Contenido

10. Comunicaciones en Red

10.1 Mecanismos IPC del Sistema 4.3BSD

Este tema está destinado a dar una visión sobre el uso de la interfaz de comunicación entre procesos del sistema 4.3 BSD. Esta interfaz permitirá comunicar procesos que se estén ejecutando

bajo el control de una misma computadora o bajo el control de computadoras distintas. En el segundo caso, es necesario que entren en juego redes de comunicación entre computadoras.

Siempre que se habla de una red de computadoras hay que distinguir entre protocolos y servicios de red. En realidad, un estudio exhaustivo de las redes debe empezar por el modelo de referencia OSI. Este es un modelo que estructura en siete capas los diferentes elementos de una serie de servicios a la capa superior y utiliza los servicios que brinda la capa inferior. Los servicios de una capa pueden verse como una interfaz de comunicación con la misma. Los protocolos, por su parte, definen la forma que van a tener las tramas de bits que se intercambian las distintas capas y cómo se va a llevar a cabo esa comunicación. El objetivo que se persigue al separar los protocolos de servicios es aislar los aspectos tecnológicos de los aspectos de uso de una red. Así, se procura definir servicios lo menos cambiantes posibles con objeto de que los usuarios no tengan que estar modificando sus aplicaciones continuamente. Por otro lado, las mejores en las tecnologías de las redes van a repercutir en diseño de protocolos que garanticen una transmisión más fiable, segura y rápida. Esto va a complicar los protocolos, pero no los servicios.

La interfaz con la red que ofrece el sistema 4.3BSD corresponde al nivel 4 (capa de transporte) del modelo OSI. Así, dos programas que se comuniquen mediante conectores van a despreocuparse totalmente de aspectos tales como:

- Canal físico de comunicación, tipo de transmisión (analógica o digital), tipo de modulación (PSK, ASK, …). Eso corresponde a la capa física.
- Forma de codificar las señales para disminuir la probabilidad de error en la transmisión. Corresponde a la capa de enlace.
- Nodos de la red por lo que tienen que pasar las tramas de bits y gestión de las redes involucradas. Corresponde a la capa de red.
- Formar paquetes a partir de la información a transmitir y buscar una ruta que una la computadora origen con el destino. Corresponde a la capa de transporte.

10.1.1 Protocolos y Conexiones

No vamos a adentrarnos en la descripción de las capas OSI, ni a nivel de servicios ni protocolos. Sin embargo, no dejaremos pasar la oportunidad de aclarar algunos de los términos o acrónimos que aparecerán en puntos posteriores.

Lo primero que sería bueno dejar claro es que la comunicación mediante conectadores es una interfaz con la capa de transporte (nivel 4) de la jerarquía OSI. Los protocolos de los niveles inferiores no los vamos a estudiar; en primer lugar, porque su exposición sería demasiado larga, y en segundo lugar, porque no es necesario. La filosofía de la división por capas de un sistema es encapsular, dentro de cada una de ellas, detalles que conciernen sólo a la capa, y presentársela al usuario como una caja negra con unas determinadas entradas y salidas, de tal forma que el usuario pueda trabajar con ella sin necesidad de conocer sus detalles de implementación.

La interfaz de acceso a la capa de transporte del sistema 4.3BSD no está totalmente aislada de las capas inferiores, por lo que a la hora de trabajar con conectores es necesario conocer algunos detalles de esas capas. En concreto, a la hora de establecer una conexión, es necesario conocer la familia o dominio de la conexión y el tipo de conexión.

- Una familia agrupa todos aquellos conectores que comparten las características comunes, tales como protocolos, convenios para formar direcciones de red, convenios para formar nombres, etc. Más adelante mencionaremos algunas de las familias más empleadas.
- El tipo de conexión indica el tipo de circuito que se va a establecer entre los dos procesos que se están comunicando. El circuito puede ser virtual (orientado a conexión) o datagrama (no orientado a conexión). Para establecer un circuito virtual, se realiza una búsqueda de enlaces libres que unas los dos ordenadores a conectar. Es algo parecido a lo que hace la red telefónica conmutada para establecer una conexión entre dos teléfonos. Una vez establecida la conexión, se puede proceder al envío secuencial de los datos, ya que la conexión es permanente. Por el contrario, los datagramas no trabajan con conexiones permanentes. La transmisión por datagramas es a nivel de paquetes, donde cada paquete puede seguir una ruta distinta y no se garantiza una recepción secuencial de la información.

