Descripción y uso de sockets para la comunicación de procesos

Dr. Roberto Gómez Cárdenas
ITESM-CEM, Dpto. Ciencias Computacionales

2 de octubre de 1998

El presente documento representa un borrador de lo que será el capítulo de un libro. Acaba de ser terminado y debido a la urgencia de tiempo no se presto mucha atención a la forma, pero el fondo es muy bueno. Cualquier aclaración, comentario o correción se agradecera que se le notifique al autor vía personal o a través del correo electrónico, (rogomez@campus.cem.itesm.mx).

1 Introducción

Las llamadas de sistema hasta ahora vistas permiten comunicar y/o sincronizar procesos que se encuentran ejecutando en la misma máquina. El desarrollo de las redes de computadoras y la importancia que tienen hoy en día hace necesario el poder comunicar procesos que residen en máquinas diferentes.

Hoy en día existen dos formas de comunicar a dichos procesos, el envío de mensajes y la llamada de procedimientos remotos; cada una cuenta con sus ventajas y desventajas. Ahora bien, tanto una como la otra reposan sobre el concepto de sockets, sin embargo los sockets como tal cuentan con primitivas de envío/recepción de mensajes.

Un socket es un punto de comunicación a través del cual un proceso puede emitir o recibir información. Desde el punto de vista del modelo OSI de la ISO es una interfaz entre la capa aplicación y transporte.

Para un proceso un socket no es otra cosa que un descriptor de archivos, lo que provoca que las operaciones de envio/recepción de mensajes sean parecidas a las de lectura/escritura en un archivo. Aparte, esto permite la redirección de la entrada y salida estándar, utilizar aplicaciones estándares sobre la red, y que todo nuevo proceso (*fork()*) herede los descriptores de sockets de su padre.

Otra de las ventajas de los sockets esta relacionada con que los programadores dediquen su atención al envío/recepción de mensajes, y al tratamiento de estos. De esta forma el usuario no tiene por que preocuparse del canal físico de comunicación, (capa física de la OSI-ISO), la forma de codificación de señales para disminuir probabilidad error en la transmisión (capa enlace), los nodos de la red por los cuales tiene que pasar (capa red), formar paquetes de la información a transmitir y tampoco por buscar la ruta que una la computadora origen con la destino (capa trnsporte).

Los sockets proporcionan un punto de comunicación entre dos procesos que residen en computadoras diferentes. Sin embargo, para poder diseñar sistemas de comunicación con sockets es necesario entender varios conceptos relacionados con la comunicación entre computadoras en TCP/IP.

2 El modelo cliente/servidor

A raíz de la aparición de las redes surge un nuevo modelo de cooperación entre entidades, conocido como cliente/servidor. En dicho modelo una entidad, el servidor, se dedica a esperar por peticiones de servicio. Una vez recibidas las atiende y, si es necesario, envia una respuesta de acuerdo al servicio. La entidad cliente es la que envia la o las peticiones.

Los sockets funcionan bajo este paradigma, se tiene que definir un código para el cliente y otro para el servidor. Para que el cliente pueda enviar peticiones al servidor es necesario que el cliente conozca los datos del servidor: su *dirección IP* y el *número de puerto* a través del cual se van a comunicar. El servidor debe de estar "escuchando" en este puerto para leer las peticiones que le lleguen, atender dichas peticiones y regresar la respuesta (si hay alguna) por el mismo lugar en que llegó la petición.

2.1 Los puertos y las direcciones IP

Para que una persona pueda comunicarse con otra vía telefónica necesita saber dos cosas, la primera es el nombre de la persona y la segunda es el número de teléfono de esta. El principio es el mismo para que dos procesos, que residen en máquinas diferentes, puedan comunicarse entre ellos. Es necesario conocer el nombre de la máquina o la dirección IP de esta, así como un número de puerto.

En un modelo cliente/servidor, para que el cliente solicite un servicio necesita saber quien otorga dicho servicio, y el servidor necesita sabe a quien le va a enviar su respuesta. Para referenciar a una computadora se utiliza su dirección IP. Esta dirección puede estar expresada de dos formas, a través de un conjunto de nombres separados de un punto, o de una serie de núnmeros también separados por un punto. Por ejemplo podemos hablar de la dirección *148.241.15.1* o del nombre *francia.cem.itesm.mx*

Una vez que se tiene identificada la máquina que ofrece el servicio, se debe de precisar cual de todos los procesos que residen en la computadora proporcionan el servicio. Los procesos son identificados a partir de un número de proceso el cual se asigna de forma local; por lo que es difícil que la máquina cliente conozca el número de proceso que presta un servicio determinado. La solución a este problema es que en lugar de dirigirnos a un proceso la petición o mensaje se envíe a un número de puerto. Este número de puerto se encuentra asociado a un servicio, y un proceso (o varios dependendiendo del servicio) se encontraran escuchando el puerto[1]. Sin embargo existen servicios (ftp, telnet, sendmail) cuyas especificaciones son conocidas por todo el mundo.

Resumiendo, un puerto es un punto de comunicación lógico de una computadora. El puerto es representado por un número, del 0 al 1024 se consideran reservados para el sistema, dejando el resto para los programadores.

Para poder comunicar dos procesos es necesario conocer las direcciones y/o nombres de las máquinas que componen la red, por lo que es importante contar con funciones y bibliotecas que nos permitan conocer diferentes aspectos y caracteríscticas de la red local. En la siguiente sección se describen diferentes llamadas de funciones que nos permiten contar con las bases para poder comunicar dos procesos vía sockets.

[1] Una analogía la encontramos en lo que es la atención en un banco, nosotros nos dirijimos a una ventanilla determinada, no nos interesa quien esta detras de la ventanilla sino el servicio que se ofrece a través de dicha ventanilla

2.2 Obteniendo información: los resolvers y otras llamadas

En la sección anterior se estableció que para poder utilizar sockets es necesario conocer el número IP de una máquina. En varias aplicaciones este tipo de información varía de acuerdo a la red o a la máquina sobre la cual se este ejecutando la aplicación.

Para poder obtener información acerca de las diferentes máquinas que componen una red de computadoras, se cuenta con funciones conocidas con el nombre de *resolvers*. Los resolvers tienen como caracteriística principal el hecho de que no regresan información puntual, sino un apuntador a una estructura que contiene toda la información asociada con la entidad relacionada con el resolver.

En la figura 1 se muestra el uso de los resolvers asociados para la obtención de información asociada a un host.

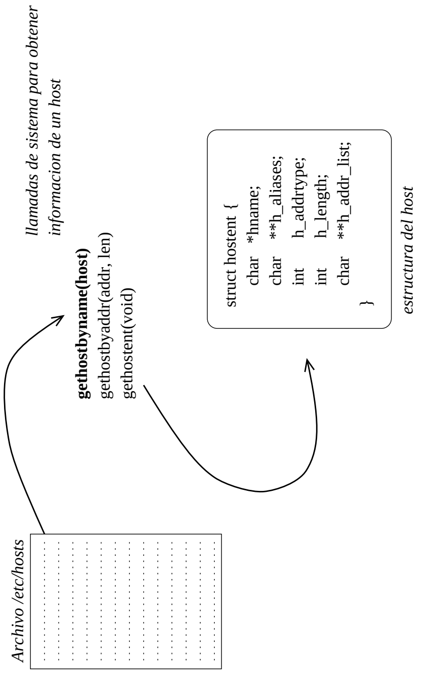

Figura 1: *Ejemplo de un resolver*

Los resolvers forman parte de NIS (Network Information Service) que se encarga de distribuir a todas las computadoras de una red local, la información contenida en los archivos de configuración. La función principal de los resolvers es proporcionar, a nivel programación, un medio para obtener esta información. En lo que concierne a este capítulo se hara referencia a dos archivos: */etc/hosts* y */etc/services*. El primero almacena las direcciones, nombres y alias de las computadoras que forman la red local. El segundo archivo almacena los nombres de los servicios, sus protocolos, y el número de puerto a través del cual se va a proporcionar el servicio.

2.2.1 Obteniendo datos del host

Los nombres de las computadoras que forman una red local estan reunidos en un archivo llamado */etc/hosts*. En ese archivo se encuentran las direcciones de las máquinas, sus nombres y sus alias. Un ejemplo del contenido de dicho archivo se presenta abajo.

```
#
# Internet host table
#
127.0.0.1 localhost
148.241.61.15 brasil loghost
```

Para poder obtener información acerca de los hosts de una red se cuenta con diferentes funciones. Esta funciones difieren entre ellas por el parámetro de entrada a partir del cual se va a buscar la información. La sintaxis de las llamadas es la siguiente:

1. `struct hostent *gethostbyaddr(const void *addr, size_t len, int type);`
2. `struct hostent *gethostbyname(const char *name);`
3. `struct hostent *gethostent(void);`

La llamada *gethostbyaddr()* obtiene información de un host a partir de una dirección, del tamaño de dicha dirección y del tipo de la dirección. También es posible conseguir información a partir del nombre de la máquina con la llamada *gethostbyname()*. Si se quiere recorrer toda la base de datos, registro por registro, se debe de utilizar *gethostent()*. Dicha llamada lee el siguiente registro en la base de datos.

