bind

Sintaxis:

```
int bind (int socketfd, struct
sockaddr *name, int addrlen);
```

Con este “system call” se asocia una dirección y número de puerto al “socket” creado. Es decir, se completa el par: (Dirección IP Local, Puerto Local) de la quíntupla mencionada.
bind() es usada por aplicaciones servidoras (orientadas o no orientas a conexión) y por clientes en aplicaciones no orientadas a conexión.
Esta función devuelve un cero en el caso de éxito o -1 en caso de error.
El puntero name a la estructura sockaddr, requiere de una explicación mas detallada. Esta estructura, que se encuentra definida en <sys.socket.h>, especifica una dirección genérica:

```
struct sockaddr {
      u_short  sa_family;
      char     sa_data[14];
};
```

Se debe tener en cuenta que la interfase de “sockets”, no está definida solamente para comunicación sobre TCP/IP. De esta manera, el primer elemento representa a la familia de direcciones (PF_INET para TCP/IP), y el segundo elemento, a la dirección específica del protocolo.

Para el caso de Internet, que es lo que se analiza en este artículo, se definen además las siguientes estructuras en <netinet/in.h>:

```
struct in_addr{
      u_long  s_addrr;
};

struct sockaddr_in {
      short    sin_family;
      u_short  sin_port;
      struct in_addr  sin_addr;
      char     sin_zero[8]
};
```

La figura 6 muestra la estructura genérica y la estructura utilizada para el dominio Internet.
El primer miembro de la estructura sockaddr_in, expresa nuevamente la familia de protocolos utilizada (PF_INET), el segundo miembro el número de puerto, el tercer miembro, una referencia a la estructura sin_addrr, que en realidad define la dirección IP (de allí que esta contiene un único elemento de 32 bits, que representa los 4 octetos de la dirección IP). El último elemento no se utiliza[11].

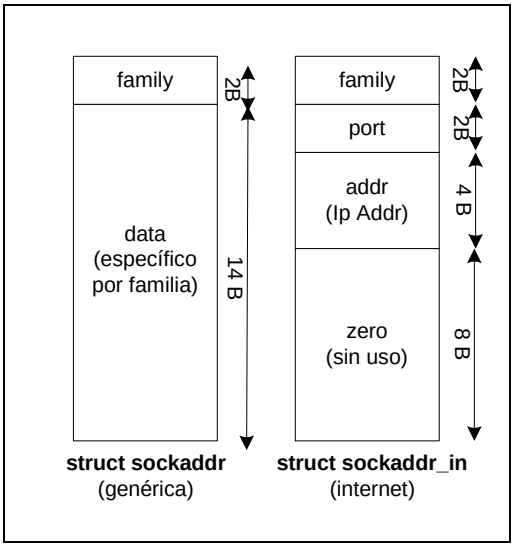

Figura 6: Estructuras de Direcciones

listen

Sintaxis:

```
int listen (int socket, int back-
log);
```

[9] Los mensajes de error que generan estas llamadas al sistema, normalmente corresponden a enteros negativos. No se cubren en el presente artículos. Los mismos pueden ser encontrados en cualquier manual de programación con sockets.
[10] . Este argumento puede ser SOCK_STREAM, para una comunicación orientada a conexión, o SOCK_DGRAM, para una comunicación no orientada a conexión.
[11] En el ejemplo que se presenta hacia el final del artículo, se muestra como se completa con datos esta estructura.

Esta función es utilizada por una aplicación servidora (en una comunicación orientada a conexión), para indicar que puede recibir conexiones. El argumento backlog indica el número de conexiones que pueden ser mantenidas en espera hasta que el servidor ejecuta el "system call" accept).[12]

accept

Sintaxis:

```
int accept (int sockfd, struct sockaddr *client, int *addrlen)
```

Con este "system call" el servidor espera la conexión con un cliente. Si es que existe alguna conexión en espera, se toma la primera de la cola y se crea **otro** descriptor de socket que será utilizado para la escritura y lectura de datos.
Este "system call" es del tipo bloqueante, es decir, que si no existen conexiones pendientes, la aplicación servidora se detiene en esta función esperando hasta que se produzca una conexión desde el cliente.
Los argumentos client y addrlen, son utilizados para obtener la dirección del proceso cliente que realizó la conexión al servidor. Por lo tanto este "system call" devuelve en realidad tres valores:

- Un entero que indica una condición de error (si es negativo) o un nuevo descriptor de "socket".
- La dirección del proceso cliente.
- La longitud de la estructura de dirección del cliente (para Internet: 16 Bytes).