10.1.2 Direcciones de Red

A la hora de referirse a un nodo de la red, cada protocolo implementa un mecanismo de direccionamiento. La dirección distingue de forma inequívoca a cada nodo o computadora y es utilizada para enrutar los datos desde el nodo origen al nodo destino. La forma de construir direcciones depende de los protocolos que se empleen en la capa de transporte y de red, por lo que no vamos a detenernos en ello. Sin embargo, hay muchas llamadas al sistema 4.3BSD que necesitan un apuntador a una estructura de dirección de conector para trabajar. Esta estructura se define en el archivo cabecera `<sys/socket.h>` y su forma es la siguiente:

```
struct sockaddr {
       u_short sa_family;  /* Familia de conectores. Se emplean las
                              constantes de la forma "AF_xxx" que
                              veremos más adelante. */
       char sa_data[14];         /* 14 bytes que contiene la dirección. Su
                              significado depende de la familia de
                              conectores que se esté empleando. */
};
```

El contenido de los 14 bytes del campo `sa_data` depende de la familia de conectores que se esté usando. Así, si usamos una familia que emplea conectores Internet, la forma de las direcciones de red será la definida en el fichero de cabecera `<netinet/in.h>`.

```
struct in_addr {
       u_long s_addr;      /* 32 bits que contiene la identificación de la
                              red y del nodo. */
};

struct sockaddr_in {
       short sin_family;               /* AF_INET. */
       u_short sin_port;               /* 16 bits con el número de puerto. */
       struct in_addr sin_addr;  /* 32 bits con la identificación de la
                                       red y del nodo. */
       char sin_zero[8];               /* 8 bytes no usados. */
};
```

Podemos ver que en la dirección Internetel campo `sin_family` es equivalente al campos `sa_family` de la dirección genérica y que los campos `sin_port`, `sin_addr` y `sin_zero` cumplen la misma función que el campo `sa_addr`.

La familia de conectores conocida como “dominio UNIX” emplea direcciones con la forma definida en `<sys/un.h>`:

```
struct sockaddr_un {
      short sun_family;           /* AF_UNIX. */
      char sun_path[108];   /* ruta. */
};
```

Estas direcciones corresponden en realidad con rutas de archivos y su longitud (110 bytes) es superior a los 16 bytes que de forma estándar tiene las direcciones del resto de las familias. Esto es posible debido a que esta familia tiene conectores se utiliza para comunicar procesos que se están ejecutando bajo el control de una misma computadora, por lo que no necesitan hacer acceso a la red.

10.1.3 Modelo Cliente-Servidor

El modelo cliente-servidor es el modelo estándar de ejecución de aplicaciones en una red.