Como puede constarse todas las llamadas regresan un apuntador a un estructura. Dicha estructura contiene la información acerca de las computadoras de la red local y se ecuentra organizada de la siguiente forma:

```
 struct hostent {
    char      *h_name;              /* nombre oficial de un host */
    char      **h_aliases;          /* lista de alias */
    int       h_addrtype;           /* tipo de direccion del host */
    int       h_length;             /* longitud de la direccion */
    char      **h_addr_list;        /* lista de direcciones del servidor de nombres */
    #define   h_addr h_addr_list[0]  /* direccion para compatibilidad */
};
```

Es importante no olvidar que los resolvers regresan un apuntador a esta estructura, y que si se quiere un campo determinado es necesario accederlo directamente. También vale la pena aclarar que para agregar nuevos campos al archivo */etc/hosts* es necesario contar con ciertos privilegios.

2.2.2 Obteniendo datos de los servicios

Un servicio esta definido por un nombre, un número de puerto y un protocolo. Existen servicios (ftp, telnet, portmapper) usados frecuentemente que tienen establecidos por default un número de puerto (0-1024) y un protocolo. Dichos servicios se encuentran definidos en el archivo */etc/services*. En la parte de abajo se muestra un ejemplo del contenido de dicho archivo.

```
#ident  "@(#)services   1.16    97/05/12 SMI"   /* SVr4.0 1.8   */
#
# Network services, Internet style
#
tcpmux          1/tcp
echo            7/tcp
```

```
echo            7/udp
discard         9/tcp           sink null
discard         9/udp           sink null
systat          11/tcp          users
daytime         13/tcp
daytime         13/udp
netstat         15/tcp
chargen         19/tcp          ttytst source
chargen         19/udp          ttytst source
ftp-data        20/tcp
ftp             21/tcp
telnet          23/tcp
smtp            25/tcp          mail
time            37/tcp          timserver
time            37/udp          timserver
domain          53/udp
domain          53/tcp
bootps          67/udp                          # BOOTP/DHCP server
bootpc          68/udp                          # BOOTP/DHCP client
hostnames       101/tcp         hostname        # usually to sri-nic
sunrpc          111/udp         rpcbind
sunrpc          111/tcp         rpcbind
```

Las siguientes funciones permiten conocer información acerca de un servicio (o de todos si es necesario) que proporciona una red. La sintaxis de cada una de las funciones es la siguiente:

1. `struct servent *getservbyname(const char *name, const char *proto);`
2. `struct servent *getservbyport(int port, const char *proto);`
3. `struct servent *getservent(void);`

La primera de ellas nos permite conseguir información a partir del nombre del nombre del servicio y del protocolo. La función *gerservbyport()* recibe como parámetros un número de puerto y un protocolo. Por último la función *getservent()* lee la siguiente entrada de la base de datos, lo cual nos puede servir para obtener información acerca de todos los servicios existentes.

Todas las llamadas regresan un apuntador a la estructura *struct servent*, la cual cuenta con los siguientes campos:

```
struct servent {
    char  *s_name;       /* nombre oficial del servicio */
    char  **s_aliases;   /* lista de alias */
    int   s_port;    /* numero de puerto */
    char  *s_proto;      /* protocolo a usar */
};
```

Al igual que el archivo */etc/hosts* se requiere de permisos especiales para poder dar de alta un servicio en el archivo de configuración */etc/services*.

2.2.3 Obteniendo el nombre del host local

Una de las características principales de un sistema cliente/servidor es la de portabilidad. Para lograr esto es necesario que los programas conozcan el nombre de la máquina sobre la cual se estan ejecutando. Existen cuatro formas de obtener el nombre de la máquina sobre la cual se esta ejecutando un proceso:

i. El comando *hostname*

El comando regresa el nombre de la máquina sobre la que se esta ejecutando; el problema es que es a nivel comando y no de llamada de sistema. Una posible solución es utilizar la llamada *system()* para redireccionar la salida del comando a un archivo y después leer el contenido de dicho archivo. El código siguiente es un ejemplo de obtención de nombre a partir del comando *hostname*:

```
#include <stdio.h>

main()
{
  FILE *fd;
  char nombre[50];

  system("hostname > /tmp/name");
  fd=fopen("/tmp/name","r");
  fscanf(fd,"%s",nombre);
  unlink("/tmp/name");
}
```

ii. El archivo */etc/hostname.e01*

Este archivo contiene el nombre de la máquina, en realidad lo que hace el comando anterior es leer el contenido de este archivo. A nivel programa es posible abrir el archivo y leer su contenido para conocer el nombre de la máquina.

iii. El comando *uname*

Con la opción $-n$ del comando *uname* es posible obtener el nombre de la máquina. El problema es el mismo que en el primer de los casos y la solución es la misma: redireccionar la salida a un archivo y después leer su contenido.

iv. La llamada de sistema *gethostname(char *name, int namelen)*

Es la forma más simple de obtener el nombre. La llamada regresa el nombre estándar de la máquina sobre la cual se esta ejecutando. Recibe como parámetros el tamaño de la variable donde va a regresar el nombre, *namelen* y la variable donde se va a almacenar el nombre de la computadora, (*name*). La llamada regresa 0 si todo salió bien y -1 si hubo algún error. Un ejemplo de uso se presenta a continuación:

```
#include <errno.h>

main()
{
   char nombre[256];
   int  n=256;

   if (gethostname(nombre, n) < 0)
      printf("Error no. %d \n",errno);
   else
      printf("El nombre es: %s \n", nombre);
}
```

La elección entre una y otra dependerá de la aplicación y del sistema en que se este trabajando.

2.2.4 Conviertiendo diferentes formatos de dirección

En una comunicación es importante que los datos que viajan a través de la red sean independientes de la arquitectura sobre la cual se esta ejecutando. Un mensaje esta formado por una secuencia de bytes la cual representa cierto tipo de información, y la forma de interpretar esos bytes puede diferir entre dos arquitecturas. Para no tener que realizar traducciones entre todas las arquitecturas existentes, se eligió realizar tan sólo dos traducciones: del host a red y de red a host. Para llevar a cabo lo anterior se cuenta con las siguientes llamadas:

1. `in_addr_t htonl(in_addr_t hostlong);`
2. `in_port_t htons(in_port_t hostshort);`
3. `in_addr_t ntohl(in_addr_t netlong);`
4. `in_port_t ntohs(in_port_t netshort);`

Estas funciones convierten cantidades de 16 y 32 bits entre formatos red y host. Las dos primeras transforman un formato host a un formato red: *htonl*() para enteros largos y *htons*() para enteros cortos. Las dos últimas convierten de red a host: *ntohl()* se utiliza en el caso de enteros largos y *ntohs()* para enteros cortos. Todas reciben como parámetro el valor a convertir y regresan el valor en el formato solicitado.

Para poder utilizar estas llamadas se requiere del archivo de biblioteca `#include <arpa/inet.h>`

2.3 Protocolos orientados conexión y no-conexión

Existen dos formas de realizar una comunicación entre dos personas que se encuentran separadas por una distancia geográfica considerable: el teléfono y el correo. En el teléfono la comunicación empieza a llevarse a cabo una vez que se estableció una línea de comunicación (hasta que la otra persona descuelga el teléfono). Por el otro lado en el correo no es necesario que se establezca una línea de comunicación, una persona deposita su carta (su mensaje) con los datos de la persona a la que va dirigida y el sistema postal se encarga de hacerla llegar a la otra persona. Cabe la pena aclarar que nada nos asegura que la carta va a llegar a su destino, sin embargo el emisor puede dedicarse a otras tareas en lo que la respuesta llega. Cuando se usa el teléfono la perdida de mensajes es nula, sin embargo la persona no puede dedicarse a otras tareas mientras se comunica con su interlocutor (aunque en la vida real hay personas que si pueden hacerlo).

El mismo panorama descrito anteriormente se aplica a una comunicación entre dos procesos a través de sockets. En el caso del teléfono se esta hablando de protocolos *orientados conexión*, y en el caso del correo estamos hablando de protocolos *orientados no-conexión*.

En el caso de protocolos orientados conexión, el cliente debe de enviarle al servidor un mensaje de solicitud de conexión. Cuando el servidor recibe dicho mensaje establece una línea de comunicación con el cliente, a través de la cual viajan todos los mensajes entre las dos entidades. La ventaja que presenta este tipo de protocolos es la confiabilidad en la comunicación, ya que es difícil que se presente perdidas de mensajes. Por el otro lado, presenta la desventaja de que consume más recursos y que requiere una completa atención por parte del servidor. El protocolo más utilizado en modo conexión es el TCP (Transmission Control Protocol).

