Interfaz de Sockets

El patrón de interacción primario que se da entre las aplicaciones de cooperación se conoce como *cliente/servidor*. La interacción cliente/servidor forma la base de la mayor parte de la comunicación por red [1]. Para construir eficientemente aplicaciones de red, es necesario conocer a fondo la interfaz de sockets y sus APIs (Application Programming Interface).

Es deseable poseer un dominio de las funciones de sockets elementales para escribir un programa cliente/servidor completo. La manera más sencilla de aprender estas funciones es mediante el uso de ejemplos básicos de programas para varios ambientes (TCP, UDP) [5].

En este documento se muestra una sencilla introducción al paradigma cliente/servidor para luego profundizar en la descripción de las funcionalidades y servicios básicos del API de sockets. Las primitivas y ejemplos mostrados siguen el API de los sistemas Unix. Se asume un conocimiento básico previo de la pila de protocolos TCP/IP.

1. Conceptos básicos

En esta sección se presentan algunos conceptos básicos que facilitarán el entendimiento de los ejemplos planteados en secciones posteriores.

1.1 El modelo Cliente/Servidor

El término *servidor* se aplica a cualquier programa que ofrece un servicio que se puede obtener en una red. Un servidor acepta la petición desde la red, realiza el servicio y devuelve el resultado al solicitante. Un programa ejecutable se convierte en *cliente* cuando envía una petición a un servidor y espera una respuesta [1].

La gran mayoría de las aplicaciones de red están escritas asumiendo que un extremo es el cliente y el otro el servidor. El propósito de la aplicación es que el servidor provea algunos servicios predefinidos a los clientes [3].

1.2 Números de puerto

TCP y UDP identifican a las aplicaciones usando números de puerto de 16 bits. Es necesario determinar cómo se hará la escogencia de esos números. Los servidores son normalmente conocidos por un número de puerto *bien conocido*. Por ejemplo, cada implementación de TCP/IP que provee un servidor FTP ofrece sus servicios en el puerto TCP 21, cada servidor Telnet está en el puerto TCP 23, etc. Estos servicios que pueden ser provistos por cualquier implementación de TCP/IP tienen un número de puerto bien conocido entre el 1 y el 1023. Los puertos bien conocidos son manejados por la IANA (Internet Assigned Numbers Authority) [3].

A un cliente normalmente no le importa cuál número de puerto usa en su extremo; lo único que necesita saber con certeza es que el número que elija sea único en su host. Los números de puerto de los clientes son llamados *efímeros* [3].

1.3 Sockets

Un socket es una abstracción a través de la cual una aplicación puede enviar y recibir datos hacia y desde otras aplicaciones que estén conectadas a la misma red [2].

Los sockets pueden ser de diferentes tipos, de acuerdo a las diferentes familias de protocolos sobre las que se sustente la comunicación. Un socket que usa la familia de protocolos TCP/IP es identificado de forma unívoca mediante una dirección IP, un protocolo *end-to-end* (TCP o UDP) y un número de puerto [2].

1.3.1 Creación de un socket

Cuando una aplicación quiere comunicarse con otra, el programa solicita al sistema operativo crear una instancia de la abstracción socket. La función que permite esto es `socket()`; sus parámetros especifican la clase de socket solicitado por el programa.

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

El primer parámetro define la familia de protocolos del socket. La constante `PF_INET` especifica a un socket que usa la familia de protocolos de Internet (IPv4). El segundo parámetro especifica el tipo de socket; el tipo determina la semántica de la transmisión de datos del socket (por ejemplo, si es confiable). La constante `SOCK_STREAM` especifica a un socket con semántica de flujo de bytes confiable, mientras que la constante `SOCK_DGRAM` especifica a un socket basado en servicio de datagramas de mejor esfuerzo [2].

El tercer parámetro especifica el protocolo end-to-end a utilizar. Para la familia `PF_INET`, es posible usar `IPPROTO_TCP` para indicar el uso de TCP, o `IPPROTO_UDP` para usar UDP.

El valor de retorno de la función `socket()` es un entero. Un valor -1 indica que ocurrió un fallo, mientras que un valor no negativo indica que la operación fue exitosa. En este último caso, el número devuelto es denominado descriptor del socket [2].

1.3.2 Destrucción de un socket

Cuando una aplicación termina de utilizar el socket llama a la función `close()`, pasando como parámetro el identificador del socket que desea cerrar.

```
int close(int socket)
```

La función `close()` le indica a la pila de protocolos que soporta al socket que inicie las acciones requeridas para cerrar la comunicación y liberar los recursos asociados al socket en cuestión. El

valor de retorno es 0 en caso de éxito, o -1 en caso de fallo. Una vez que se ha invocado la función `close()` no es posible enviar o recibir más datos a través de ese socket [2].