Con este "system call" se completa la quíntupla mostrada en la figura 3.

connect

Sintaxis:

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

Con esta función el cliente inicia la conexión con el servidor. sockfd es el descriptor de "socket" devuelto por socket() (del proceso cliente) El segundo y tercer argumento son un puntero a la dirección del "socket" y su longitud.

Con TCP, connect resulta en el establecimiento de la conexión TCP entre el sistema local y remoto[13].
Se destaca que en una aplicación orientada a conexión, no es necesario por parte del cliente, realizar un "bind" previo a connect.

Transferencia de Datos y Finalización de la Conexión

Una vez que se estableció la comunicación entre el cliente y el servidor, la transferencia de datos se realiza con las invocaciones típicas del sistema de I/O, tales como read() o write().
Otros "system calls" más utilizados para transferencia de datos en redes son:

- *send (int sockfd, char *msg, int len, int flags).*
  Este "system call", sería el equivalente a write, ya que envía el mensaje **msg** a través del "socket" especificado por sockfd. La longitud del mensaje está indicado por len. El campo "flags" normalmente se inicializa en 0[14].
- *sendto (int sockfd, char *msg, int len, int flags, struct sockaddr *to, int addrlen)*.
  Este "system call" es utilizado en el caso de protocolo no-orientado a conexión. El argumento "to" referencia a una dirección específica al protocolo donde se enviará el mensaje (msg). Ya que esta dirección depende de la familia de protocolo, se debe especificar su longitud: argumento addrlen.
- *recv (int sockfd, char *buffer, int len, int flags)*.
  Este "system call" sería el equivalente a read, ya que recibe o lee el mensaje desde un descriptor de socket. Los argumentos son similares a los de send.
- *recvfrom (int sockfd, char *buffer, int len, int flags, struct sockaddr *from, int *addrlen)*.
  Esta primitiva especifica en el argumento "from", de quien pretende recibir el mensaje.

Se destaca el hecho, que send( ) y sendto ( ) son de naturaleza **no-bloqueante**, es decir que el proceso que la invoca, ejecuta la próxima sentencia en la aplicación; mientras que recv( ) y recvfrom ( ) son **bloqueantes**, es decir el proceso que invoca alguna de estas primitivas, se detiene esperando la llegada del mensaje.

[12] Este número es normalmente 5, indicando que 5 conexiones (remotas) pueden ser mantenidas en espera hasta la ejecución de accept ( ).

[13] Es decir, con este "network call" se realiza el típico "handshake" de tres vías utilizado para el establecimiento de una conexión TCP.

[14] Este campo permite mandar mensajes fuera de banda, mediante el mecanismo de "urgent pointer" que provee TCP.

Cuando los procesos concluyen la transferencia de datos, ejecutan el "system call" **close()**, que envía cualquier dato que se encuentra en la cola y finaliza la conexión TCP.

Otras Llamadas al Sistema

Existen un amplio número de llamadas al sistema, que son necesarias tener en cuenta para escribir rutinas de comunicación. Entre ellas se encuentran[15]:

- *bzero (char *dest, int nbytes):* escribe el número de bytes nulos especificados (nbytes) en el destino especificado.
- *inet_aton (char *ptr):* convierte la cadena de caracteres que identifica una dirección IP en la notación x.y.w.z a un número binario.
- *inet_ntoa (struct in_addr inaddr):* realiza la acción inversa a la función anterior.
- *gethostbyname(char *name):* busca y traduce el nombre de host (name) a una dirección. Para ello utiliza el esquema de resolución de nombres (archivos hosts o DNS).
- *htons (u_short hostshort)*[16]: convierte datos binarios de orden de bytes de host en datos binarios de orden de bytes de red. Es decir convierte 16 bits binarios a "big-endian".[17]
- *htonl (u_long hostlong):* idem a la función anterior pero para 32 bits.
- *ntohs (u_short netshort):* inverso de htons.
- *ntohl (u_long netlong):* inverso de htonl.