Cuando se esta usando un protocolo orientado no conexión, el cliente envía su petición de servicio directamente al servidor, sin realizar ninguna solicitud de conexión. El servidor toma el mensaje, realiza el servicio y, si es necesario, envia la respuesta al cliente. La ventaja es que no requiere de una atención continua de parte del servidor, sin embargo la posibilidad de perdida de mensaje es mayor. Por otro lado, existe la posibilidad de que alguien se haga pasar por otra persona, por lo que es necesario verificar la identidad del emisor de un mensaje (se podría confundir una respuesta con una petición o viceversa). El protocolo UDP (User Datagram Protocol) es el más utlizado en comunicaciones orientadas no-conexión.

3 La comunicación cliente/servidor con sockets

Todo sistema cliente/servidor consta de dos programas: un programa servidor que va a escuchar las peticiones, atenderlas y darles una respuesta. El otra programa, el cliente, se encarga de enviar la petición, esperar una respuesta y procesar dicha respuesta. Esto trae como consecuencia que todo sistema construido a partir de sockets conste de dos partes la del cliente y la del servidor, el diseño de uno sin el otro no tiene sentido. La figura 2 presenta los pasos a seguir por el cliente y por el servidor para que puedan comunicarse.

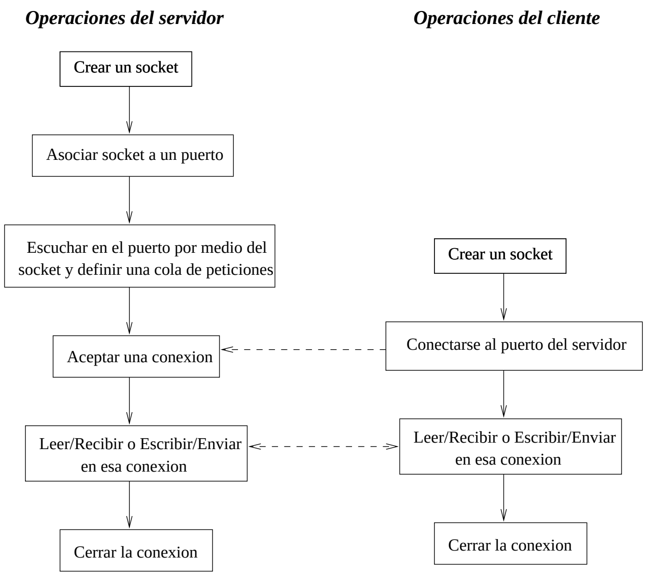

Figura 2: *Pasos a seguir para poder establecer una comunicación con sockets*

A pesar de que, tanto el cliente como el servidor, deben de seguir todos los pason descritos en la figura anterior, el diseño de ambos tiene que tomar en cuenta diferentes aspectos. Existen varias estrategias para construir un servidor y se deben de considerar otros aspectos para su diseño. De la misma forma, el cliente requiere de ciertas consideraciones a tomar en cuenta, como por ejemplo el time-out en caso de que el servidor este caído.

3.1 Consideraciones a tomar en cuenta para una comunicación cliente/servidor

Para poder establecer una comunicación entre cliente y servidor es necesario que los dos esten de acuerdo en tres aspectos:

1. La familida o dominio de direcciones.

2. Los tipos de sockets.
3. El tipo de protocolo a utilizar: orientado conexión u orientado no conexión, y el protocolo en sí mismo.

En secciones anteriores se vió la diferencia entre los protocolos orientados conexión y los orientados no-conexión. En la siguiente sección se tocarán todo lo relacionado con las diferentes familias de sockets existentes.

3.1.1 Las familias de direcciones o dominios

Se refiere al formato que las direcciones le van a dar al socket así como a los diferentes protocolos soportados por las comunicaciones que se realizan vía sockets de un determinado dominio. La estructura general de una familia de direcciones es:

```
struct sockaddr {
    u_short sa_family;         /* familia direccion   */
    char    sa_data[14];
};
```

Muchas de las llamadas de sistema relacionadas con sockets utilizan como parámetro el tipo de familia. Para evitar definir una llamada de sistema por cada tipo de familia, se utiliza el tipo genérico anterior.

Para una aplicación en particular se deberá usar la estructura correspondiente a la familia que se esta ulizando. Las dos principales familias utilizadas son la familia Unix y la familia Internet.

La familia o **dominio Unix** sirve para comunicar dos procesos que residen en la misma computadora. El punto de contacto entre los dos es un archivo. Como todos los sockets cuenta con una estructua en la cual se almacenan toda la información necesaria para el manejo de este. Dicha estructura es la siguiente:

```
struct sockaddr_un {
    short   sun_family;             /* AF_UNIX */
    char    sun_path[108];          /* Nombre del path */
};
```

Al campo *sun_family* se le debe asignar el valor AF_UNIX, mientras que el campo *path* recibe el path y el nombre del archivo que va a servir de puente de comunicación entre el cliente y el servidor.

La familia o **dominio Internet** tienen por objeto el comunicar dos procesos que residen en máquinas diferentes. La estructura que almacena los datos necesarios para lograr lo anterior es la siguiente:

```
struct sockaddr_in {
    short   sin_family;        /* familia direccion   */
    u_short sin_port;          /* numero de puerto */
    struct  in_addr sin_addr;  /* direccion */
    char    sin_zero[8];       /* relleno */
};
```

El campo *sin_family* se usa para almacenar el tipo de socket y debe de asignarsele el valor AF_INET. El campo *sin_port* almacena el puerto a través del cual se va establecer la comunicación y el campo *sin_addr* contiene la dirección de la máquina. Como puede apreciarse este campo esta definido por la estructura:

```
struct in_addr{
    u_long   s_addr;
};
```

el último campo, *sin_zero*, es un campo de relleno utilizado con el fin de que las diferentes familias tengan el mismo tamaño.

Existen otros tipos de familias de direcciones entre las que encontramos:

- AF_NS: protocolo de XEROX NS
- AF_CCITT: protocolos CCITT, protocolos X.25, etc
- AF_SNA: IBM SNA (Systems Network Architecture), arquitectura de redes de sistemas utilizado por IBM
- AF_DECnet: redes DEC
- AF_APPLETALK: soporte protocolo APPLETALK

una lista entera de todas las familias se puede ver en el archivo */usr/include/sys/socket.h*

3.1.2 Los tipos de sockets

Existen diferentes tipos de sockets, un tipo de socket define un tipo de propiedades de las comunicaciones en las cuales esta implicado, (ver [Ri92]).

En un sistema de comunicaciones es importante contar con cierto número de propiedades. Las más importantes son:

1. *La confiabilidad.* Se refiere al hecho de que no se pierda ningún dato.
2. *Orden emisión recepción.* Es importante que los datos lleguen en el mismo orden en que fueron transmitidos.
3. *La no duplicación de datos.* Sólo llega al destino un ejemplar de cada dato emitido.
4. *Comunicación modo conexión.* Debe ser posible establecer este tipo de comunicación.
5. *Conservación límites de mensajes.* Los límites de los mensajes emitidos se pueden encontrar en el destino.
6. *El envío de mensajes "urgentes".* Corresponde a la posibilidad de enviar datos fuera del flujo normal y por consecuencia accesibles inmediatamente (también conocidos como datos fuera de flujo u "out of band").

Una transmisión de información normal entre dos entidades cuenta con las cuatro primeras propiedades, pero no las dos últimas. Dependiendo de la aplicación seán las propiedades que se requieran y, por ende, el tipo de socket elegido. Entre los tipos de sockets disponibles encontramos:

- **El tipo SOCK_DGRAM** o **datagrama**
  Corresponde a los sockets destinados a la comunicación en modo no conexión para el envío de datgarmas de tamaño limitado. Las comunicaciones que utilizan este tipo de socket tienen la propiedad cinco.

- **El tipo SOCK_STREAM** o **stream**
  Los sockets de este tipo permiten comunicaciones fiables en modo conexión (propiedades a, b, c y d) y eventualmente autorizan (según el protocolo aplicado) los mensajes fuera de banda.

- **El tipo SOCK_RAW**
  Permite el acceso a los protocolos de más bajo nivel (por ejemplo el protocolo IP en el dominio Internet). Su uso esta reservado al superusuario y permite implementar nuevos protocolos.

- **El tipo SOCK_SEQPACKET**
  Corresponde a las comunicaciones que poseen las propiedades de la uno a la cinco. Las comunicaciones se encuentran en el dominio XEROX NS.

En este capítulo se estudiarán los dos tipos de sockets más utilizados, es decir los dos primeros.

3.1.3 Los protocolos

Independientemente de si se usa un protocolo orientado conexaón o no-conexión es importante que el cliente y el servidor se comuniquen usando el mismo protocolo.

Existe una gran variedad de protocolos que se pueden utilizar para que dos procesos puedan comunicarse. Los dos más utilizados son TCP (Transmission Control Protocol) y UDP (User Datagram Protocol). La eleccción del protocolo a utilizar depende de la familia de direcciones a utilizar y del tipo de socket. En la tabla 3 se presenta una relación de los diferentes tipos de familias que existen.