1.3.3 Estructura de direcciones de sockets

La mayoría de las funciones de sockets requieren un puntero a una estructura de dirección de sockets como argumento. Cada suite de protocolos define su propia estructura de direcciones. Los nombres de estas estructuras comienzan por `sockaddr_`, y terminan con un sufijo único para cada suite de protocolos [5].

1.3.4 La estructura de direcciones para IPv4

Una estructura de direcciones IPv4 para sockets es llamada `sockaddr_in`, y está definida en la cabecera `<netinet/in.h>`. La estructura se muestra a continuación en el Ejemplo 5.1 [5]:

```
struct in_addr {
    in_addr_t  s_addr;          /*Direccion IPv4 de 32 bits*/
                                /*En orden de red, big endian*/
};

struct sockaddr_in {
    uint8_t         sin_len;     /*long de la estruc (16)*/
    sa_family_t     sin_family;  /*AF_INET*/
    in_port_t       sin_port;    /*num de Puerto de 16 bits*/
    struct in_addr  sin_addr;    /*direccion IP*/
    char            sin_zero[8]; /*no usado*/
};
```

Ejemplo 5.1. Estructura para direcciones

2. Construcción de mensajes

En el intercambio de información, el emisor y el receptor deben estar de acuerdo en cómo esa información será codificada, quién envía la información, en qué momento, y cómo finalizará la comunicación. Los protocolos TCP/IP transportan los bytes de datos de usuario sin examinarlos o modificarlos, brindando a las aplicaciones gran flexibilidad en la codificación de la información. El protocolo de aplicación debe especificar el formato de los datos de forma que el receptor pueda interpretarlos correctamente [2].

2.1 Codificación de los datos

Para la codificación de los datos pueden usarse diversos formatos. Sin embargo, es importante notar las implicaciones de la elección de un formato sobre otro.

Por ejemplo, si se quieren enviar varios datos numéricos en un mismo mensaje, es posible codificar cada dígito numérico en un carácter ASCII. Esto implicaría la necesidad de agregar un carácter especial entre cada número, de forma de saber cuándo termina uno y comienza el otro, y un carácter especial que indique el final de la lista [2].

Otra opción es colocar los números en formato binario. En este caso, es importante fijar los tamaños de cada número, y garantizar que ambos sistemas (emisor y receptor) usen el mismo tamaño [2]. Además, la representación binaria conlleva un problema adicional, que se discute en la siguiente sección.

2.2 Ordenamiento de los bytes

Consideremos un entero de 16 bits. Existen dos formas de almacenar los 2 bytes en memoria: con el byte menos significativo en la dirección de inicio, conocido como *ordenamiento little endian*, o con el byte más significativo en la dirección de inicio, conocido como *ordenamiento big endian* [5]. Estos ordenamientos se muestran en la Figura 5.1.

Número a almacenar: 0x01234567

Dirección	0x100	0x101	0x102	0x103
Big endian	01	23	45	67
Little endian	67	45	23	01

Figura 5.1: Representacion Big Endian vs. Little Endian

Lamentablemente no existe un estándar para el ordenamiento de los bytes, por lo que existen sistemas que usan ambos formatos [5]. Esto implica que la comunicación entre dos sistemas que no empleen el mismo método de ordenamiento puede ser caótica, generando una mala interpretación de los datos. Afortunadamente existe una solución. Por convención, el ordenamiento que usa la red es big endian; existen rutinas estándar para convertir enteros de 2 o 4 bytes del formato nativo del host al formato de la red en el momento del envío, y también para la conversión inversa en la recepción [2].

2.3 Alineación y relleno

Cuando los mensajes contienen múltiples campos en formato binario de diferentes tamaños, se deben tomar consideraciones de alineación. En algunos sistemas, una estructura de datos con campos heterogéneos (por ejemplo, 4 y 2 bytes) serán rellenadas para alcanzar frontera de palabra, este *relleno* debe ser previsto para evitar malas interpretaciones en la recepción.

Algunas formas de evitar el relleno por alineación son colocar campos múltiplos de palabras, o reordenar los campos para que estén alineados a frontera de palabra [2].

3. Sockets para TCP

La Figura 5.2 [5] muestra la línea de tiempo de un escenario típico entre un cliente y un servidor TCP. Primero el servidor se inicia, luego el cliente solicita una conexión con el servidor. Se asume que el cliente envía una petición (*request*) al servidor, y este le envía una respuesta. La interacción continúa hasta que el cliente cierra la conexión.

