ÍNDICE

1. PROGRAMACIÓN DISTRIBUIDA UTILIZANDO SOCKETS

1.1. El concepto de socket

Los sockets son uno de los mecanismos que permiten a una aplicación distribuida acceder a los servicios proporcionados por los protocolos de la capa de transporte (TCP, UDP en Internet). El concepto de socket fue introducido en la versión 4.2BSD de Unix. Un socket puede verse como un punto final de comunicación donde se encuentran el programa de aplicación y la capa de transporte. Como analogía podríamos pensar en una bandeja de correo saliente en la cual depositamos el correo que queremos enviar y, varias veces al día, la persona encargada de repartir el correo en nuestra organización la inspecciona y recoge de ella las cartas que queramos enviar y tramita su envío. En el extremo del destinatario, habrá algún tipo de buzón donde el cartero deja las cartas para que las recoja el destinatario.

Para los programadores de aplicaciones el socket va a ser una abstracción del sistema de ficheros de una máquina Unix, ya que se accede a un socket a través de un descriptor y las funciones que se utilizan para leer y escribir del socket son las mismas que se utilizan para leer y escribir ficheros

1.2. Tipos de direcciones

Siempre que dos procesos no relacionados quieran comunicarse debe de haber algo que ambos deben de conocer, por así decirlo un sitio de encuentro donde ambos puedan reunirse. Por ejemplo, en el caso de que los procesos se comuniquen a través de colas de mensajes, ambos deben de conocer la clave que utilizan para identificar la cola. En el caso de los sockets, hay varias formas de identificar a un socket.

Muchas de las llamadas al sistema que describiremos a continuación (por ejemplo **bind**) necesitan un puntero a una estructura de identificación del socket. La definición de esta estructura se encuentra en el fichero <sys/socket.h> y se muestra a continuación:

```
struct sockaddr {
  u_short sa_family;     /* familia de direcciones: AF_XXX */
  char sa_data[14];      /* una dirección de hasta 14 bytes específica de cada protocolo */
}
```

Esta estructura de identificación del socket es genérica y permite que las direcciones de los sockets sean pasados a las funciones que las utilizan independientemente del esquema de direccionamiento que utilice el socket.

La forma más sencilla de identificar un socket es la que se conoce como esquema de direccionamiento o identificación de tipo Unix. En este esquema los sockets se identifican por nombres de ficheros tipo Unix. Este esquema de direccionamiento sólo permite comunicar procesos que residen en la misma máquina. La estructura utilizada para identificar un socket tipo Unix se muestra a continuación:

```
struct sockaddr _un{
  short sun_family;      /* familia de direcciones: AF_UNIX*/
  char sun_path[108];    /* un nombre de fichero de hasta 108 bytes */
}
```

La otra forma de identificar un socket y la más universalmente utilizada es el esquema de direccionamiento Internet. Bajo este esquema los sockets se identifican por una dirección IP y un número de puerto. Este esquema de direccionamiento permite no sólo comunicar procesos que se encuentran en la misma máquina, sino también procesos que se encuentran en máquinas distintas conectadas a través de la red. La estructura utilizada para identificar un socket de tipo Internet se muestra a continuación:

```
struct sockaddr _in{
  short sin_family;            /* familia de direcciones: AF_INET*/
  u_short sin_port;            /* Número de puerto */
  struct in_addr sin_addr;     /* Dirección IP */
  char sin_zero[8];            /* relleno */
}
```

Hay que observar que ambas estructuras tienen un campo etiqueta inicial cuyo valor debe fijarse para indicar a la función que las recibe como argumento qué tipo de direccionamiento es el que se está utilizando. La estructura **sockaddr_in** que acabamos de ver contiene un campo de tipo **struct in_addr** el cual almacena la dirección IP del socket. Esta estructura está definida de la siguiente manera:

```
struct in_addr{
  u_long  s_addr;        /* Direccion IP */
}
```

1.3. Tipos de sockets

Hay varios tipos de sockets que pueden utilizar los programadores en el desarrollo de sus aplicaciones distribuidas:

- Sockets orientados a conexión que proporcionan flujos de datos bidireccionales, fiables, secuenciados y no duplicados sin límites en el tamaño de los datos a transmitir.
- Sockets no orientados a conexión que proporcionan flujos de datos bidireccionales en los que los datagramas pueden no llegar a destino o llegar desordenados o quizás duplicados.
- Los sockets “en bruto” (raw sockets) permiten acceder a los protocolos de comunicación subyacentes que soportan las abstracciones de socket. Estos sockets normalmente no están orientados a conexión y dependen de la interfaz que suministre el protocolo.

1.4. Operaciones sobre sockets

1.4.1. Creación de un socket: socket()

Para crear un socket se utiliza esta función cuya sintaxis presentamos a continuación.

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

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

Esta función hace que el sistema cree un socket en el dominio especificado y del tipo especificado. Si el protocolo se deja sin especificar (un valor 0), el sistema seleccionará un protocolo apropiado de entre los que se pueden utilizar en el dominio y que pueda ser utilizado para soportar el tipo de socket solicitado.

Familia	Tipo	Protocolo	Protocolo por defecto
AF_INET	SOCK_DGRAM	IPPROTO_UDP	UDP
AF_INET	SOCK_STREAM	IPPROTO_TCP	TCP
AF_INET	SOCK_RAW	IPPROTO_ICMP	ICMP
AF_INET	SOCK_RAW	IPPROTO_RAW	(en bruto)

Esta función retorna un valor entero positivo (o –1 en caso de fallo) que representa el descriptor del socket y que puede ser utilizado por el resto de funciones de manejo de socket. La familia se especifica mediante una constante simbólica definida en **<sys/socket.h>** y será **AF_UNIX** para un esquema de direccionamiento tipo **UNIX** o bien **AF_INET** para un esquema de direccionamiento Internet. Los tipos de los sockets se representan también por constantes simbólicas definidas en el mencionado fichero de cabecera y pueden ser **SOCK_STREAM** (sockets orientados a conexión), **SOCK_DGRAM** (sockets orientados a datagramas) y **SOCK_RAW** para trabajar directamente a nivel del protocolo IP. Con esta llamada lo único que conseguimos es especificar el elemento protocolo de la asociación representada por la 5-tupla:

{protocolo, dirección local, puerto local, dirección remota, puerto remoto}

Las otras llamadas que veremos a continuación se encargarán de especificar el resto de elementos de la tupla antes de que el socket pueda ser de utilidad.