	AF_UNIX	**AF_INET**	**AF_NS**
SOCK_STREAM	*si*	*TCP*	*SPP*
SOCK_DGRAM	*si*	*UDP*	*IDP*
SOCK_RAW		*IP*	*si*
SOCK_SEQPACKET			*SPP*

Figura 3: *Tabla relación protocolos y tipo de sockets*

Los recuadros de la tabla en los que aperece la palabra *si*, se refiere a que si existe una relación pero el protocolo no tiene nombre. Las casillas vacías se refieren a los casos en los que no hay nada implementado.

El usuario determina el protocolo a utilizar o puede dejar que el sistema lo elija de forma automática de acuerdo a la familia de direcciones y al tipo de sockets utilizado.

3.2 Aspectos a considerar en el diseño de un servidor

El servidor es el encargado de esperar las peticiones, de atenderlas y de enviar una respuesta a quien las envió. En el caso de que exista más de un cliente realizando peticiones a un servidor, este debe de ser capaz de diferenciar el origen de las diferentes peticiones.

En el caso de trabajar con protocolos orientados conexión la solución es proporcionar a cada cliente un canal privado de comunicación a través del cual se reciban las peticiones y se envíen las respuestas. Por el otro lado, con protocolos orientados no conexión, es necesario identificar al emisor para poder otorgarle, o negarle, un servicio. Esto último se realiza a través de llamadas de sistema como *recvfrom()* o *sendto()*. Un problema adicional se presenta cuando alguien se hace pasar por dicho proceso.

Otro problema que se presenta cuando se tiene más de un cliente es atender a todos. Existen tres estrategias a seguir para la atención de varios clientes: servidor serial, servidor padre y servidor hijo. A continuación se vera en más detalle las características de cada una de ellas.

3.2.1 Servidor serial

El servidor serial solo atiende un proceso a la vez. Una vez que termina de atender la petición solicitada verifica si tiene o no otra petición por atender. Las peticiones no atendidas son formadas en un cola de atención cuya longitud es definida por el servidor. En caso de que la cola se llene las peticiones, y lleguen más estas serán rechazadas. El algoritmo de un servidor serial es el siguiente:

```
while (true)
begin
  escuchar puerto por peticiones
  crear canal privado de comunicacion bidireccional
  leer peticion cliente
  atender peticion
  responder al cliente
  cerrar el canal de comunicacion
end
```

3.2.2 Servidor padre

El servidor padre tiene por objeto evitar que la cola se llene y que sean rechazadas las peticiones. Este servidor crea un hijo, (a través de la llamada *fork()* que atiende cada una de las peticiones. Es importante remarcar que el padre crea un canal de comunicación con el cliente y lo hereda a su hijo. Una vez heredado el canal, el padre regresa a escuchar el puerto por si llegan nuevas peticiones.

```
while (true)
begin
  escuchar puerto por peticiones
  crear canal privado de comunicacion bidireccional
  if (fork() == 0 ) then
  begin
     leer peticion cliente
     atender peticion
     responder al cliente
     cerrar el canal de comunicacion
  end
end
```

Este tipo de estrategia ejemplifica la utilidad que tiene el hecho de que los procesos vean el socket como un descriptor de archivos.

3.2.3 Servidor thread

Uno de los problemas del servidor padre es la reducción del desempeño de la máquina. Una solución de esto es utilizar *threads* en lugar de procesos para atender cada una de las peticiones. Los *threads* tienen la característica de no producir tanta carga al sistema. El principio es el mismo que el de estrategia servidor-thread, sólo que en lugar de heredar el descriptor a un hijo se le hereda a un thread.

La figura 4 presenta las tres estrategías.

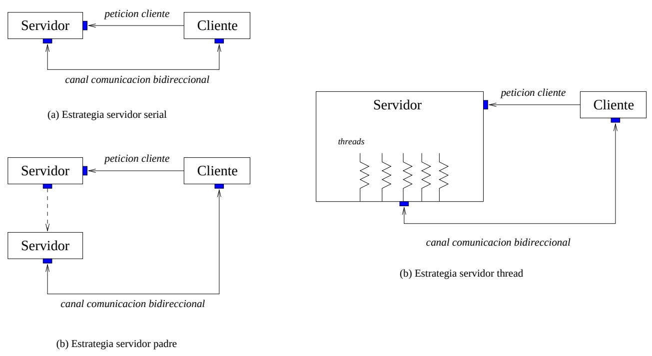

Figura 4: *Las tres posibles estrategias de un servidor*

La elección de una estrategía con respecto a las otras depende completamente de la aplicación y de los recursos con que se cuente. Vale la pena aclarar que estas estrategias son independientes del modo de conexión elegido.

3.3 Aspectos a considerar en el diseño de un cliente

A diferencia de un servidor, no existen diferentes estrategías para lo que es el diseño de un cliente. Debido a la naturaleza del modelo cliente/servidor, el cliente solo se tiene que preocupar por enviar su petición y aguardar por la respuesta.

Existen dos aspectos principales a cuidar en el diseño de un cliente: el identificar de quien es el mensaje que llega y que hacer en caso de que el servidor o la línea de comunicación no funcionen.

Para identificar al emisor se deben utilizar las mismas llamadas que se mencionaron en la sección anterior, *recvfrom()* y *sendto()*.

Las consecuencias de estar esperando una respuesta de un servidor caído es que el cliente se quede un tiempo indefinido en espera. Esto se soluciona con la implementación de un time-out. La construcción de un time-out se base en la llamada de sistema *signal()*, el enunciado *goto* del lenguaje C y la llamada de sistema *alarm()*.

El principio es el siguiente: antes de enviar la petición se debe de activar la recepción de la señal SIGALARM, la acción a realizar cuando llegue dicha señal podría ser el desplegar que se esta esperando una respuesta. El siguiente paso es enviar la petición y activar un despertador *alarm()* por el tiempo que se quiera esperar. Después de pasado ese tiempo si no ha llegado respuesta alguna, el proceso recibirá SIGALARM ejecutará el procedimiento asociado y podra abortat o reenviar el mensaje. El pseudocódigo de lo anterior se presenta a continuación:

```
   signal(SIGALARM, nada);
e: envia(peticion)
   alarm(t);
   esperando(respuesta);
   si (llega(SIGALARM)) entonces
     ejecuta(nada);
     goto e;
```

3.4 Las llamadas de sistema para la creación y uso de sockets

Independientemente de la estrategía elegida, del tipo de direcciones y de sockets utilizadas, se tienen que seguir varios pasos para la creación y uso de sockets. A continuación se presentan las llamadas de sistemas disponibles para llevar a cabo lo anterior.

3.4.1 Creación del socket: *socket()*

El socket se crea a partir de la llamada *socket()*, la sintaxis de esta llamada es la siguiente:

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

donde *domain* es la familia de direcciones con que se va trabajar, *type* es el tipo de socket y *protocol* es el protocolo a utilizar. Si *protocol* tiene un valor de cero el sistema elige el protocolo por sí mismo. La función regresa un descriprot si todo salió bine y -1 si hubo algún error.

3.4.2 Asociar el socket a un puerto *bind()*

Una vez creado el socket es necesario "asociarlo" a un puerto para que el proceso pueda ponerse a escuchar a través de él. Esta asociación se hace a partir de la llamada de sistema *bind()*:

```
int bind(int sockfd, const struct sockaddr *address, size_t address_len )
```

donde *sockfd* es el socket creado a partir de la llamada anterior, *address* contiene todos los datos de la dirección donde se va a transmitir a través del socket y *address_len* contiene el tamaño de la dirección. La función regresa 0 si todo salió bine y -1 si hubo algún error.

3.4.3 Escuchando en el puerto: *listen()*

Utilizado por el servidor en protocolos orientados conexión. Una vez que el socket fue asociado a una dirección es posible ponerse escuchar a través de él. Ya que el socket esta asociado a una dirección basta con accedar al socket para atomáticamente acceder a la dirección. La llamada de sistema para escuchar tiene la sintaxis siguiente:

```
int listen(int sockfd, int backbg)
```

donde *sockfd* es el socket a través del cual se va a escuchar, y *backbg* es el número máximo de procesos en espera, es decir la longitud de la cola de espera. Generalmente este parámetrose deja con un valor de cinco. La llamada regresa un cero si todo sale bien y -1 si hubo algún error.

3.4.4 Aceptando una llamada: *accept()*

En protocolos orientados conexión el servidor debe esperar una petición de conexión para después crear un canal de comunicación a través del cual se emiten y reciben mensajes. La llamada de sistema que crea este canal de comunicación es *accept()*. La sintaxis de la llamada es:

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

El parámetro *sockfd* representa al descriptor del socket, *cltaddr* es la dirección del proceso con el que se estableció el canal de comunicación (el cliente) y *addrlen* es la longitud de la dirección. La llamada regresa un descriptor a través del cual se van a recibir y enviar los mensajes del cliente. Este descriptor es una copia de las propiedades del descriptor *sockfd* pasado como parámetro. Si se presenta algún error regresa un valor de -1.