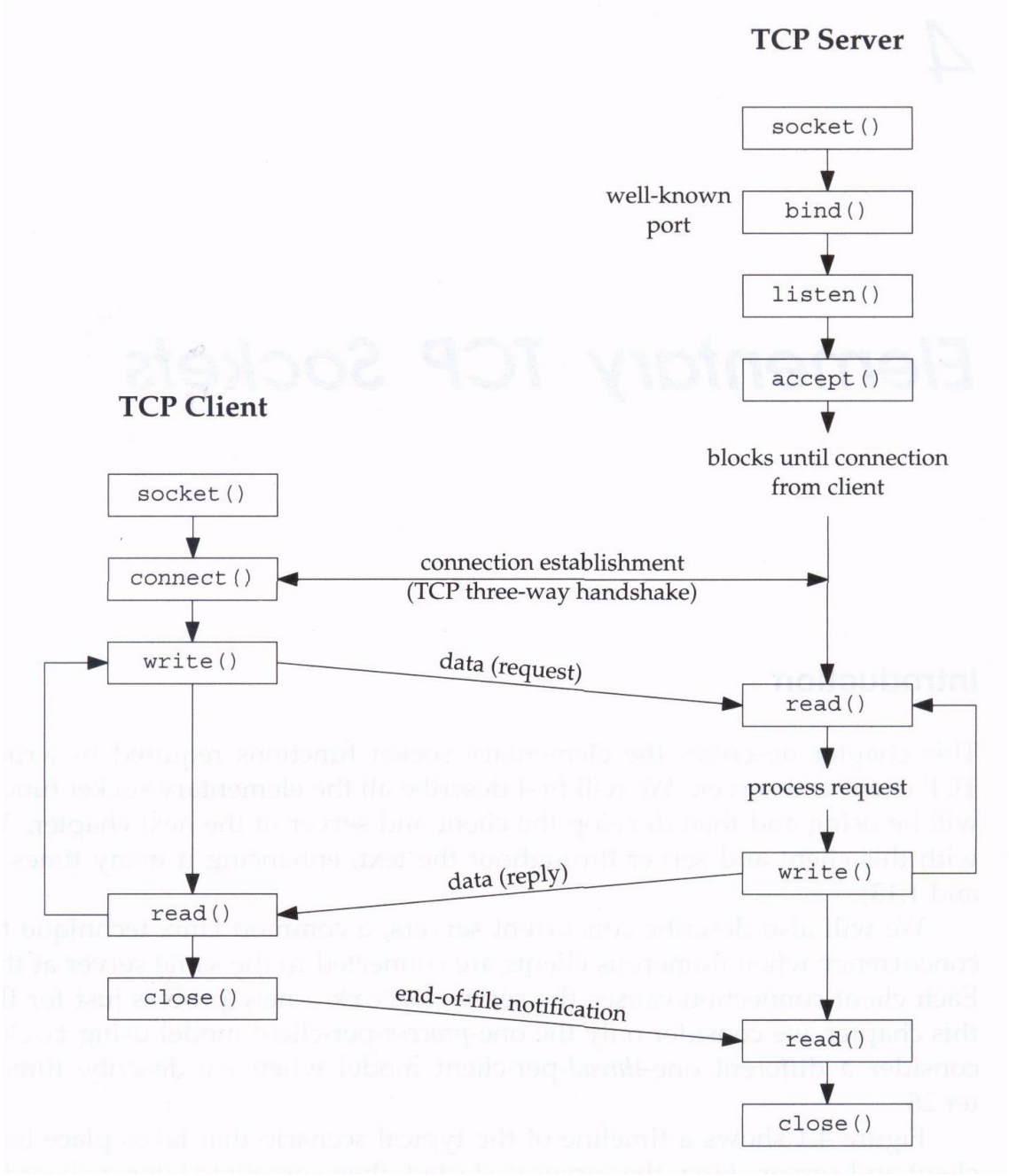

Figura 5.2. Funciones de sockets para cliente/servidor elemental TCP

La función `socket()` fue descrita previamente. La función `connect()` es usada por el cliente TCP para establecer una conexión con el servidor.

```
int connect(int sockfd, const struct sockadd *servadd, socklen_t addlen);
```

Donde *sockfd* es el descriptor del socket devuelto por la función `socket()`. El segundo y tercer argumento son la estructura de la dirección y su longitud. La función devuelve 0 en caso de éxito, y -1 en caso de error [5].

La función `bind()` asigna una dirección al socket (normalmente, una dirección IP y un número de puerto) [5].

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

Cuando un socket es creado, se asume como activo. La función `listen()` convierte un socket desconectado en pasivo, indicando al sistema operativo que debe aceptar solicitudes de conexión entrantes para él.

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

El segundo parámetro de esta función especifica el número máximo de conexiones que el kernel debe encolar para este socket.

La función `accept()` es invocada por un servidor para devolver la siguiente conexión de la cola asignada a ese socket [5].

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

El Ejemplo 5.2 muestra un cliente eco TCP. El servidor simplemente devuelve en forma de "eco" lo que reciba del cliente. La cadena de caracteres a ser devuelta es provista como un argumento de la línea de comandos en el cliente.