1.4.2. Asociación de nombre a un socket: bind()

Esta función nos va a permitir asociar un nombre a un socket que todavía no lo tiene (se acaba de crear mediante una invocación a la función **socket()**). La sintaxis de esta función se muestra a continuación:

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

int bind (int socket, const struct sockaddr * addres, int addreslen);
```

Esta función se encarga de especificar los elementos dirección local y puerto local de la asociación representada en la 5-tupla anterior.

El primer argumento de la función es el descriptor del socket al cual queremos asociar una dirección.

El segundo argumento es un puntero a una estructura específica de cada tipo de direccionamiento empleado (estructura **sockaddr_un** para la familia de direcciones **AF_UNIX** y **sockaddr_in** para la familia de direcciones **AF_INET**).

El tercer argumento es el tamaño de esta estructura. Si el **bind()** tiene éxito devuelve 0, en caso contrario devuelve un valor de –1.

La función **bind()** se emplea en tres situaciones:

- Para que un proceso servidor registre su dirección ante el sistema. Esto es necesario tanto para los servidores orientados a conexión como para los no orientados a conexión.
- Un cliente que quiera registrar una dirección específica para sí mismo.
- Los clientes no orientados a conexión necesitan asegurarse de que el sistema les asocia una única dirección, de forma que en el otro extremo el servidor tenga una dirección de retorno válida a la que enviar las respuestas.

1.4.3. Establecimiento del tamaño máximo de cola de peticiones pendientes: listen()

Esta función la invocan solamente los servidores orientados a conexión. Su sintaxis es la siguiente:

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

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

El argumento **backlog** especifica cuántas conexiones pueden estar encoladas en el sistema mientras se espera a que el servidor ejecute una llamada a **accept()**. El argumento de **backlog** normalmente toma el valor 5 que es el máximo valor actualmente permitido.

1.4.4. Aceptar peticiones entrantes: accept()

Después de que un servidor orientado a conexión ejecuta la llamada al sistema **listen()**, debe de invocar a **accept()** para esperar por las peticiones de conexión de los clientes. La sintaxis de esta función es la siguiente:

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

int accept (int sockfd, struct sockaddr * addres, size_t *addreslen);
```

Esta función toma la primera petición de la cola de peticiones de conexión y crea otro socket con las mismas propiedades del socket cuyo descriptor se pasa como primer argumento. Si no hay peticiones de conexión pendientes, esta llamada bloquea al proceso que la invoca hasta que se reciba la primera.

El segundo y tercer argumentos se utilizan para retornar la dirección del proceso cliente que se conecta. El tercer argumento es lo que se denomina un argumento de valor-resultado, ya que el proceso que invoca a la función fija su valor antes de la invocación y después la función utiliza esa posición de memoria para devolver un resultado. Concretamente para esta llamada al sistema, se almacena en la dirección de memoria apuntada por **addreslen** el tamaño de la estructura **sockaddr** que se pasa como segundo argumento y al retornar la llamada al sistema el valor de la dirección de memoria apuntada por **addreslen** contiene el número actual de bytes que la llamada al sistema devuelve en la dirección de memoria apuntada por el segundo argumento. Para el esquema de direccionamiento de Internet, como las direcciones ocupan siempre lo mismo (16 bytes), el valor que almacenamos en la dirección de memoria apuntada por **addreslen** (16) es igual al número de bytes que la llamada al sistema retorna en la dirección de memoria apuntada por el segundo parámetro. En el esquema de direccionamiento tipo Unix el tamaño de la estructura puede diferir de la longitud actual.

Esta función devuelve hasta tres valores: un valor entero que si es –1 indica que hubo un fallo y si es positivo va a ser un nuevo descriptor de socket a través del cual el cliente se comunica con el servidor, la dirección del proceso cliente como segundo valor y como tercer valor el tamaño de esta dirección.

Esta función se encarga de especificar los últimos 2 elementos desconocidos de la 5-tupla de asociación, a saber, la dirección de remota y el puerto remoto.

1.4.5. Conexión a un socket remoto: connect()

Esta función la debe invocar el proceso cliente (después de crear el socket del cliente) para establecer una conexión con un proceso servidor. La sintaxis de esta función es la siguiente:

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

int connect (int sockfd, const struct sockaddr * servaddres, size_t *addreslen);
```

El primer parámetro es el descriptor del socket del cliente devuelto al invocar a la función **socket()**.

El segundo y tercer parámetros son punteros a la dirección del servidor y al tamaño de la dirección del socket del servidor respectivamente.

Si la función se ejecuta con éxito devuelve un 0 mientras que en caso de producirse algún error se devuelve –1.

Para la mayoría de los protocolos orientados a conexión, como por ejemplo el TCP, esta llamada supone el establecimiento de la conexión entre el sistema local y el sistema remoto. El cliente no tiene que invocar a **bind()** antes de invocar a **connect()**. La conexión normalmente hace que los otros cuatro elementos de la asociación que quedan por especificar sean especificados.

Un cliente no orientado a conexión puede utilizar también esta llamada al sistema para almacenar la dirección del servidor donde enviará los datos y así no tener que especificar la dirección por cada datagrama que enviamos.

Para más información sobre esta función consultar las páginas del man.

1.4.6. Cerrar un socket: close()

Esta función se utiliza para cerrar un socket. Su sintaxis se muestra a continuación.