Es una llamada bloqueante desde el punto de vista que el proceso se bloquea hasta que llegue una demanda de conexión.

3.4.5 Conectandose al puerto: *connect()*

Las dos últimas llamadas son usadas por el servidor para escuchar demandas de conexión y crear canalaes de comuniación. Por el otro lado el cliente cuenta con la llamada *connect()* para solicitar una conexión. Esta llamada presenta la sintaxis siguiente:

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

donde *sockfd* es el descriptor de socket creado a partir de la llamada *socket()*, *servaddr* es la dirección del servidor y *addrlen* es el tamaño de la dirección.

La llamada regresa 0 si todo salió bien y -1 si no se pudo establecer ninguna conexión.

3.4.6 Recepción de mensajes: *read(), recv()* y *recvfrom()*

Existen tres llamadas que se pueden utilizar para la recepción de mensajes, y presentan la siguiente sintaxis:

```
int read( int fildes, void *buf, size_t nbyte);
int recv((int sockfd, const void *buffer, size_t  length, int flags}
ssize_t recvfrom(int sockfd, void *buffer, size_t length, int flags,
                 struct sockaddr *address, size_t *address_len);
```

La primera es la utilizada para la lectura de archivos, se lee de *fildes*, *nbytes* bytes de informaciíon y la lectura se alamcena en *buf*.

La llamada *recv()* es muy parecida a la anterior, lee un mensaje de longitud *length* del socket *sockfd*, dejando el contenido en la variable *buffer*; el último parámetro es utilizado para protocolos especiales, pero para nuestras aplicaciones se le asigna un valor de 0.

La última llamada, *recvfrom()* espera por un mensaje de *sockfd* de tamaño *length* dejando la lectura en *buffer*; en la variable *address* se tiene la información del emisor y en *address_len* la longitud de esa dirección. El parámetro *flags* es para protocolos especiales, aunque para las aplicaciones de este capítulo se le asignará el valor de cero. Si también los parámetros *address_len* y *address* tienen un valor de cero la llamada se comporta como una llamada *read()*

Todas las llamadas regresan el número de bytes recibidos si todo salió bien, o -1 si hubo algún error. La elección entre una y otra depende del tipo de protocolo que se este usando: conexión o no conexion. Las dos primeras son utilizadas en protocolos tipo conexión mientras que la última se utiliza en protocolos orientado no conexión.

3.4.7 Enviando mensajes: *write(), send()* y *sendto()*

Se cuenta con tres llamadas para el envío de mensajes, a continuación se presentan las sintaxis de estas llamadas:

```
int write( int fildes, void *buf, size_t nbyte);
int send((int sockfd, const void *buffer, size_t  length, int flags}
ssize_t sendto(int sockfd, const void *message, size_t length, int flags,
               const struct sockaddr *dest_addr, size_t dest_len);
```

La primera es la utilizada para la escritura de archivos, se lee de *fildes*, *nbytes* bytes de informaciíon y la lectura se alamcena en *buf*.

La llamada *send()* es muy parecida a la anterior, envia un mensaje de longitud *length* a través del socket *sockfd*. Se asume que el mensaje esta almacenado en la variable *buffer*; el último parámetro es utilizado para protocolos especiales, pero para nuestras aplicaciones se le asigna un valor de 0.

La última llamada, *sendto()*, envía el mensaje *buffer* de tamaño *length* a través de *sockfd*. En la variable *address* se guarda la información del emisor y en *address_len* la longitud de esa dirección. El parámetro *flags* es para protocolos especiales, aunque para las aplicaciones de este capítulo se le asignará el valor de cero. Si también se asigna un valor de cero a los parámetros *address_len* y *address* la llamada se comporta como una llamada *write()*

Todas las llamadas regresan el número de bytes enviados si todo salió bien, o -1 si hubo algún error. Las dos primeras se utilizan para en el caso de protocolos orientados conexión mientras que la última se utiliza en protocolos orientados no conexión.

3.4.8 Cerrando la comunicación: *close()*

El socket es un descriptor por lo que para cerrarlo se debe de utilizar la llamada de sistema *close()*. La sintáxis de dicha llamada es:

```
int close(int fildes);
```

donde *fildes* es el descriptor del socket.

Si no se cierra el socket el puerto utilizado para la comunicación no podrá ser utilizado de nuevo por ningún otro proceso. Incluso si por alguna razón se tiene que interrumpir el programa es conveniente cerrar el socket antes de abortar, ya que de lo contrario no se podra reiniciar el programa, sobretodo el servidor.

4 Comunicación orientada conexión

En este modo de comunicación es necesario que el cliente le solicite al servidor una conexión. Una vez establecida dicha conexión el cliente y el servidor pueden empezar a intercambiar mensajes. Generalmente es el cliente el que emite el primer mensaje pero esto hecho no se puede generalizar.

En la figura 5 se presenta un esquema general de los pasos requeridos para poder comunicar dos procesos en modo conexión.

4.1 El servidor

El servidor debe de crear un socket, y asociarlo a un puerto. Una vez asociado el socket al puerto el servidor debe de esperar por peticiones de conexión al puerto asociado. Cuando la petición llegue se debe de crear un canal de comunicación que sirva de puente de comunicación. Una vez establecido el canal, se puede empezar a enviar y a recibir mensajes. Al final el servidor debe de cerrar el socket para que el puerto pueda ser utilizado por otros procesos. Mientras el servidor este usando el puerto ningún otro proceso podra hacer uso de él.

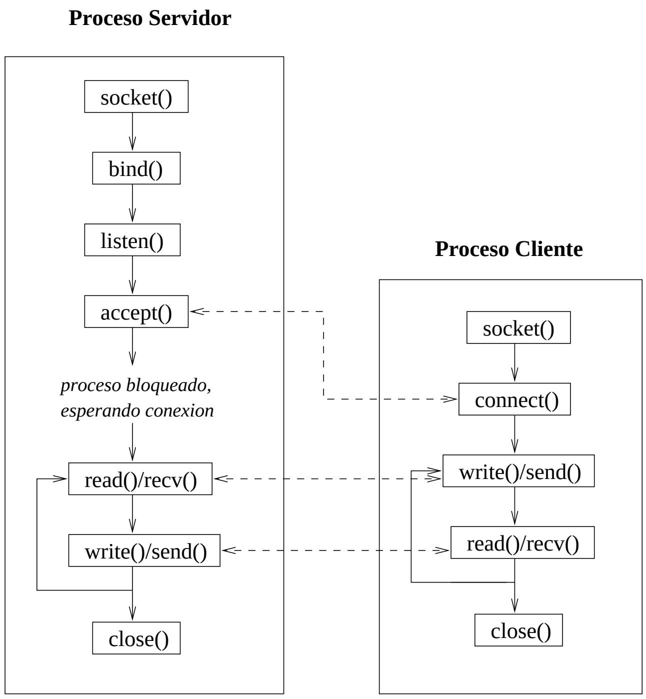

Figura 5: *Pasos a seguir para establecer una comunicación en modo conexión*

Cabe la pena aclarar que estos pasos son independientes de la estrategia de atención del servidor. Todos los pasos se deben de realizarse en el orden establecido, las variantes necesarias para implementar una determinada estrategia harán uso de otras llamadas.

1. Creación del socket
2. Asociando el socket al puerto
3. Escuchando en el puerto
4. Aceptando una petición
5. Recibiendo la petición
6. Atendiendo la petición
7. Enviando la respuesta
8. Cerrando la conexión

La atención de la petición depende del servicio solicitado. En la mayor parte de los casos, se trata de una simple llamada a una función que implementa el servicio.

Es lógico que si el cliente esta intentanto conectarse con el servidor en un determinado puerto y el servidor esta escuchando otro puerto, la conexión no podra llevarse a cabo.

4.2 El cliente

Un proceso debe de crear un socket, asociarlo a un número de puerto, (el cual varía de acuerdo a la aplicación, solicitar una conexión al servidor. Una vez establecida la conexión el cliente puede empezar a emitir/recibir mensajes. Por último debe de cerrar el socket, lo que provoca el cierre del canal de comunicación. A continuación se describen los llamadas de sistema necesarias para llevar a cabo lo anterior, puede compararse esta lista con lo descritos en la figura 2 y 5:

1. Creación del socket
2. Asociación del socket a un puerto
3. Conectarse al servidor
4. Envío de la petición
5. Recepción de la respuesta
6. Cerrando la comuniación

5 Comunicación orientada no conexión

A diferencia de la comunicación orientada conexión la comunicación orientada no-conexión no necesita establecer un canal de comunicación entre las dos entidades comunicantes. Tan sólo es necesario que el cliente envie su petición, que el servidor la reciba, la atienda y en caso necesario, envie la respuesta.

Vale la pena aclarar que en esta comunicación los mensajes contienen la dirección del destinatario, y pueden perderse, lo cual lo hace poco confiable.