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

#define   MAX    5000


int main(int argc, char *argv[])
{
int s;
struct sockaddr_in ser, cli;
char buf[MAX];

if(argc!=4)
   {
   fprintf(stderr, "Usage: %s destinationIP port message\n", argv[0]);
   return(1);
   }

if((s=socket(PF_INET, SOCK_STREAM, 0))==-1)
   {
   fprintf(stderr, "Problem to create socket\n");
   return(2);
   }

cli.sin_family=AF_INET;
cli.sin_port=htons(0);
/* Escoger cualquier puerto disponible en la maquina. */
cli.sin_addr.s_addr=htonl(INADDR_ANY);
```

```
/* Escoge la direccion IP del cliente (cualquiera). */

if(bind(s, (struct sockaddr *) &cli, sizeof(struct sockaddr_in))==-1)
   {
   fprintf(stderr, "Problem when naming the socket\n");
   return(3);
   }

ser.sin_family=AF_INET;
ser.sin_addr.s_addr=inet_addr(argv[1]);
ser.sin_port=htons(atoi(argv[2]));

connect(s, (struct sockaddr*) &ser, sizeof(struct sockaddr_in));
/* Establece la conexion con el servidor. */

/* Envia la cadena al servidor. */
send(s, argv[3], strlen(argv[3])+1, 0);
printf("->Enviando: %s, a: %s por el puerto: %s\n",argv[3], argv[1], argv[2]);

/* Recibe la cadena del servidor. */
recv(s, buf, sizeof(buf) , 0);
printf("<-Recibido: %s\n", buf);

/* Cierra el socket. */
close(s);

return(0);
}
```

Ejemplo 5.2. Cliente eco TCP

El Ejemplo 5.3 muestra el servidor eco correspondiente al Ejemplo 5.2.

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

#define   MAX    5000


int main(int argc, char *argv[])
{
int s_ser, s_cli;
struct sockaddr_in ser, cli;
char buf[MAX];
int length;

if(argc!=2)
   {
   fprintf(stderr, "Usage: %s port\n", argv[0]);
   return(1);
   }

if((s_ser=socket(PF_INET, SOCK_STREAM, 0))==-1)
   {
   fprintf(stderr, "Problem to create socket\n");
   return(2);
   }
```

```
ser.sin_family=AF_INET;
ser.sin_port=htons(atoi(argv[1]));
/* Asigna el puerto especificado en la linea de comandos. */
ser.sin_addr.s_addr=htonl(INADDR_ANY);
/* Escoge la direccion IP del servidor (todas). */

if(bind(s_ser, (struct sockaddr *) &ser, sizeof(struct sockaddr_in))==-1)
   {
   fprintf(stderr, "Problem when naming the socket\n");
   return(3);
   }
/* Asigna un nombre al socket. */

listen(s_ser, 10);
/* Crea la cola. El puerto queda abierto. */
printf("Servidor escuchando en el puerto: %s\n", argv[1]);

/* El while permite atender a multiples clientes. */
while(1)
   {
   length=sizeof(struct sockaddr_in);
   s_cli=accept(s_ser, (struct sockaddr *) &cli, &length);
   /* Acepta una conexion. */

   recv(s_cli, buf, sizeof(buf) ,0);
   /* Recibe la cadena del cliente. */

   send(s_cli, buf, strlen(buf)+1, 0);
   /* Devuelve la cadena al cliente. */
   printf("Recibido de %s: %s\n",  inet_ntoa(cli.sin_addr), buf);

   close(s_cli);
   /* Cierra la conexion con el cliente actual. */
   }

close(s_ser);
/* Cierra el socket. */

return(0);
}
```

Ejemplo 5.3. Servidor eco TCP.

4. Sockets para UDP

Los sockets UDP no necesitan conectarse antes de usarse. Tan pronto como es creado, un socket UDP puede ser usado para enviar/recibir mensajes hacia/desde cualquier dirección. Para permitir la especificación de la dirección de destino en cada mensaje, el API provee una rutina que normalmente es usada con los sockets UDP: `sendto()`. De manera similar, la rutina `recvfrom()` devuelve la dirección de origen del mensaje recibido, así como el mensaje en sí [2].

El Ejemplo 5.4 muestra el cliente eco UDP, y el Ejemplo 5.5 muestra el servidor correspondiente.

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

```
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdlib.h>

#define   MAX    5000


int main(int argc, char *argv[])
{
int s;
struct sockaddr_in ser, cli;
char buf[MAX];
int length;

if(argc!=4)
   {
   fprintf(stderr, "Usage: %s destinationIP port message\n", argv[0]);
   exit(1);
   }

if((s=socket(PF_INET, SOCK_DGRAM, 0))==-1)
   {
   fprintf(stderr, "Problem to create socket\n");
   exit(2);
   }

cli.sin_family=AF_INET;
/* Escoger cualquier puerto disponible en la maquina. */
cli.sin_port=htons(0);
/* Escoge cualquier IP de la maquina. */
cli.sin_addr.s_addr=htonl(INADDR_ANY);

/* Asigna un nombre al socket. */
if(bind(s, (struct sockaddr *) &cli, sizeof(struct sockaddr_in))==-1)
   {
   fprintf(stderr, "Problem when naming the socket\n");
   exit(3);
   }

ser.sin_family=AF_INET;
ser.sin_addr.s_addr=inet_addr(argv[1]);
ser.sin_port=htons(atoi(argv[2]));

/* Envia la cadena al servidor. */
sendto(s, argv[3], strlen(argv[3])+1, 0, (struct sockaddr *) &ser,
       sizeof(struct sockaddr_in));
printf("->Enviando: %s, a: %s por el puerto: %s\n",argv[3], argv[1], argv[2]);

/* Recibe la cadena del servidor. */
length=sizeof(struct sockaddr_in);
recvfrom(s, buf, sizeof(buf) , 0, (struct sockaddr *) &ser, &length);
printf("<-Recibido: %s\n", buf);

/* Cierra el socket. */
close(s);

exit(0);
}
```