```
int close(int fd);
```

Si el socket que está siendo cerrado está asociado con un protocolo que proporciona entrega fiable de paquetes como el TCP, el kernel debe de asegurarse de que cualquier dato que permanezca dentro de los buffers del kernel que todavía no haya sido transmitido lo sea. Normalmente el sistema retorna después de invocar a la función **close()** inmediatamente, pero el kernel todavía intentará enviar los datos que todavía permanezcan encolados. Se pueden especificar opciones para el socket que le indiquen al kernel que una vez cerrado debe de limpiar los buffers y no debe de intentar la transmisión de los datos pendientes.

Si tiene éxito esta función devuelve 0 o –1 en caso de error.

1.4.7. Lectura de datos de un socket orientado a conexión: read()

Una vez que hemos establecido la conexión, el descriptor del socket se comporta como cualquier otro descriptor. Para leer datos del socket utilizaremos esta función. Su sintaxis es la siguiente:

```
#include <unistd.h>
ssize_t read(int socket, void *buf, size_t nbytes);
```

Donde socket es el descriptor del socket, buf es un puntero a la zona de memoria donde se almacenarán los datos leídos y nbytes es el tamaño máximo del buffer donde se almacenarán los datos leídos. Esta función devuelve el número de bytes leídos del socket o –1 en caso de error.

Para más información consultar las páginas del man.

1.4.8. Escritura de datos en un socket orientado a conexión: write()

Una vez que hemos establecido la conexión, el descriptor del socket se comporta como cualquier otro descriptor. Para escribir datos en el socket utilizaremos esta función. Su sintaxis es la siguiente:

```
#include <unistd.h>
ssize_t write(int socket, const void *buf, size_t nbytes);
```

Esta función escribe hasta nbytes en el socket referenciado por el descriptor desde el buffer que comienza en buf. Esta función devuelve el número de bytes escritos en el socket o –1 en caso de error.

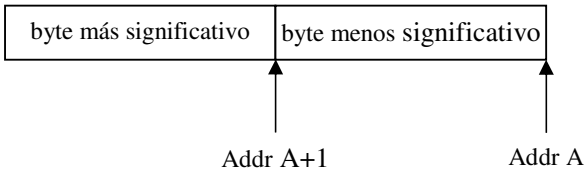
Para más información consultar las páginas del man.

1.4.9. Rutinas de ordenación de bytes

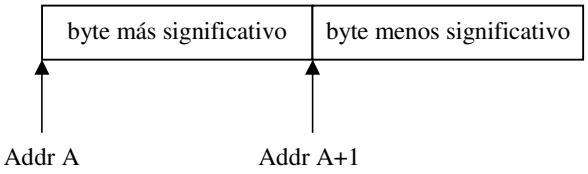
Desgraciadamente no todos los ordenadores almacenan en el mismo orden todos los bytes que forman una variable cuyo almacenamiento necesite más de un byte. Si pensamos en un tipo entero que ocupe 2 bytes, hay dos maneras de almacenar en memoria una variable de este tipo: con el byte menos significativo almacenado en las posiciones de memoria más bajas (little-endian) o con el byte más significativo almacenado en las posiciones de memoria más bajas (big-endian).

Arquitecturas como la familia 68000 de Motorola y la SPARC son del tipo *big-endian*. Arquitecturas como VAX, los transputers y la familia X86 de Intel son *little-endian*. La solución a este problema consiste en un protocolo de red que defina un formato de los datos cuando viajan por la red. El protocolo TCP utiliza un formato big-endian para los enteros de 16 y 32 bits que aparecen en las cabeceras del protocolo, pero el protocolo no tiene control sobre el formato de los datos que las aplicaciones transfieren a través de la red.

Ordenación little endian:



Ordenación big endian:



Las siguientes funciones se encargan de gestionar las diferencias de ordenación potenciales entre las diferentes arquitecturas y los diferentes protocolos de red:

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

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

La función **htonl()** se utiliza para convertir un entero largo del formato interno de representación de la máquina al formato de representación de la red.

La función **htons()** se utiliza para convertir un entero corto del formato interno de representación de la máquina al formato de representación de la red.

La función **ntohl()** se utiliza para convertir un entero largo del formato de representación de la red al formato de representación de la máquina.

La función **ntohs()** se utiliza para convertir un entero corto del formato de representación de la red al formato de representación de la máquina.

Estas funciones trabajan con enteros sin signo, aunque también trabajan con enteros con signo. Implícito en estas funciones se encuentra que un entero corto ocupa 16 bits y un entero largo 32 bits.

1.4.10. Miscelánea de envío recepción de datos para protocolos no orientados a conexión

Estas llamadas al sistema son similares a las anteriores **read()** y **write()** pero necesitan argumentos adicionales.

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

int send (int sockfd, char *buff, int nbytes, int flags);
int sendto (int sockfd, char *buff, int nbytes, int flags, struct sockaddr * destino, int
addrlen);
int recv (int sockfd, char *buff, int nbytes, int flags);
int recvfrom (int sockfd, char *buff, int nbytes, int flags, struct sockaddr *origen, int
*addrlen);
```

Los primeros tres argumentos de las cuatro llamadas al sistema son similares a los primeros tres argumentos de **read()** y **write()**. El argumento flags o bien es cero o bien es un OR a nivel de bits entre las siguientes constantes:

- **MSG_OOB**: permite enviar o recibir datos que deben de ser procesados sin ser buffereados.
- **MSG_PEEK**: permite inspeccionar si hay nuevos mensajes (**recv** o **recvfrom**) sin tener que descargarlos.
- **MSG_DONTROUTE**: permite evitar el enrutado realizado por el protocolo de la capa subyacente (**send** o **sendto**).