Ejemplo

A continuación se brinda un ejemplo simple de una aplicación distribuida cliente/servidor. Para este ejemplo, se utilizó un protocolo orientado a conexión, con el uso de "sockets" de flujo ("Stream Sockets"). El programa se realizó utilizando el lenguaje de programación C++ sobre sistema operativo Linux.

El ejemplo implementa el diagrama mostrado en la figura 4. La aplicación servidora, mostrada en la figura 7, está continuamente esperando mensajes que le envían las aplicaciones clientes, mostrada en la figura 8. Una vez que le llega uno de tales mensajes lo muestra en pantalla.

Este ejemplo, aunque simple, muestra la mayoría de los "system calls" descriptos en el artículo y además ejemplifica la estructura general de las aplicaciones cliente/servidor sobre TCP/IP que utilizan la interfase de "sockets".

---

[15] Para una descripción mas detallada, se puede consultar la bibliografía citada, o bien utilizar las páginas del manual de Linux.

[16] La función recibe su nombre de: **htons h**ost **to n**etwork **s**hort. Las otras funciones siguen la misma nomenclatura.

[17] Estas funciones se utilizan porque la forma de ordenamiento de los bytes de los procesadores no es uniforme. La notación de orden de bytes de red, sigue la notación "big-endian", mientras que algunos procesadores, como los procesadores Intel, siguen la notación "little-endian". Con estas funciones, se convierte (si es necesario) a la notación "big-endian" el orden de los bytes de red.

```
#include <stdio.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>

#define PORT 3550         //Puerto de conexion
#define BACKLOG 5         //Numero de conexiones permitidas

int main ()
{ int fd1, fd2;                 //Descriptores de archivo
  int tama_sin;
  struct sockaddr_in servidor;   // Info de la direccion de servidor
  struct sockaddr_in cliente;    // Info de la direccion del cliente
  printf ("Servidor\n");
  if ((fd1 = socket (AF_INET, SOCK_STREAM, 0)) < 0)
    { printf ("Error en Socket()\n");
      exit (-1);
    }
  servidor.sin_family = AF_INET;
  servidor.sin_port = htons (PORT);     // Convierte PORT al numero de red
  servidor.sin_addr.s_addr = INADDR_ANY;      // coloca ip automaticamente
  bzero (&(servidor.sin_zero), 8);      //Coloca ceros en resto estructura
  if (bind (fd1, (struct sockaddr *) &servidor, sizeof (struct sockaddr)) < 0)
    { printf ("Error en Bind()");
      exit (-1);
    }
  if ((listen (fd1, BACKLOG)) < 0)  // espera conexiones en un socket
    { printf ("Error en listen()\n");
      exit (-1);
    }
  while (1)
    { tama_sin = sizeof (struct sockaddr_in);
      if ((fd2 = accept (fd1, (struct sockaddr *) &cliente, &tama_sin)) < 0)
       { printf ("Error en accept()\n");
         exit (-1);
       }
      printf ("Tenes una conexion desde %s \n", inet_ntoa (cliente.sin_addr));
      send (fd2, "Bienvenido al servidor", 24, 0);
      close (fd2);
    }
  close (fd1);
}
```

Figura 7: Código en C++ de Aplicación Servidora[18]

```
#include <stdio.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>

#define PORT 3550               //Puerto de conexion
#define MAX_TAMA_DATO 100       //tamaño maximo en bytes a transmitir

int main ()
{   int fd, numbytes;                  //Descriptores de archivo
    char buf[MAX_TAMA_DATO], ip[100];  //Buffer temporario
    struct hostent *he;
    struct sockaddr_in servidor;       //Info de la direccion del servidor
    printf ("Cliente \nHost o Dirección IP: ");
    scanf("%s",&ip);
    if((he = gethostbyname(ip)) == NULL) //Puede resolver el nombre o ip ?
      { printf ("Error en Gethostbyname()\n");
       exit(-1);
```