Ejemplo 5.4. Cliente eco UDP.

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

#define   MAX    5000

int main(int argc, char *argv[])
{
int s;
struct sockaddr_in ser, cli;
char buf[MAX];
int length;

if(argc!=2)
   {
   fprintf(stderr, "Usage: %s port\n", argv[0]);
   exit(1);
   }

if((s=socket(PF_INET, SOCK_DGRAM, 0))==-1)
   {
   fprintf(stderr, "Problem to create socket\n");
   exit(2);
   }

ser.sin_family=AF_INET;
ser.sin_port=htons(atoi(argv[1]));
/* Asigna el puerto especificado en la linea de comandos.  */
ser.sin_addr.s_addr=htonl(INADDR_ANY);
/* Escoge la direccion IP del servidor (todas).            */

if(bind(s, (struct sockaddr *) &ser, sizeof(struct sockaddr_in))==-1)
   {
   fprintf(stderr, "Problem when naming the socket\n");
   exit(3);
   }
/* Asigna un nombre al socket.                             */

printf("Servidor escuchando en el puerto: %s\n", argv[1]);

/* El while permite atender a multiples clientes. */
while(1)
   {
   length=sizeof(struct sockaddr_in);
   recvfrom(s, buf, sizeof(buf) ,0, (struct sockaddr *) &cli, &length);
   /* Recibe la cadena del cliente. */

   printf("Recibido de %s: %s\n",  inet_ntoa(cli.sin_addr), buf);

   sendto(s, buf, strlen(buf)+1, 0, (struct sockaddr *) &cli, sizeof(cli));
   /* Devuelve la cadena al cliente. */
   }
/* Cierra el socket. */
close(s);

exit(0);
}
```

Ejemplo 5.5. Servidor eco UDP.

5. Sockets crudos

Es posible construir aplicaciones que usen IP directamente, eliminando el uso de protocolos de capa de transporte; esta técnica recibe el nombre de sockets crudos (raw sockets), y es usada con menor frecuencia. Los sockets crudos proveen tres características que no ofrecen los sockets TCP y UDP [5]:

- Permiten escribir y leer mensajes ICMP e IGMP.
- Un proceso puede leer y escribir datagramas IP con algún valor específico en el campo *protocol* que no sea tratado por el kernel.
- Con un socket crudo, un proceso puede construir su propia cabecera IP, usando la opción de socket `IP_HDRINCL`.

En esta sección se describen la creación, entrada y salida de los sockets crudos.

5.1 Creación de sockets crudos

Los pasos para la creación de un socket crudo son [5]:

- La función `socket` crea un socket crudo cuando el segundo argumento es `SOCK_RAW`. El tercer argumento (protocolo) es normalmente un valor distinto de cero. El Ejemplo 5.7 [5] muestra la creación de un socket crudo para IPv4.

```
int    sockfd;

sockfd = socket (PF_INET, SOCK_RAW, protocol);
```

Ejemplo 5.7. Creación de un socket crudo.

  donde *protocol* es una de las constantes `IPPROTO_XX`, definidas en `<netinet/in.h>`, como `IPPROTO_ICMP`.

- La opción `IP_HDRINCL` puede habilitarse como se muestra en el Ejemplo 5.8 [5]:

```
const int on = 1;

if (setsockopt(sockfd, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on))<0)