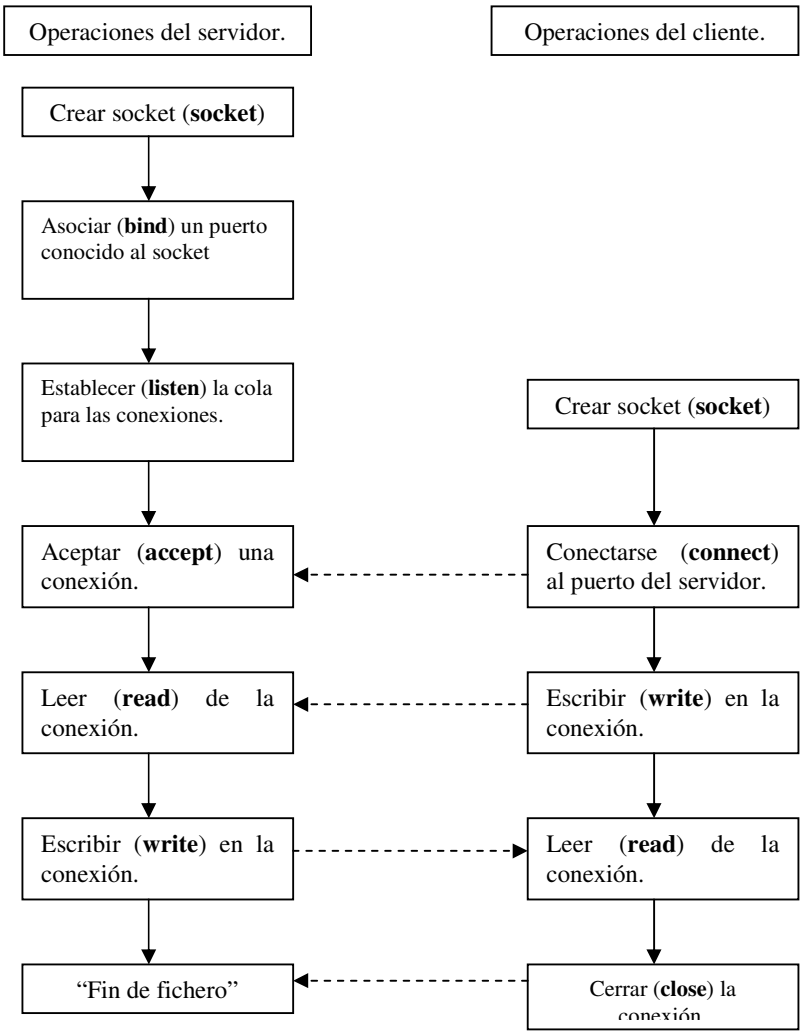
Todas estas cuatro llamadas al sistema retornan la longitud de los datos escritos o leídos como valor devuelto por la función.

1.5. Operaciones típicas en un entorno cliente-servidor orientado a conexión

Vamos, a continuación, a comentar las operaciones típicas realizadas por un cliente y un servidor orientados a conexión, que es el caso más típico que nos podemos encontrar en la Internet (browsers, ftp, telnet, etc.).

En este caso la relación cliente-servidor es asimétrica. El servidor espera pasivamente esperando a que se conecten los clientes. El servidor no conoce de donde le pueden venir las peticiones de conexión. El cliente es el que toma la iniciativa y se conecta con el servidor que precisa. Esta asimetría se refleja en una secuencia de llamadas a operaciones sobre sockets necesarias tanto en el servidor como en el cliente.

En la siguiente figura queda reflejada la secuencia de acciones realizada por uno y otro extremo implicados en la comunicación. En negrita se encuentran las operaciones sobre sockets correspondientes.

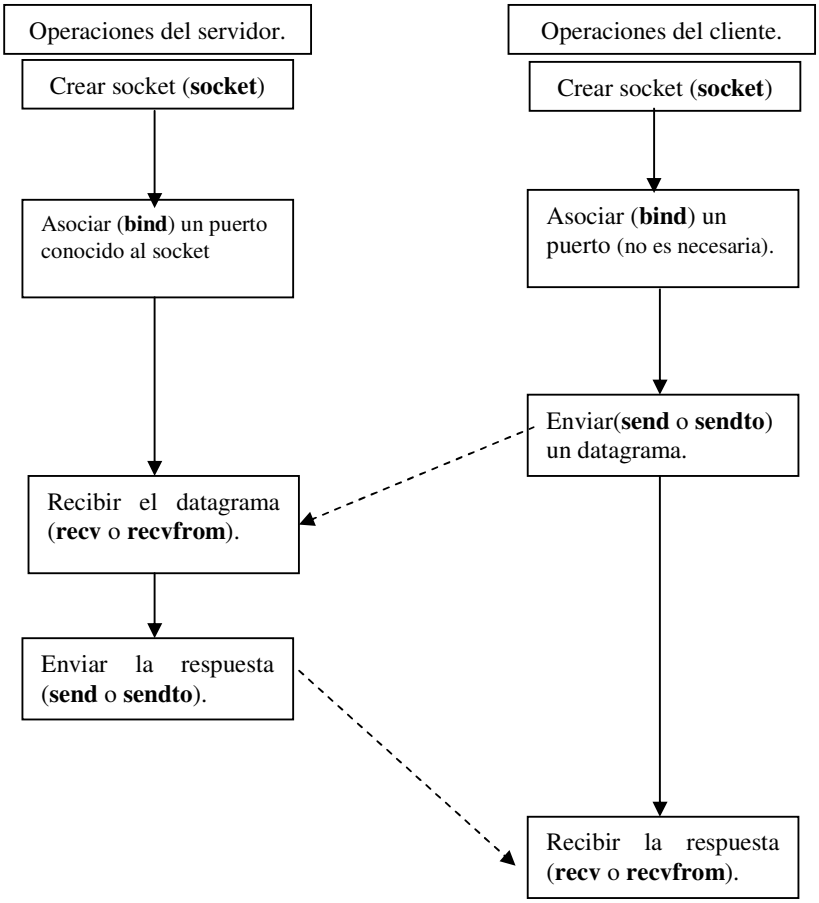


En la figura anterior el diálogo establecido entre el cliente y el servidor implica que primero el servidor lee del socket y el cliente escribe en él, pero puede ser al contrario y normalmente será algo que dependerá de la aplicación.

El servidor espera las conexiones de los clientes en un puerto. Este puerto debe ser conocido por el cliente que se quiera conectar. Se debe de evitar elegir como número de puerto los que aparezcan en el fichero /etc/services, ya que estos corresponden a servicios del S.O. estándares. Así, los 1024 primeros puertos de una máquina están reservados para los servicios del S.O., por tanto deberemos elegir para nuestro servidor un número de puerto no asignado por encima del 1023.