En la figura 6 se presenta un esquema general de los pasos necesarios para poder establecer una comunicación orientada no-conexión entre el cliente y servidor:

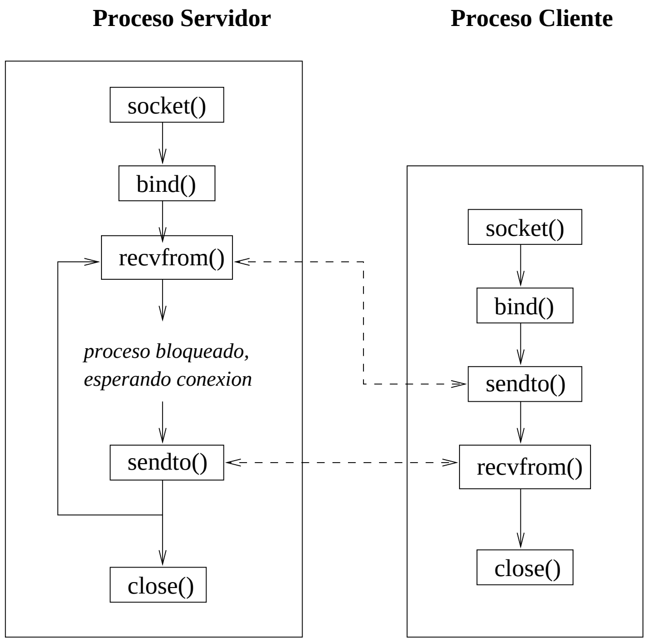

Figura 6: *Pasos a seguir para establecer una comunicación en modo no conexión*

5.1 El servidor

Los pasos a seguir por un servidor para atender las peticiones de los clientes difieren de aquellos utilizados en modo conexión. No es necesario que el servidor espere solicitudes de conexión y cree un canal de comunicación para poder empezar a recibir y enviar mensajes. El servidor debe de crear un socket asociarlo a un puerto y ponerse a esperar que las peticiones lleguen para después antenderlas y enviar la respuesta si esta es necesaria.

Por el lado del servidor este debe de realizar los siguientes pasos:

1. Creación de un socket
2. Asociar el socket a un puerto
3. Esperar por un mensaje
4. Atención de la petición
5. Envío de la respuesta
6. Cerrar la comunicación

Como se puede apreciar el código del servidor es más reducido en el caso del modo no-conexión que el del modo conexión.

5.2 El cliente

Los pasos que un cliente debe de realizar para poder establecer una comunicación con el servidori en un modo no-conexión no difieren mucho de los realizados en modo conexi¯on. La principal diferencia radica en que no es necesario enviar una petición de conexión. En efecto después de crear el socket y asociarlo a un puerto, el cliente envia directamente al cliente su mensaje. Dependiendo de la aplicación puede esperar por una respuesta o ponerse a hacer otra cosa. Las llamadas de sistema necesarias para que el cliente se comunique con el servidor son las siguientes:

1. Crear un socket
2. Asociar el socket a un número de puerto
3. Enviar la respuesta
4. Esperar la respuesta
5. Cerrar la comunicación

Siguiendo los pasos anteriores el código del cliente no puede asegurar que el mensaje llegue al servidor, ni detectar que el servidor esta caído o que la vía de comunicación se rompió. Con el fin de evitar esperas infinitas por parte del cliente se sugiere implementar un *time-out* para volver enviar el mensaje o darse por vencido e intentarlo después si así se desea.

6 Ejemplo de uso de sockets de familia Unix en modo conexión

El ejemplo consiste en averiguar el identificador del servidor o el identificador de su padre. Las opciones entre una y otra dependen de la opción de la línea de comandos. Se utilizaron direcciones de familia Unix y sockets de tipo stream, el archivo que sirve de enlace a los dos procesos se llama *socket* y es creado en el mismo directorio en donde se ejecute el programa.

A continuación se muestran los codigos del servidor y del cliente.

6.1 El código del servidor

El servidor escucha por peticiones, recibe la petición, verifica de que petición sse trata, la atiende y envia la respuesta. Puede recibir dos tipos de mensajes de petición:

id: se desea conocer el identificador del servidor *padre*: se desea conocer el identificador del padre del servidor

El servicio esta definido en la función *atiende()*. En esta ocasión se decidió dejar escribir la función de atención en el mismo archivo que el código del servidor, pero bien podría estar definida en otro archivo.

El servidor esta diseñado de tal forma que solo atiende a un cliente a la vez, si se desea que el servidor atienda a otras peticiones se tiene que volver a ejecutar el servidor.

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

#define MSGSIZE 1024     /* longitud del mensaje */

char *atiende();         /* rutina de atencion */
char  pet[MSGSIZE];  /* mensaje de peticion */
char  *resp;             /* la respuesta */

main()
{
   int   sd, sd_actual;        /* descriptores de sockets */
   struct sockaddr_un  sock;   /*  direccion socket */

   /* asignacion valores a la direccion */
   strcpy(sock.sun_path, "./socket");
   sock.sun_family = AF_UNIX;

   /* obtencion de un socket tipo AF_UNIX   */
   if ( (sd = socket(AF_UNIX, SOCK_STREAM, 0)) == -1) {
     perror("socket");
     exit(1);
   }

   /* asociando el socket al archivo ./socket */
   if (bind(sd, (struct sockaddr *) &sock, sizeof(sock)) == -1) {
     perror("bind");
     exit(1);
   }

   /* ponerse a escuchar a traves del socket */
   if (listen(sd, 5) == -1) {
     perror("listen");
     exit(1);
   }

   /* esperando que un cliente solicite un servicio */
   if ( (sd_actual = accept(sd, 0, 0)) == -1) {
     perror("accept");
     exit(1);
   }

   /* tomar un mensaje del cliente */
   if (recv(sd_actual, pet, MSGSIZE, 0) == -1) {
     perror("recv");
     exit(1);
   }
```

```
   /* atendiendo la peticion */
   resp = atiende(pet);

   /* enviando la respuesta del servicio */
   if (send(sd_actual, resp, MSGSIZE, 0) == -1) {
     perror("send");
     exit(1);
   }

   /* cerrar los dos sockets */
   close(sd_actual);   close(sd);

   /* borrando archivo conexion, para poder usarlo despues */
   unlink("./socket");
}

/* rutina atencion */

char *atiende(peticion)
  char *peticion;
{
  static char ident[20];
  int id;

  if ( strcmp(peticion,"id") == 0)    /* obtencion identificador servidor */
    id = getpid();
  else
    id = getppid();                   /* obtencion idnetificador padre servidor */

  sprintf(ident,"%d",id);
  return(ident);
}
```

6.2 El código del cliente

El cliente envia lee la opción de la línea de comandos y de acuerdo a dicha opción envía el mensaje de petición. Si la opción fue `-i` envía el mensaje *id* solicitando el identificador del proceso, y si la opción fue `-p` envía el mensaje *padre* solicitando el identificador del padre del servidor. Espera la respuesta y al final imprime el resultado recibido.

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

#define  MSGSIZE   1024     /* longitud maxima mensaje */

char pet[MSGSIZE];          /* mensaje de peticion */
char resp[MSGSIZE];         /* mensaje respuesta */


main(argc, argv)
  int argc;
  char *argv[];
{
```

```
   int     sd;                  /* descriptor socket */
   struct sockaddr_un   sock;  /*  direcciones socket */

   /* verificacion numero parametros */
   if (argc != 2) {
      fprintf(stderr,"Error, uso: %s opcion \n",argv[0]);
      exit(1);
   }

   /* asignacion valores a la direccion */
   strcpy(sock.sun_path, "./socket");
   sock.sun_family = AF_UNIX;

   /* obtencion de un socket tipo AF_UNIX  (igual que el servidor) */
   if ( (sd = socket(AF_UNIX, SOCK_STREAM, 0)) == -1) {
      perror("socket");
      exit(1);
   }

   /* conectandose  al archivo ./socket */
   if (connect(sd, (struct sockaddr *)&sock, sizeof(sock)) == -1) {
      perror("connect");
      exit(1);
   }

   /* construccion de la peticion */
   if ( strcmp(argv[1],"-i") == 0)
      strcpy(pet,"id");
   else
      if ( strcmp(argv[1],"-p") == 0)
         strcpy(pet,"padre" );
      else {
         fprintf(stderr,"Opcion %s desconocida \n",argv[1]);
         exit(1);
      }

   /* enviar mensaje al servidor */
   if (send(sd, pet, MSGSIZE, 0) == -1) {
     perror("send");
     exit(1);
   }

   /* esperar por la respuesta */
   if (recv(sd, resp, MSGSIZE, 0) == -1) {
     perror("recv");
     exit(1);
   }

   /* imprimir los resultados */
   if (strcmp(argv[1],"-i") == 0)
     printf("El identificador del servidor es: %s \n", resp);
   else
     printf("El identificador del padre es: %s \n",resp);

   /* cerrando la comunicacion */
   close(sd);
}
```

7 Ejemplo de uso de sockets tipo Internet en modo conexión

A diferencia del ejemplo anterior en este ejemplo un proceso puede preguntarle a otro la hora, su identificador o el identificador de su padre. La opción entre un servicio u otro se da a nivel de la línea de comandos. Se utilizan direcciones de tipo Internet, con sockets de tipo stream, el protocolo es por lo tanto TCP y lo elige el sistema.