    error
```

Ejemplo 5.8. Uso de la opción IP_HDRINCL.

- La función `bind` puede ser llamada en el socket crudo, pero no es común. Esta función sólo fija la dirección local: no hay concepto de puerto en sockets crudos. Si no es invocada, el kernel fija la dirección IP primaria de la interfaz de salida.

- La función `connect` puede ser llamada en el socket crudo, pero no es muy común. Esta función sólo fija la dirección foránea.

5.2 Salida de sockets crudos

La salida de un socket crudo sigue las siguientes reglas [5]:

- La salida normal es realizada invocando a la función `sendto` o `sendmsg` y especificando la dirección IP del destino. Las funciones `write` y `send` también pueden ser invocadas si el socket está "conectado" (función `connect`).

- Si la opción `IP_HDRINCL` no está fijada, la dirección de inicio de los datos que debe enviar el kernel especifica el primer byte que sigue a la cabecera IP, ya que el kernel construirá la cabecera IP y la concatenará a los datos del proceso.

- Si la opción `IP_HDRINCL` está fijada, la dirección de inicio de los datos que debe enviar el kernel especifica el primer byte de la cabecera IP. El proceso construye el paquete IP completo, excepto por: (i) el campo de identificación puede ser 0, lo que le indica al kernel que fije este valor; (ii) el kernel siempre calcula y almacena el checksum de la cabecera; y (iii) las opciones IP pueden o no ser incluidas.

- El kernel fragmenta los paquetes que exceden el MTU (Maximun Transfer Unit) de la interfaz saliente.

5.3 Entrada de sockets crudos

La primera pregunta que se debe contestar acerca de la entrada a un socket crudo es: ¿cuáles datagramas IP debe pasar el kernel al socket crudo? Se aplican las siguientes reglas [5]:

- Los paquetes TCP o UDP recibidos nunca se entregan a un socket crudo. Si un proceso debe leer datagramas IP que contienen paquetes TCP o UDP, los paquetes deben ser leídos en la capa de enlace de datos.

- La *mayoría* de los paquetes ICMP son entregados al socket crudo luego de que el kernel ha finalizado de procesar el mensaje ICMP. Algunas implementaciones (derivadas de Berkeley) entregan a un socket crudo todos los paquetes ICMP recibidos a excepción de echoRequest, timestampRequest, y addressMaskRequest, que son tratados por el kernel.

- *Todos* los mensajes IGMP son pasados a un socket crudo luego de que el kernel ha finalizado de procesarlos.

- *Todos* los datagramas IP con un valor en el campo *protocol* que el kernel no conozca son pasados a un socket crudo.

- Si el datagrama llega fragmentado, ninguno de ellos se entrega a un socket crudo hasta que todos los fragmentos han llegado y el datagrama ha sido reensamblado.

Cuando el kernel tiene un datagrama IP que debe entregar a un socket crudo, se realiza una serie de pruebas para determinar cuáles son los sockets que deben recibir el datagrama. Una copia del datagrama IP se entrega a cada socket que apruebe la verificación hecha por el kernel. La prueba consta de tres verificaciones, que son [5]:

- Si se especifica un protocolo distinto de cero cuando se crea el socket crudo (tercer argumento de `socket`), entonces el campo *protocol* del datagrama entrante debe coincidir con este valor.

- Si una dirección local está "asociada" al socket crudo a través de la función `bind`, entonces la dirección de destino del datagrama recibido debe coincidir con la dirección asociada al socket.

- Si una dirección IP destino fue especificada para el socket crudo mediante la función `connect`, entonces la dirección IP origen del datagrama recibido debe coincidir con la dirección con la que se hizo la conexión.

Si un socket crudo se crea con protocolo 0, y no fueron invocadas las funciones `bind` y `connect`, entonces el socket crudo recibe una copia de cada datagrama que el kernel entrega a los sockets crudos [5].

6. Manejo de Difusión

Toda la discusión hasta este momento (incluyendo los ejemplos) ha tratado con direccionamiento unicast: un proceso se comunica exactamente con un proceso, lo que puede generar un uso ineficiente de los recursos de la red cuando se desea enviar datos a múltiples receptores. A pesar de que TCP sólo soporta este tipo de direccionamiento (unicast), UDP soporta difusión [5]. Existen dos casos básicos de difusión, broadcast y multicast.

6.1 Broadcast

El envío broadcast de datagramas UDP es similar al de datagramas unicast. La diferencia principal es la forma de la dirección. En la práctica, hay dos tipos de direcciones broadcast: local y dirigido. Una dirección IPv4 de broadcast local (255.255.255.255) envía el mensaje a todos los hosts en un dominio de broadcast. Mensajes de este tipo no son reenviados por routers [2].

El broadcast dirigido permite enviar mensajes de este tipo a todos los hosts en una red específica. Una dirección de broadcast dirigido mantiene su porción de identificación de red, y coloca todos los bits de la porción de host en 1 [2].

Por defecto, los sockets no pueden enviar mensajes tipo broadcast. Para habilitar su uso, se fija la opción SO_BROADCAST, como se muestra en el Ejemplo 5.9 [2].