1.6. Operaciones típicas en un entorno cliente-servidor no orientado a conexión

Para un entorno cliente-servidor no orientado a conexión, las llamadas al sistema son diferentes. El cliente no establece una conexión con el servidor. En vez de eso, el cliente solo envía un datagrama al servidor utilizando la llamada al sistema **sendto()** la cual requiere la dirección del destinatario (el servidor) como parámetro. De forma similar, el servidor no tiene que aceptar la conexión del cliente. En vez de eso el servidor solo invoca a la llamada al sistema **recvfrom()** y espera hasta que llegan datos procedentes de algún cliente. La función **recvfrom()** retorna la dirección de red del proceso cliente junto con el datagrama de forma que el servidor puede luego enviar la respuesta al proceso cliente.



1.7. Operaciones sobre sockets avanzadas

A continuación comentaremos otras llamadas al sistema que se pueden utilizar para trabajar con los sockets.

1.7.1. Obtener el "nombre" del proceso con el que estamos conectados: getpeername()

Esta función nos va a permitir obtener el nombre del proceso con el que estamos conectados a través del socket. Por nombre entendemos su dirección IP remota y el número de puerto remoto. La sintaxis de esta función es:

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

int getpeername(int sockfd, struct sockaddr * peer, size_t *addreslen);
```

El último argumento de esta función es un parámetro de valor-resultado.

Para obtener más información consultar las páginas del man.

1.7.2. Obtener el "nombre" asociado al socket: getsockname()

Esta función retorna la dirección IP local y el número de puerto local asociado al socket. La sintaxis de esta función es la siguiente:

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

int getsockname(int sockfd, struct sockaddr * peer, size_t *addreslen);
```

Al igual que ocurre con la función anterior, el último argumento de esta función es un parámetro de valor-resultado.

Para más información consultar las páginas del man.

1.7.3. Obtener/establecer las opciones del socket: getsockopt() y setsockopt()

Estas dos llamadas al sistema manipulan las opciones asociadas con el socket. Para más información consultar las páginas del man.

1.7.4. Destrucción de los datos todavía no enviados/recibidos del socket: shutdown()

La forma normal de terminar una conexión de red es invocando a la función **close()**. Como se mencionó anteriormente, **close()** intenta normalmente enviar cualquier dato que todavía esté pendiente de enviar. La operación **shutdown()** proporciona un mayor control sobre la conexión full-duplex. La sintaxis de esta función es la siguiente:

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

int shutdown(int sockfd, int como)
```

Si el segundo argumento es 0, no se podrán recibir más datos en el socket. Si el segundo argumento es igual a 1 no se enviarán más datos por el socket. Si el segundo argumento fuese 2 se deshabilita el envío y la recepción de datos. Hay que recordar que un socket es un mecanismo de comunicación full-duplex. Los datos que fluyen en una dirección son lógicamente independientes de los datos que fluyen en la dirección contraria. Ese es el motivo por el cual esta función permite cerrar una dirección independientemente de la otra.

1.8. Multiplexación de la entrada: la llamada select()

Un problema clásico de programación en procesos monohilo es el procesamiento de datos asociados a múltiples descriptores de ficheros. Por ejemplo, un proceso servidor puede tener múltiples conexiones de red, cada una representada por un descriptor de fichero. El bloqueo que se da al invocar la llamada al sistema **read()** presenta un problema para un proceso que esté pendiente de la llegada de datos en más de un descriptor de fichero. El servidor puede bloquearse esperando por datos en un descriptor mientras que ya hay datos disponibles en otro. La llegada de nuevos datos, desde la perspectiva del servidor, ocurre de forma aleatoria, por tanto, el proceso no tiene manera de conocer qué entrada llegará primero, si el intento por averiguarlo resulta equivocado, el proceso puede quedar congelado indefinidamente.

Ha habido varias soluciones a este problema. Un programa podría realizar entrada/salida sondeada utilizando entrada/salida no bloqueante sobre cada descriptor de fichero (utilizando la llamada al sistema **fcntl()** con el flag **O_NONBLOCK**). Esto no es eficiente pero funciona.

Una forma más elegante de hacerlo es mediante la llamada al sistema **select()** introducida en 4.2BSD. Esta llamada permite al programa bloquearse hasta que hay datos disponibles en uno o más descriptores de fichero especificados en la llamada al sistema.

La sinopsis de esta llamada al sistema es la siguiente:

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

int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval
*timeout);
```

El primer parámetro de **select()** es el número de bits por examinar en los conjuntos de descriptores de archivo. Este valor debe de ser, al menos, una unidad mayor que el descriptor de archivo más grande que se va a examinar. El parámetro **readfds** indica el conjunto de descriptores que habrá de vigilarse para operaciones de lectura. De manera similar **writefds** representa el conjunto de descriptores que deben vigilarse para operaciones de escritura y **exceptfds** indica los descriptores de archivo a vigilar para condiciones excepcionales.

Al retornar de la llamada, **select()** borra todos los descriptores en cada uno de los conjuntos antes mencionados. El valor retornado por **select()** es el número de descriptores que están listos o –1 en caso de error.

El último parámetro es un valor de tiempo que especifica un **timeout** para el caso de que se quiera que **select()** retorne después de cierto lapso transcurrido, incluso aunque no haya descriptores listos. Cuando **timeout** es **NULL**, **select()** puede hacer el bloqueo indefinidamente. Si **select()** es interrumpida por una señal entonces devuelve –1 y pone **errno** a **EINTR**.

Históricamente el conjunto de descriptores fue implantado como una máscara de bits, pero esto no funciona para más de 32 descriptores de archivo. Ahora es usual representar los conjuntos de descriptores por medio de campos de bits en arrays de enteros, para lo cual se proporcionan las macros **FD_SET**, **FD_CLR**, **FD_ISSET** y **FD_ZERO** para manejar los conjuntos de descriptores de una forma independiente de la implementación.