7.1 Código del servidor

El servidor debe estar ejecutando en la máquina brasil y debe estar escuchando en el puerto 34567. Aparte de los dos mensajes de petición del ejemplo anterior (*id* y *padre*) se cuenta con el mensaje *hora* para solicitar la hora.

Este también es un ejemplo de implementación de una estrategia servidor-serial, el servidor cicla en infinito esperando por peticiones. Es necesario abortar el programa para poder terminar con la ejecución del servidor.

El código presenta una opción que permite abortar el programa tecleando `<ctrl-c>` . En dicho caso el programa despliega un mensaje y cierra los sockets para que puedan ser usados después.

La función que realiza el servicio también se llama *atiende()*, y difiere muy poco de la del ejemplo anterior.

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <time.h>
#define  MSGSIZE  1024      /* longitud maxima del puerto*/
#define  PUERTO   34567    /* numero puerto arbitrario */

char *atiende();           /* rutina de atencion   */
void cierralos();          /* funcion de abotar programa */
char pet[MSGSIZE];         /* mensaje de peticion  */
char *resp;                /* mensaje de respuesta */
int  sd, sd_actual;    /* descriptores de sockets */

main()
{
    int    cc, fromlen, tolen, addrlen;    /* varios */
    struct sockaddr_in   sin, pin;  /*  direcciones socket */

/* activando funcion de abortar */
    signal(SIGINT, cierralos);

/* obtencion de un socket tipo internet */
    if ( (sd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
      perror("socket");
      exit(1);
    }

/* asignar direcciones en la estructura de direcciones */
    sin.sin_family = AF_INET;
```

```
    sin.sin_addr.s_addr = INADDR_ANY;
    sin.sin_port = htons(PUERTO);        /* convirtiendo a formato red*/

/* asociando del socket al numero de puerto */
    if ( bind(sd, (struct sockaddr *)&sin, sizeof(sin)) == -1) {
      perror("bind");
      exit(1);
    }

    /* ponerse a escuchar a traves del socket */
   if (listen(sd, 5) == -1) {
      perror("listen");
      exit(1);
   }


   while (1) {     /* ciclo infinito para recepcion de peticiones */

      /* esperando que un cliente solicite un servicio */
      if ( (sd_actual = accept(sd, (struct sockaddr *)&pin, &addrlen)) == -1) {
        perror("accept");
        exit(1);
      }

      /* tomar un mensaje del cliente */
      if (read(sd_actual, pet, MSGSIZE) == -1) {
        perror("read");
        exit(1);
      }

      /* atendiendo la peticion */
      resp = atiende(pet);

      /* enviando la respuesta del servicio */
      if (write(sd_actual, resp, MSGSIZE) == -1) {
        perror("write");
        exit(1);
      }
   }

   /* cerrar los dos sockets */
   close(sd_actual);   close(sd);
}

char *atiende(peticion)
 char *peticion;
{
  static char ident[20];
  int id;

  id =-5;
  if ( strcmp(peticion,"id") == 0)
    id = getpid();
  if ( strcmp(peticion,"padre") == 0)
    id = getppid();
  if ( strcmp(peticion,"hora") == 0)
    id = time(NULL);

  sprintf(ident,"%d",id);
```

```
  return(ident);
}

void cierralos()
{
  printf("\nAbortando servidor: cerrando descriptores ... \n");
  close(sd_actual);
  close(sd);
  exit(1);
}
```

7.2 Código del cliente

El cliente sabe que la máquina que esta prestando el servicio se llama brasil y que el puerto es el 34567. Se manejan las mismas opciones que el ejemplo anterior solo que se añade la opción `-h` para emitir el mensaje *hora* y solicitar la hora al proceso remoto.

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


#define  MSGSIZE   1024         /* longitud mensaje peticion/respuesta */
#define  HOST    "brasil" /* nombre host servidor */
#define  PUERTO    34567 /* numero puerto arbitrario */

char pet[MSGSIZE];         /* mensaje de peticion */
char resp[MSGSIZE];        /* mensaje de respuesta */

main(argc, argv)
   int argc;
   char *argv[];
{
   int     sd;       /* descriptores de sockets */
   struct hostent  *hp;    /* estructura del host */
   struct sockaddr_in   sin, pin;  /* direcciones socket */
   struct tm *hora;               /* estructura del tiempo */
   time_t hora_servidor;          /* la hora del servidor */

/* verificando parametros de entrada */
   if ( argc != 2) {
     fprintf(stderr,"Error, uso: %s -iph \n",argv[0]);
     exit(1);
   }

/* encontrando todo lo referente acerca de la maquina host */
   if ( (hp = gethostbyname(HOST)) == 0) {
      perror("gethosbyname");
      exit(1);
   }

/* llenar la estructura de direcciones con la informacion del host */
```

```
   pin.sin_family = AF_INET;
   pin.sin_addr.s_addr = ((struct in_addr *) (hp->h_addr))->s_addr;
   pin.sin_port = htons(PUERTO);

/* obtencion de un socket tipo internet */
   if ( (sd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
      perror("socket");
      exit(1);
   }

/* conectandose al PUERTO en el HOST  */
   if (connect(sd, (struct sockaddr*)&pin, sizeof(pin)) == -1) {
     perror("connect");
     exit(1);
   }

/* construccion de la peticion */
   if ( strcmp(argv[1],"-i") == 0)
     strcpy(pet,"id");
   else
     if ( strcmp(argv[1],"-p") == 0)
      strcpy(pet,"padre" );
     else
       if ( strcmp(argv[1],"-h") == 0)
         strcpy(pet,"hora");
       else {
         fprintf(stderr,"Opcion %s desconocida, (-iph) \n",argv[1]);
         exit(1);
       }

/* enviar mensaje al PUERTO del  servidor en la maquina HOST */
   if ( write(sd, pet, MSGSIZE, 0) == -1) {
     perror("write");
     exit(1);
   }

/* esperar por la respuesta */
   if ( read(sd, resp, MSGSIZE,0) == -1) {
     perror("read");
     exit(1);
   }

/* imprimir los resultados */
   if (strcmp(argv[1],"-i") == 0)
     printf("El identificador del servidor es: %s \n", resp);
   if (strcmp(argv[1],"-p") == 0)
     printf("El identificador del padre del servidor es: %s \n",resp);
   if (strcmp(argv[1],"-h") == 0) {
     printf("La hora del servidor es: ");
     hora_servidor = (time_t)atoi(resp);
     hora = (struct tm *)localtime(&hora_servidor);
     printf("%d horas con %d minutos y %d segundos \n",
               hora->tm_hour, hora->tm_min, hora->tm_sec);
   }

/* cerrando la comunicacion */
   close(sd);
}
```

8 Ejemplo de uso de sockets tipo Internet en modo no conexión

En este ejemplo, al igual que el anterior, un proceso puede preguntarle a otro la hora, su identificador o el identificador de su padre. La opción entre un servicio u otro se da a nivel de la línea de comandos. Se utilizan direcciones de tipo Internet, con sockets de tipo datagrama, el protocolo es por lo tanto UDP y lo elige el sistema.

Otra diferencia con el ejemplo anterior es el hecho de que el servidor y el cliente eligen casi todos los parámetros a partir de la línea de comando (puerto a escuhar, nombre de la máquina, opción y otras más).

8.1 Código del servidor

El servidor puede estar ejecutando en cualquier máquina y a través de la línea de comandos se le pasa el puerto por el cual le van a llegar las peticiones y por el cual va a enviar su respuesta.

En este caso el servidor sigue una estrategia servidor-padre, creando un hijo que atienda cada petición que llega. El proceso ejecuta un ciclo infinito esperando por las peticiones, para terminar la ejecución del servidor es necesario abortarlo tecleando `<ctrl-c>` . Al abortar, igual que en el ejemplo anterior, los sockets son cerrados.

No se incluye el código de la función *cierralos()*, ni el código de la función *atiende()*, ya que es el mismo que el del ejemplo anterior.