[18] Para poder compilar este código, utilizar la siguiente línea de comando: $ gcc *filename.c* –o *salida*

```
      }
    if((fd = socket(AF_INET,SOCK_STREAM,0))<0)
      { printf ("Error en Socket()\n");
       exit(-1);
      }
    servidor.sin_family = AF_INET;
    servidor.sin_port = htons (PORT);
    servidor.sin_addr = *((struct in_addr *)he->h_addr);
    bzero (&(servidor.sin_zero), 8);
    if ( connect(fd,(struct sockaddr *)&servidor,sizeof(struct sockaddr))<0)
      { printf ("Error en Connect()");
        exit(-1);
      }
   if(( numbytes=recv(fd,buf,MAX_TAMA_DATO,0))<0)
      { printf ("Error en recv()\n");
        exit(-1);
      }
   buf[numbytes]='\0';
   printf("Mensaje del servidor: %s\n",buf);
   close(fd);
}
```

Figura 8: Código en C++ de Aplicación Cliente

Conclusión

Con la presente publicación, se concluye que la programación de aplicaciones en redes utilizando la interface de sockets, constituye una herramienta simple de utilizar y de mucha potencialidad. Con esta interface se pueden construir aplicaciones distribuidas en redes del tipo cliente servidor, especialmente diseñadas para trabajar en Internet.
Se dejaron algunos temas para futura publicaciones o investigación por parte de los lectores, como por ejemplo, el manejo de servidores concurrentes e iterativos, el análisis de “system call” más avanzadas, la implementación de un Internet Superserver, y otras.

Bibliografía

[1] Protocolo TCP/IP utilizado en Internet - Roberto Fanjul y Pablo Rovarini – Revista de Ciencias Exactas e Ingeniería – UNT – Año 10 - N° 20 – Octubre 2001.
[2] Interprocess Communications in UNIX, John Gray, 2nd. Edition, Prentice Hall, 1998.
[3] Linux Socket Programming – Sean Walton – Sams – 2001.
[4] Unix Network Programming - W. Richard Stevens - Prentice Hall - 1990.
[5] An Introduction to Socket Programming - Reg Quinton - http://www.uwo.ca/its/doc/courses/notes/socket/ - 1997
[6] Notas en Sistemas Operativos - Sergio Saade - Top Graph - 2000.
[7] Internetworking with TCP/IP Vol. I: Principles, Protocols and Architecture, 2nd Edition - Douglas Comer - Prentice Hall - 1991.
[8] Internetworking with TCP/IP Vol. III: Principles, Protocols and Architecture - Douglas Comer & David Stevens - Prentice Hall - 1.994.
[9] Redes Globales de Información con Internet y TCP/IP – Douglas Comer – Prentice Hall – 1996.
[10] TCP/IP Network Administration – Cray Hunt – O´Reilly & Associates, Inc. - 1.992

Sergio Saade

Sergio D. Saade recibió el título de Ingeniero Electricista (Orientación Electrónica) en la Facultad de Ciencias Exactas y Tecnología (FACET) de la Universidad Nacional de Tucumán (UNT) en Mayo de 1.986.
Entre 1988 y 1990, financiado con becas de la "Rotary Foundation" y de la "Organización de Estados Americanos (OEA)" realizó estudios de postgrado en el departamento de "Electrical & Computer Enginnering" de la "University Of California”, Estados Unidos de América, obteniendo el título de "Master of Science".
Desde 1991 se desempeña como Profesor Asociado en el Depto. de Electricidad, Electrónica y Computación, de la FACET, UNT, estando a cargo del dictado de las materias “Sistemas Operativos” y “Redes de Area Local” de la carrera Ing. en Computación.
En la actualidad, además de sus funciones docentes, está a cargo de la Dirección de la Carrera de Ingeniería en Computación.

José L. Borelli

José L. Borelli recibió el titulo de Técnico en Computación en la Escuela Técnica N°3 Gral. Martín Miguel de Güemes de la ciudad de Salta en el año 1.996.

En 1997 ingresó a la carrera de Ingeniería en Computación en la Facultad de Ciencias Exactas y Tecnología, de la Universidad Nacional de Tucumán, en donde actualmente esta cursando las últimas asignaturas y realizando su Trabajo de Graduación que versa sobre Seguridad en Redes Linux.

Desde 1999 se desempeña como ayudante estudiantil de las cátedras de “Bases de Computación”, “Programación 1” y “Programación 2” y desde 2001 como pasante en el área de soporte en una empresa de Computación y Comunicaciones de Tucumán.