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

void FD_SET(int fd,fd_set *fdset);
void FD_CLR(int fd,fd_set *fdset);
int FD_ISSET(int fd, fd_set *fdset);
void FD_ZERO(fd_set *fdset);
```

La macro **FD_SET** establece el bit **fdset** que corresponde al descriptor de archivo **fd**, mientras que la macro **FD_CLR** lo borra. La macro **FD_ZERO** borra todos los bits de **fdset**. Estas macros deben de ser empleadas para construir máscaras del descriptor antes de hacer la llamada a **select()**. La macro **FD_ISSET** se utiliza después de **select()** para determinar si el correspondiente bit del descriptor de archivo **fd** está definido en la máscara **fdset()**.

Para más información consultar las páginas del man.

1.9. Obtención de información del estado de los sockets desde la línea de comando

La utilidad **netstat** puede utilizarse para mostrar una gran cantidad de información sobre la actividad de la red así como diversa información sobre el estado de los sockets. Este comando tiene diversas opciones que nos muestran información diversa de los puntos finales de comunicación.

Para más información consultar las páginas del man.

ANEXO A. CLIENTE/SERVIDOR ORIENTADOS A CONEXIÓN EN DOMINIO INTERNET

```
/*
Fichero conexion.h
   Fichero de cabecera para el cliente y el servidor orientados a
   conexion que utiliza el esquema de direccionamiento Internet.
*/


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

#define BUFFER_LECTURA         80
#define PUERTO_SERVIDOR        4756
```

```
/*
Fichero cliente.c
   Ejemplo de cliente orientado a conexion que utiliza el esquema de
   direccionamiento Internet.

*/

#include <stdio.h>
#include <string.h>
#include "conexion.h"


void main (int argc, char * argv[])
{
  FILE *fp;
  struct sockaddr_in servidor; /* La direccion del servidor se compone aquí */
  struct hostent *host_info;
  char datos[BUFFER_LECTURA];
  char *nombre_servidor;
  int sock,leidos;

  if (argc<2)
  {
    fprintf(stderr,"Forma de uso: cliente fich [dir_servidor]\n");
    exit(1);
  }
  /* Obtenemos la direccion del servidor desde la línea de comandos */

  nombre_servidor=(argc>2) ? argv[2] : "localhost";

  /* Creamos el socket del cliente */

  sock = socket(AF_INET, SOCK_STREAM, 0);
  if (sock<0)
  {
    fprintf(stderr,"Error creando el socket. \n");
    exit(2);
  }
  host_info = gethostbyname(nombre_servidor);
  if (host_info==NULL)
  {
    fprintf(stderr,"%s: nombre de host desconocido: %s\n",argv[0],nombre_servidor);
    exit(3);
  }
  /* Establecer la direccion de conexion del socket del servidor para
     luego conectarnos
   */
  servidor.sin_family=host_info->h_addrtype;
  memcpy((char *) &servidor.sin_addr,host_info->h_addr,host_info->h_length);
  servidor.sin_port=htons(PUERTO_SERVIDOR);

  if (connect(sock, (struct sockaddr *) &servidor, sizeof(servidor))<0)
  {
    fprintf(stderr,"Error conectando con el servidor\n");
    exit(4);
  }
```

```
  /* Ahora que ya estamos conectados con el servidor entramos en un bucle en
  el que leemos desde el socket todas las líneas que nos envía el servidor y
  las almacenamos en un fichero
  */
  if ((fp=fopen(argv[1],"w"))==NULL)
  {
    fprintf(stderr,"Error al abrir el fichero %s\n",argv[1]);
    exit(5);
  }
  while ((leidos=read(sock,datos,BUFFER_LECTURA))>0)
  {
    fwrite(datos,sizeof(char),leidos,fp);
  }
  fclose(fp);
  shutdown(sock,0);
  close(sock);
}
```

```
/*
Fichero servidor.c
   Ejemplo de servidor orientado a conexion que utiliza el esquema de
   direccionamiento Internet.

*/

#include <stdio.h>
#include "conexion.h"


void main(int argc,char *argv[])
{
  FILE *fp;
  int sock,fd;
  struct sockaddr_in servidor,cliente;
  int cliente_len,leidos;
  char datos[BUFFER_LECTURA];

  if (argc<2){
    fprintf(stderr,"Forma de uso: servidor fichero\n");
    exit(1);
  }
  if ((fp=fopen(argv[1],"r"))==NULL)
  {
    fprintf(stderr,"No se pudo abrir el fichero %s\n",argv[1]);
    exit(2);
  }
  sock=socket(AF_INET,SOCK_STREAM,0);
  if (sock<0)
  {
    fprintf(stderr,"No se pudo crear el socket.\n");
    exit(3);
  }
  servidor.sin_family=AF_INET;
  servidor.sin_addr.s_addr=htonl(INADDR_ANY);
  servidor.sin_port=htons(PUERTO_SERVIDOR);
  if (bind(sock,(struct sockaddr *) &servidor, sizeof(servidor))<0){
    fprintf(stderr,"Error vinculando el socket.\n");
    exit(4);
  }
  listen(sock,5);
  cliente_len=sizeof(cliente);
  fd=accept(sock,(struct sockaddr *) &cliente,&cliente_len);
  while (! feof(fp))
  {
    leidos=fread(datos,sizeof(char),BUFFER_LECTURA,fp);
    write(fd,datos,(size_t) leidos);
  }
  shutdown(fd,1);
  close(fd);
  close(sock);
  fclose(fp);
}
```

```
# Escuela Técnica Superior de Ingeniería Informática (Gijon)
# Sistemas de Computación (Curso 5)
# Aplicaciones Distribuidas en Unix con Sun RPC

#*************************************************
# Variables
#*************************************************
```

```
CC=cc
CFLAGS= -verbose
CLIENTE = cliente
SERVIDOR= servidor

todo : $(CLIENTE) $(SERVIDOR)

#************************************************
# Generacion del cliente y del servidor
#************************************************

$(CLIENTE) : cliente.c conexion.h
        $(CC) $(CFLAGS) -o $(CLIENTE) cliente.c

$(SERVIDOR) : servidor.c conexion.h
        $(CC) $(CFLAGS) -o $(SERVIDOR) servidor.c

#************************************************
# Utilidades
#************************************************

limpiaobjs :
        rm -f *.o
```

ANEXO B. CLIENTE/SERVIDOR NO ORIENTADOS A CONEXIÓN EN DOMINIO INTERNET

```
/*
Fichero conexion.h
   Fichero de cabecera necesario para el cliente y servidor
   no orientado a conexion que utilizan el esquema de
   direccionamiento Internet.