```
setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, (void *)
            &broadcastPermission,sizeof(broadcastPermision));
```

Ejemplo 5.9. Habilitar uso de broadcast.

6.2 Multicast

Como en el caso de broadcast, el multicast de UDP es muy similar al de unicast. Nuevamente, la principal diferencia es la forma de la dirección, una dirección multicast define un conjunto de receptores. Un rango del espacio de direcciones IPv4 fue asignado para uso multicast (clase D), comprendido entre 224.0.0.0 hasta 239.255.255.255 [2].

Las únicas diferencias significativas entre emisores broadcast y multicast son [2]:

- En el caso multicast no es necesario fijar los permisos para el envío de datagramas.
- En el caso multicast se fija el valor del TTL.

7. Demonios

Un demonio es un proceso que se ejecuta en segundo plano (*background*) y no está asociado con un terminal [5]. Los sistemas operativos generalmente tienen muchos procesos demonios ejecutándose, realizando diferentes tareas administrativas.

Existen varias formas de iniciar un demonio [5]:

1. Durante el arranque.
2. Muchos servidores de red son iniciados por el "superservidor" `inetd`. Una vez activo escucha peticiones de red y cuando llega una solicitud, invoca al servidor apropiado.
3. La ejecución de programas en intervalos regulares es realizada por el demonio `cron`, y los programas que éste invoca se ejecutan como demonios.
4. Pueden ser ejecutados desde un terminal de usuario, bien sea en segundo plano o no.

7.1 inetd

En un sistema Unix típico puede haber varios servidores en existencia esperando solicitudes de los clientes (FTP, Telnet, TFTP, etc.). Antiguamente, cada uno de estos servicios tenía un proceso asociado. Estos procesos eran iniciados en el arranque del sistema, desde el archivo `/etc/rc`, y cada proceso realizaba tareas de inicio casi idénticas: crear un socket, hacer un `bind` del socket con un puerto bien conocido del servidor, esperar una conexión (TCP) o un datagrama (UDP)

según sea el caso, y luego invocar a la función `fork`[1]. El proceso hijo presta servicios al cliente y el proceso padre espera la próxima solicitud. Existen dos problemas con este modelo:

1. Todos los demonios tienen un código de inicio muy similar.
2. Cada demonio toma una entrada en la tabla de procesos, a pesar de estar dormido la mayoría del tiempo.

Versiones más recientes simplifican este esquema al proveer un superservidor: `inetd`. Este demonio puede ser usado por servidores que usen tanto TCP como UDP. No maneja otros protocolos.

Una vez iniciado, el `inetd` lee y procesa su archivo de configuración, que especifica los servicios que deberá manejar y qué acciones tomar en la llegada de un nuevo requerimiento de cliente. Las acciones que sigue el demonio `inetd` son [5]:

- Al inicio, lee su archivo de configuración y crea un socket de tipo aproximado (flujo o datagramas) para cada servicio especificado en el archivo.
- Se invoca la función `bind` para el socket, especificando el puerto y la dirección IP.
- Para sockets TCP se invoca la función `listen`, para aceptar conexiones entrantes. Este paso es ignorado para sockets UDP.
- Luego de que se crean los sockets, se invoca a la función `select` para esperar cuando los sockets pasen al estado "listo para leer" o *readable*.
- Cuando un socket está listo para ser leído, si se trata de un socket TCP se invoca la función `accept` para aceptar la nueva conexión.
- El demonio `inetd` invoca a la función del sistema `fork`, y el proceso hijo maneja la solicitud del cliente.

  Luego se invoca un `exec` para ejecutar el programa servidor apropiado que maneje las solicitudes de los clientes, según los parámetros especificados en el archivo de configuración.
- Si el socket maneja un flujo TCP, el proceso padre debe cerrar los sockets conectados. El proceso padre invoca nuevamente a la función `select`, en espera del próximo socket a ser leído.

[1] La función al sistema `fork` crea una copia del proceso invocador; la copia es llamada "hijo". El proceso invocador se denomina "padre".

8. Opciones avanzadas: ioctl

La función `ioctl` ha sido tradicionalmente la interfaz de sistema usada para todo lo que no encajaba elegantemente en otra categoría [5]. Esta llamada al sistema provee una interfaz de comandos genérica usada por un proceso para acceder a las características de dispositivos que no son soportadas por las llamadas estándar al sistema [4]. El prototipo de la función es:

```
#include <unistd.h>
int ioctl(int fd, unsigned long com, …);
```

El primer parámetro es un descriptor, usualmente un dispositivo o una conexión de red. Cada tipo de descriptor soporta su propio conjunto de comandos especificados por el segundo argumento. Un tercer argumento es un puntero cuyo tipo dependerá del comando invocado [4]. La función devuelve 0 en caso de éxito, y -1 en caso de error; y afecta un archivo abierto referenciado por el argumento *fd* [5].

Un uso común de `ioctl` por los programas de red (típicamente servidores) es obtener información de las interfaces del host cuando el programa inicia: las direcciones, soporte de broadcast y multicast, etc [5].

8.1 Información de interfaz

Uno de los primeros pasos usados por muchos programas que tratan con interfaces de red es obtener del kernel las interfaces configuradas en el sistema. Esto se logra con la solicitud `SIOCGIFCONF`, que usa la estructura `ifconf`, que a su vez usa la estructura `ifreq` [5].

Antes de invocar a `ioctl`, se asigna un buffer y una estructura `ifconf` que es posteriormente inicializada. La Figura 5.3 [5] muestra este proceso, asumiendo un tamaño de buffer de 1024 bytes. El tercer argumento de la función `ioctl` es un apuntador a la estructura `ifconf`.

Asumiendo que el kernel devuelve dos estructuras `ifreq`, se obtiene el ordenamiento mostrado en la Figura 5.4. Las porciones marcadas fueron modificadas por `ioctl`. El buffer fue llenado con las dos estructuras devueltas, y el campo `ifc_len` de la estructura `ifconf` fue actualizado para mostrar la cantidad de información almacenada en el buffer. Se asume que cada estructura `ifreq` ocupa 32 bytes.

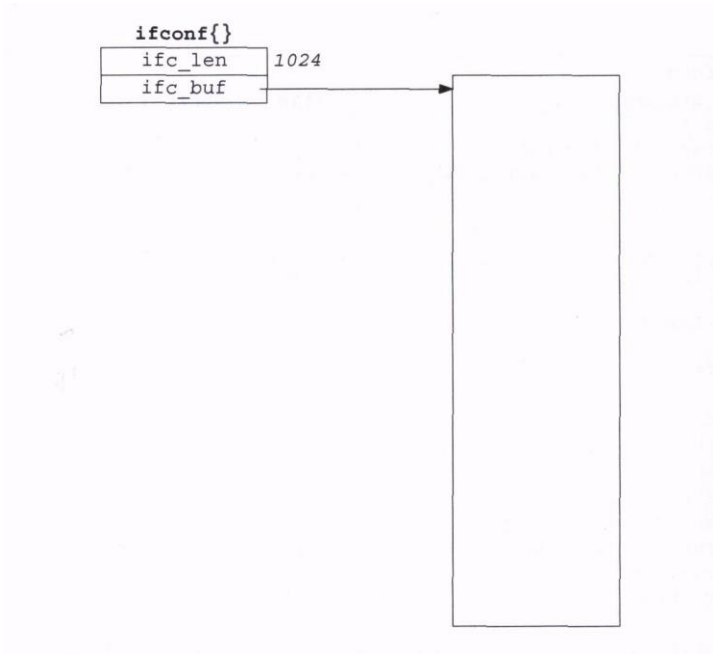

Figura 5.3. Inicialización de la estructura `ifconf` antes de `SIOCGIFCONF`

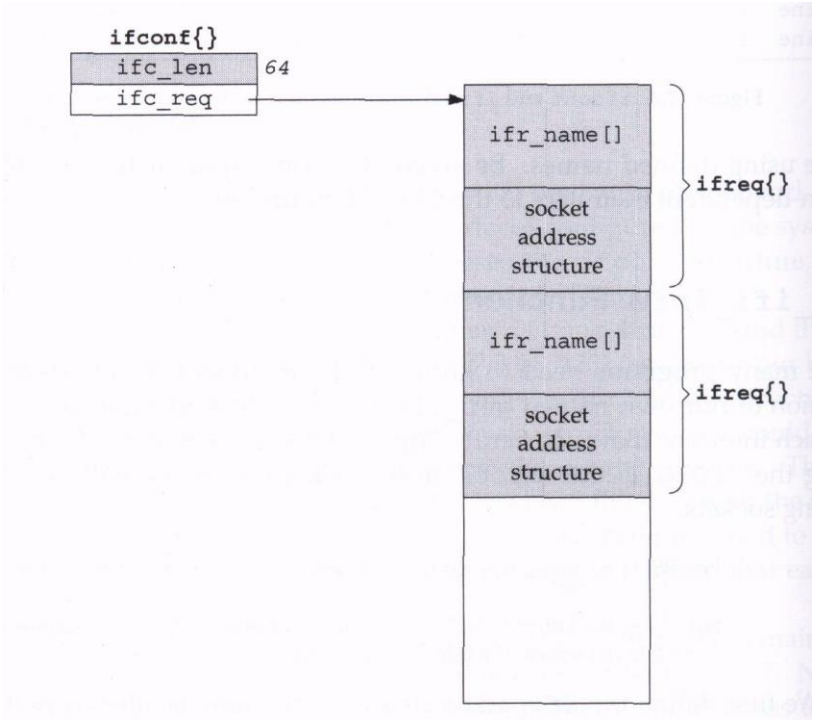

Figura 5.4. Valores devueltos por `SIOCGIFCONF`

9. Referencias

1. Comer, D. TCP/IP: Principios básicos, protocolos y arquitectura. Prentice Hall. 3ª ed. 1996.
2. Donahoo, M. and Calvert, K. TCP/IP Sockets in C: Practical Guide for Programmers. Morgan Kaufmann. 2001.
3. Stevens, R. TCP/IP Illustrated, Volume 1: The Protocols. Addison Wesley. 1994.
4. Stevens, R. and Wright, G. TCP/IP Illustrated, Volume 2: The Implementation. Addison Wesley. 1995.
5. Stevens, R. Et Al. UNIX Network Programming Volume 1: The Sockets Networking API. Addison Wesley Professional. 3rd ed. 2004.