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

#define  MSGSIZE  1024        /* longitud maxima mensaje      */

char    *atiende();           /* rutina de atencion           */
void    aborta();             /* funcion de abotar programa   */
char    pet[MSGSIZE];         /* mensaje de peticion          */
char    *resp;                /* mensaje de respuesta         */
int    sd;       /* descriptores de sockets      */

main(argc, argv)
  int argc;
  char *argv[];
{

  int      cc, addrlen;    /* varios: recepcion y longitud msj  */
  struct sockaddr_in   myaddr; /* mi direccion internet             */
  struct sockaddr_in   claddr;  /* direccion internet del cliente    */
  struct hostent *hclient; /* datos host cliente              */
  char cliente[100];                   /* nombre cliente                    */

   /* activando funcion de abortar */
```

```
  signal(SIGINT, aborta);

/* verificar los parametros */
if (argc != 2)  {
   fprintf(stderr,"Error, uso: %s <puerto> \n",argv[0]);
   exit(1);
}

/* obtencion de un socket tipo internet */
if ( (sd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
  perror("socket");
  exit(1);
}

/* anadiendo informacion sobre el cliente */
myaddr.sin_family = AF_INET;
myaddr.sin_addr.s_addr = INADDR_ANY;
myaddr.sin_port = atoi(argv[1]);

/* asociando el socket al puerto  */
if (bind(sd, (struct sockaddr*)&myaddr, sizeof( struct sockaddr_in)) == - 1) {
  perror("bind");
  exit(1);
}

while (1) {       /* ciclo infinito para recepcion de peticiones */

  printf("Proceso servidor %d esperando peticion \n",getpid());
  /* tomar el mensaje cuando este es enviado */
  cc = recvfrom(sd, pet, MSGSIZE, 0, (struct sockaddr *)&claddr, &addrlen);
  if ( cc == -1) {
    perror("recvfrom");
    exit(1);
  }
  /* Obtencion nombre cliente */
   hclient = gethostbyaddr((char *)&(claddr.sin_addr.s_addr), 4, AF_INET);
   if (hclient== NULL)
     strcpy(cliente,"desconocido");
   else
     strcpy(cliente, (*hclient).h_name);

  /* Se crea un hijo para que atienda la peticion */
  if (fork() == 0) {
     printf("\tProceso hijo %d atendiendo la peticion de %s\n",getpid(), cliente);
     resp = atiende(pet);             /* realizar el servicio  */

     /* enviar la respuesta al cliente */
     addrlen = sizeof(struct sockaddr_in);
     printf("Respondiendo: %s \n",resp);
     if (sendto(sd, resp, MSGSIZE, 0,  (struct sockaddr*)&claddr, addrlen) == -1) {
        perror("sendto");
        exit(1);
     }
     exit(1);
  }
}

/* Es conecctionless - no es necesario el accept() no hay socket extra que cerrar */
  close(sd);
```

```
}
```

8.2 Código del cliente

El cliente recibe como parámetros de la línea de comandos el nombre de la máquina donde se encuentra ejecutando el servidor, el número de puerto a través del cual se va a enviar las peticiones y se van a recibir las respuestas, y también las ocpiones de servicio, (identificador servidor `-i` , padre servidor `-p` y petición de hora `-h`).

El cliente cuenta con una opción de time-out. Si después de cinco segundos de espera no se obtiene ninguna respuesta se envia de nuevo la petición. Si al cabo de cinco intentos no llega ninguna respuesta el programa aborta.

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

#define  MSGSIZE  1024   /* longitud maxima del mensaje */

char pet[MSGSIZE];       /* mensaje de peticion */
char resp[MSGSIZE];      /* mensaje de respuesta */
int  segs;               /* contador de intentos */

void  despliega();

main(argc, argv)
   int argc;
   char *argv[];
{

   int                 sd;              /* descriptor del socket */
   int                 retry;         /* contador intentos de envio peticion */
   int                 addrlen;   /* varios */
   char                *host;           /* nombre del host */
   struct hostent      *hp;             /* out port descriptor */
   struct servent      *sp;         /* el portmap para el servicio */
   struct sockaddr_in   myaddr;   /* mi direccion internet (cliente) */
   struct sockaddr_in   svaddr;  /* direccion internet (servidor) */
   struct sockaddr_in   claddr;    /* direccion internet emisor respuesta */
   struct tm *hora;                /* estructura del tiempo */
   time_t hora_servidor;           /* la hora del servidor */



/* verificar los parametros */

   if (argc != 4)  {
      fprintf(stderr,"Error, uso: %s <host> <puerto> -iph \n",argv[0]);
      exit(1);
   }
```

```
/* llenar la estructura de direcciones con la informacion del host */

   host =argv[1];
   hp = gethostbyname(host);
   if (hp == NULL) {
     fprintf(stderr,"No se encontro %s en /etc/hosts \n",host);
     exit(1);
   }
   svaddr.sin_addr.s_addr = ((struct in_addr *)(hp->h_addr))->s_addr;
   svaddr.sin_port = atoi(argv[2]);
   svaddr.sin_family = AF_INET;

/* obtencion de un socket tipo internet */

   if ( (sd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
     perror("socket");
     exit(1);
   }

/* anadiendo informaci{on sobre el cliente */

   myaddr.sin_family = AF_INET;
   myaddr.sin_port = 0;
   myaddr.sin_addr.s_addr = INADDR_ANY;

/* asociado el socket al puerto  */
 if (bind(sd, (struct sockaddr *)&myaddr, sizeof( struct sockaddr_in)) == - 1) {
     perror("bind");
     exit(1);
   }
/* construccion de la peticion */

   if ( strcmp(argv[3],"-i") == 0)
     strcpy(pet,"id");
   else
     if ( strcmp(argv[3],"-p") == 0)
      strcpy(pet,"padre" );
     else
       if ( strcmp(argv[3],"-h") == 0)
         strcpy(pet,"hora");
       else {
         fprintf(stderr,"Opcion %s desconocida (-iph) \n",argv[3]);
         exit(1);
       }

/* intenta enviar un mensaje al servidor */
/* si despues de 5 segundos no responde le vuelbe a enviar un mensaje */
/* despues del quinto intento aborta el programa */

       signal(SIGALRM, despliega);
       segs=0;
       retry = 5;
       addrlen = sizeof(struct sockaddr_in);
again: if ( sendto(sd, pet, MSGSIZE, 0, (struct sockaddr *)&svaddr, addrlen) == -1) {
         perror("sendto");
         exit(1);
       }
       alarm(5); /* signal SIGALRM es repetida cada 5 segundos */
```

```
        if ( recvfrom(sd, resp, MSGSIZE, 0, (struct sockaddr *)&claddr, &addrlen) == -1)
        {
           if (errno == EINTR) {
             if (--retry) {
               goto again;
             }
             else {
               fprintf(stderr,"Abortando programa: servidor no respondio despues 5 intentos \n");
               exit(1);
             }
           }
           else {
             perror("recvfrom");
             exit(1);
           }
        }
        alarm(0);         /*resetear la alarma */
/* en caso de exito: se imprimen resultados */


   if (strcmp(argv[3],"-i") == 0)
     printf("El identificador del servidor es: %s \n", resp);
   if (strcmp(argv[3],"-p") == 0)
     printf("El identificador del padre del servidor es: %s \n",resp);
   if (strcmp(argv[3],"-h") == 0) {
     printf("La hora del servidor es: ");
     hora_servidor = (time_t)atoi(resp);
     hora = (struct tm *)localtime(&hora_servidor);
     printf("%d horas con %d minutos y %d segundos \n",
               hora->tm_hour, hora->tm_min, hora->tm_sec);
   }

/* cerrando la comunicacion */

   close(sd);

}

void  despliega()
{
   segs+=5;
   fprintf(stderr,"Servidor no responde despues de %d sgs, intentado de nuevo ...\n",segs);
   signal(SIGALRM, despliega);
}
```

9 Conclusiones

Los sockets es una herramienta que permite que dos procesos se comuniquen aún si los dos se encuentran en máquinas diferemtes. Su uso es sencillo aunque requiere de cierto conocimiento de base del protocolo TCP/IP.

Uno de los problemas que presenta es el de depuración, ya que no es sencillo depurar un programa con sockets, sin embargo si se pone atención a la sintaxis de las llamadas de sistema no se tiene que presentar problema alguno.

Referencias

[Ste97] Advanced Programming in the Unix Environment, W. Richard Stevens, Ed. Addison Wesley

[Dei90] Como programar en C/C++, H.M. Deitel / P.J. Deitel Ed. Prentice Hall

[Blo91] Power Programming with RPC, John Bloomer, O'Reilly & Associates, Inc.

[Bro92] Unix Distributed Programming, Chris Brown, Ed. Prentice Hall

[Ri92] Comunicaciones en Unix, Jean Michel Rifflet; Ed. Mc. Graw Hill

[St90] Unix Networking Programming, Stevens, Ed. Prentice Hall

[St95] TCP/IP Illustrated, Vol. I The Protocols, W. Richard Stevens, Ed. Addison Wesley

[WrSt95] TCP/IP Illustrated, Vol. II The Implementation, Gary R. Wright, W. Richard Stevens, Ed. Addison Wesley

[Sa94] TCP/IP and ONC/NFS Internetworking in a Unix Environment, Michael Santifaller, Ed. Addison Wesley

[Li94] Expert C Programming Deep C Secrets, Peter Van Der Linden Ed. Prentice Hall

[Co91] Internetworking with TCP/IP, Vol I, Principles, Protocols and Architecture, Douglas E. Comer, Ed. Prentice Hall

[Ga94] Unix Programación Avanzada, Fco. Manuel Garcíaz. Addison Wesley Iberoamericana