*/


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

#define BUFFER_LECTURA         80
#define PUERTO_SERVIDOR                4756
```

```
/*
Fichero cliente.c
   Ejemplo de cliente no orientado a conexion que utiliza el esquema
   de direccionamiento Internet.
*/


#include <stdio.h>
#include <string.h>
#include "conexion.h"


void main (int argc, char * argv[])
{
  FILE *fp;
  struct sockaddr_in servidor; /* La direccion del servidor se compone aquí */
  struct sockaddr_in cliente;   /* La direccion del cliente */
  struct hostent *host_info;    /* Información de la maquina del servidor */
  char datos[BUFFER_LECTURA];
  char *nombre_servidor;
  int sock,leidos;
  int server_len;

  if (argc<2)
  {
    fprintf(stderr,"Forma de uso: cliente fich [dir_servidor]\n");
    exit(1);
  }
  /* Obtenemos la direccion del servidor desde la línea de comandos */

  nombre_servidor=(argc>2) ? argv[2] : "localhost";

  /* Creamos el socket del cliente */

  sock = socket(AF_INET, SOCK_DGRAM, 0);
  if (sock<0)
  {
    fprintf(stderr,"Error creando el socket. \n");
    exit(2);
  }
  /* Asociamos una direccion local. Cualquier número de puerto valdrá.
     Este paso no es esencial para el cliente. Si no lo hacemos el sistema
     elegirá por nosotros un número de puerto arbitrario.
  */
  cliente.sin_family = AF_INET;
  cliente.sin_addr.s_addr=htonl(INADDR_ANY);
  cliente.sin_port=0; /* el 0 significa que el sistema escoja cualquier puerto libre */
  if (bind(sock,(struct sockaddr *) &cliente, sizeof(cliente))<0)
  {
    fprintf(stderr,"Fallo el bind.\n");
    exit(3);
  }
  host_info = gethostbyname(nombre_servidor);
  if (host_info==NULL)
```

```
  {
    fprintf(stderr,"%s: nombre de host desconocido: %s\n",argv[0],nombre_servidor);
    exit(4);
  }
  /* Establecer la direccion de conexion del socket del servidor para
     luego conectarnos
   */
  servidor.sin_family=AF_INET;
  memcpy((char *) &servidor.sin_addr.s_addr,host_info->h_addr,host_info->h_length);
  servidor.sin_port=htons(PUERTO_SERVIDOR);

  /* Ahora que ya estamos conectados con el servidor entramos en un bucle en
  el que leemos desde el socket todas las líneas que nos envía el servidor y
  las almacenamos en un fichero
  */
  if ((fp=fopen(argv[1],"r"))==NULL)
  {
    fprintf(stderr,"Error al abrir el fichero %s\n",argv[1]);
    exit(5);
  }
  while (! feof(fp))
  {
    leidos=fread(datos,sizeof(char),BUFFER_LECTURA,fp);
    server_len=sizeof(servidor);
    if (sendto(sock,datos,leidos,0,(struct sockaddr *) &servidor,server_len)!=leidos)
    {
      fprintf(stderr,"Error en el sendto.\n");
      exit(6);
    }
  }
  fclose(fp);
  shutdown(sock,1);
  close(sock);
}
```

```
/*
Fichero servidor.c
   Ejemplo de servidor no orientado a conexion que utiliza el
   esquema de direccionamiento Internet.
*/

#include <stdio.h>
#include <errno.h>
#include "conexion.h"


void main(int argc,char *argv[])
{
  FILE *fp;
  int sock;
  struct sockaddr_in servidor,cliente;
  int cliente_len,leidos,escritos;
  char datos[BUFFER_LECTURA];

  if (argc<2){
    fprintf(stderr,"Forma de uso: servidor fichero\n");
    exit(1);
  }
  if ((fp=fopen(argv[1],"w"))==NULL)
  {
    fprintf(stderr,"No se pudo abrir el fichero %s\n",argv[1]);
    exit(2);
  }
  sock=socket(AF_INET,SOCK_DGRAM,0);
  if (sock<0)
  {
    fprintf(stderr,"No se pudo crear el socket.\n");
    exit(3);
  }
  servidor.sin_family=AF_INET;
  servidor.sin_addr.s_addr=htonl(INADDR_ANY);
  servidor.sin_port=htons(PUERTO_SERVIDOR);
  if (bind(sock,(struct sockaddr *) &servidor, sizeof(servidor))<0){
    fprintf(stderr,"Error vinculando el socket.\n");
    exit(4);
  }
  cliente_len=sizeof(cliente);
  do
  {
    cliente_len=sizeof(cliente);
    leidos=recvfrom(sock,datos,BUFFER_LECTURA,0,(struct sockaddr *)&cliente,&cliente_len);
    if (leidos<0)
```

```
    {
      fprintf(stderr,"Error %d en el recvfrom.\n",errno);
      exit(5);
    }
    escritos=fwrite(datos,sizeof(char),leidos,fp);
    if (escritos!=leidos)
    {
      fprintf(stderr,"Error de escritura en el fichero.\n");
      exit(6);
    }
  }
  while (leidos==BUFFER_LECTURA);
  shutdown(sock,0);
  close(sock);
  fclose(fp);
}
```

```
# Escuela Técnica Superior de Ingeniería Informática (Gijon)
# Sistemas de Computación (Curso 5)
# Aplicaciones Distribuidas en Unix con Sun RPC

#*************************************************
# Variables
#*************************************************

CC=cc
CFLAGS= -verbose
CLIENTE = cliente
SERVIDOR= servidor

todo : $(CLIENTE) $(SERVIDOR)

#*************************************************
# Generacion del cliente y del servidor
#*************************************************

$(CLIENTE) : cliente.c conexion.h
        $(CC) $(CFLAGS) -o $(CLIENTE) cliente.c

$(SERVIDOR) : servidor.c conexion.h
        $(CC) $(CFLAGS) -o $(SERVIDOR) servidor.c

#*************************************************
# Utilidades
#*************************************************

limpiaobjs :
        rm -f *.o
```

ANEXO C. CLIENTE/SERVIDOR ORIENTADOS A CONEXIÓN EN DOMINIO UNIX

```
/*
Fichero conexión.h

   Fichero de cabecera para el cliente y servidor orientados a
   conexion que utilizan el esquema de direccionamiento Unix.
*/

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

#define BUFFER_LECTURA         80
#define SOCKET_UNIX            "./unixmailbox"
```

```
/*
Fichero cliente.c
   Ejemplo de cliente orientado a conexion que utiliza el esquema de
   direccionamiento Unix.
*/

#include <stdio.h>
#include <string.h>
#include "conexion.h"


void main (int argc, char * argv[])
{
  FILE *fp;
  struct sockaddr_un servidor; /* La direccion del servidor se compone aquí */

  char datos[BUFFER_LECTURA];
  int sock,leidos;
  int len_servidor,cliente_len;

  if (argc!=2)
  {
    fprintf(stderr,"Forma de uso: cliente fich \n");
    exit(1);
  }

  /* Establecer la direccion de conexion del socket del servidor para
     luego conectarnos
   */

  bzero((char *) &servidor,sizeof(servidor));
  servidor.sun_family=AF_UNIX;
  strcpy(servidor.sun_path,SOCKET_UNIX);
  len_servidor=strlen(servidor.sun_path)+sizeof(servidor.sun_family);


  /* Creamos el socket del cliente */

  sock = socket(AF_UNIX, SOCK_STREAM, 0);
  if (sock<0)
  {
    fprintf(stderr,"Error creando el socket. \n");
    exit(2);
  }

  /* Nos conectamos al servidor */

  if (connect(sock, (struct sockaddr *) &servidor, len_servidor)<0)
  {
    fprintf(stderr,"Error conectando con el servidor\n");
    exit(3);
  }
  /* Ahora que ya estamos conectados con el servidor entramos en un bucle en
  el que leemos desde el socket todas las líneas que nos envía el servidor y
  las almacenamos en un fichero
  */
  if ((fp=fopen(argv[1],"w"))==NULL)
  {
```

```
    fprintf(stderr,"Error al abrir el fichero %s\n",argv[1]);
    exit(4);
  }
  while ((leidos=read(sock,datos,BUFFER_LECTURA))>0)
  {
    fwrite(datos,sizeof(char),leidos,fp);
  }
  fclose(fp);
  shutdown(sock,0);
  close(sock);
}
```

```
/* Fichero servidor.c
   Ejemplo de servidor orientado a conexion que utiliza el esquema de
   direccionamiento Unix
*/

#include <stdio.h>
#include "conexion.h"

void main(int argc,char *argv[])
{
  FILE *fp;
  int sock,fd;
  struct sockaddr_un servidor,cliente;
  int servidor_len,cliente_len,leidos;
  char datos[BUFFER_LECTURA];

  if (argc<2){
    fprintf(stderr,"Forma de uso: servidor fichero\n");
    exit(1);
  }
  if ((fp=fopen(argv[1],"r"))==NULL)
  {
    fprintf(stderr,"No se pudo abrir el fichero %s\n",argv[1]);
    exit(2);
  }
  sock=socket(AF_UNIX,SOCK_STREAM,0);
  if (sock<0)
  {
    fprintf(stderr,"No se pudo crear el socket.\n");
    exit(3);
  }
  bzero((char *) &servidor,sizeof(servidor));
  servidor.sun_family=AF_UNIX;
  strcpy(servidor.sun_path,SOCKET_UNIX);
  servidor_len=strlen(servidor.sun_path)+sizeof(servidor.sun_family);
  if (bind(sock,(struct sockaddr *) &servidor, servidor_len)<0){
    fprintf(stderr,"Error vinculando el socket.\n");
    exit(4);
  }
  listen(sock,5);
  cliente_len=sizeof(cliente);
  fd=accept(sock,(struct sockaddr *) &cliente,&cliente_len);
  while (! feof(fp))
  {
    leidos=fread(datos,sizeof(char),BUFFER_LECTURA,fp);
    write(fd,datos,(size_t) leidos);
  }
  shutdown(fd,1);
  close(fd);
  close(sock);
  fclose(fp);

}
```

```
# Escuela Técnica Superior de Ingeniería Informática (Gijon)
# Sistemas de Computación (Curso 5)
# Aplicaciones Distribuidas en Unix con Sun RPC

#************************************************
# Variables
#************************************************

CC=cc
CFLAGS= -verbose
CLIENTE = cliente
SERVIDOR= servidor

todo : $(CLIENTE) $(SERVIDOR)

#************************************************
# Generacion del cliente y del servidor
#************************************************

$(CLIENTE) : cliente.c conexion.h
        $(CC) $(CFLAGS) -o $(CLIENTE) cliente.c

$(SERVIDOR) : servidor.c conexion.h
        $(CC) $(CFLAGS) -o $(SERVIDOR) servidor.c

#************************************************
# Utilidades
#************************************************

limpiaobjs :
        rm -f *.o
```

BIBLIOGRAFÍA

Unix. Distributed Programming

Autor: Chris Brown
Editorial: Prentice-Hall
ISBN: 0-13-075896-5

Network Programming Interfaces

Editorial: Unix Press (Prentice-Hall)
ISBN: 0-13-017641-9

Unix Network Programming

Autor: W. Richard Stevens
Editorial: Prentice-Hall
ISBN: 0-13-949876-1

Unix. Manual Pages