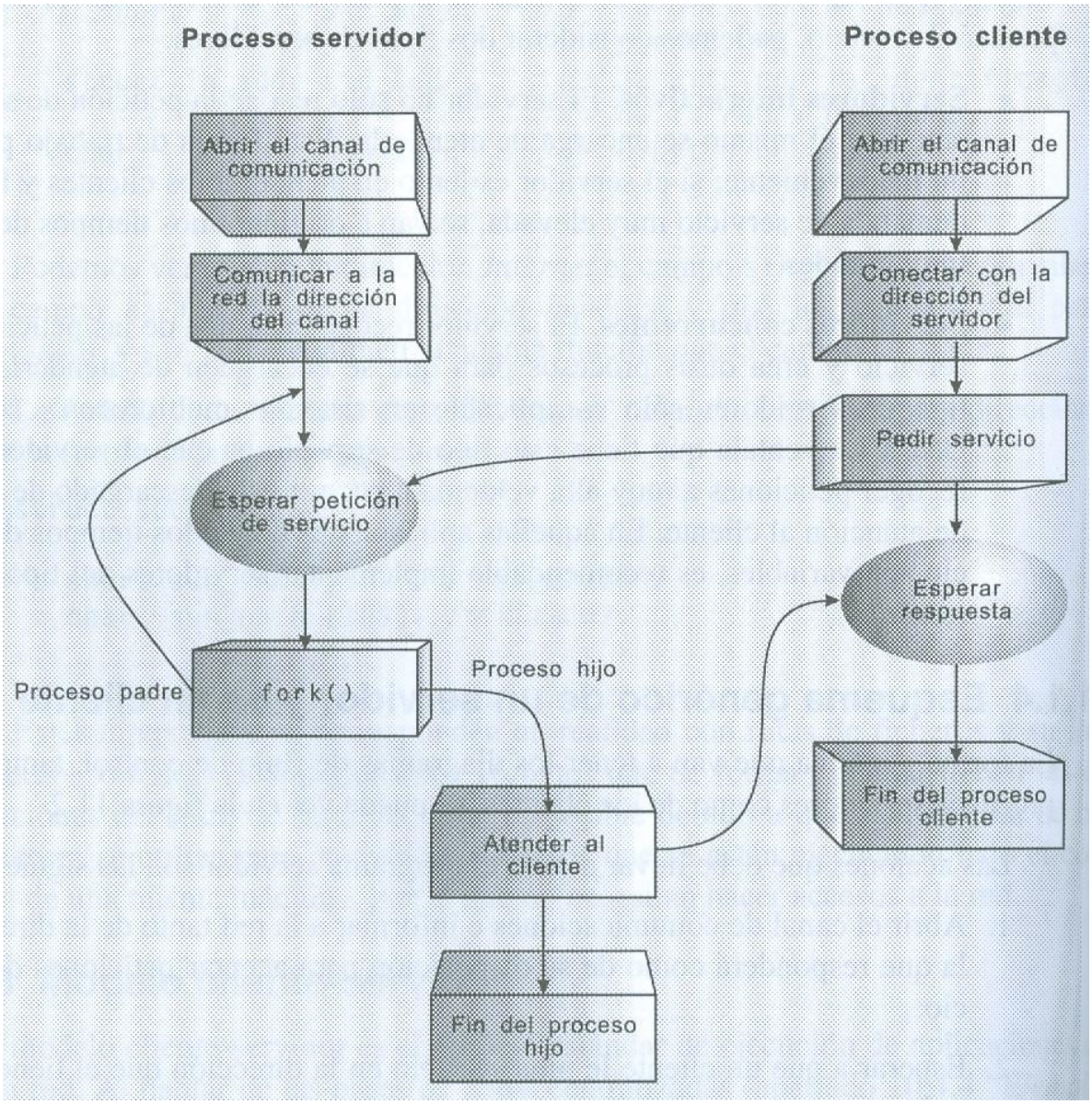
Un servidor es un proceso que se está ejecutado en un nodo de la red y que gestiona el acceso a un determinado recurso. Un cliente es un proceso que se ejecuta en el mismo modo o en diferente nodo y que realizar peticiones al servidor. Las peticiones están originadas por la necesidad de acceder al recurso que gestiona el servidor.

El servidor está continuamente esperando peticiones de servicio. Cuando se produce una petición, el servidor despierta y atiende al cliente. Cuando el servicio concluye, el servidor vuelve al estado de espera. De acuerdo con la forma de prestar el servicio, podemos considerar dos tipos de servidores:

- Servidores interactivos. El servidor no sólo recoge la petición de servicio, sino que él mismo se encarga de atenderla. Esta forma de trabajo presenta un inconveniente: si el servidor es lento en atender a los clientes y hay una demanda de servicio muy elevada, se van a originar unos tiempos de espera muy grandes.
- Servidores concurrentes. El servidor recoge cada una de las peticiones de servicio y crea otro proceso para que se encarguen de atenderlas. Este tipo de servidores sólo es aplicable en sistemas multiproceso, como es UNIX. La ventaja que tiene este tipo de servicio es que el servidor puede recoger peticiones a muy alta velocidad, porque está descargado de la tarea de atención al cliente. En aquellas aplicaciones donde los tiempos de servicios son variables, es recomendable implementar servidores del tipo concurrente.

10.1.4 Esquema genérico de un servidor y de un cliente

La forma genérica que van a tener los diagramas de flujo de control, tanto de los procesos servidores como de los clientes, se puede ver en la siguiente figura.



Las acciones que debe llevar a cabo el programa servidor son las siguientes:

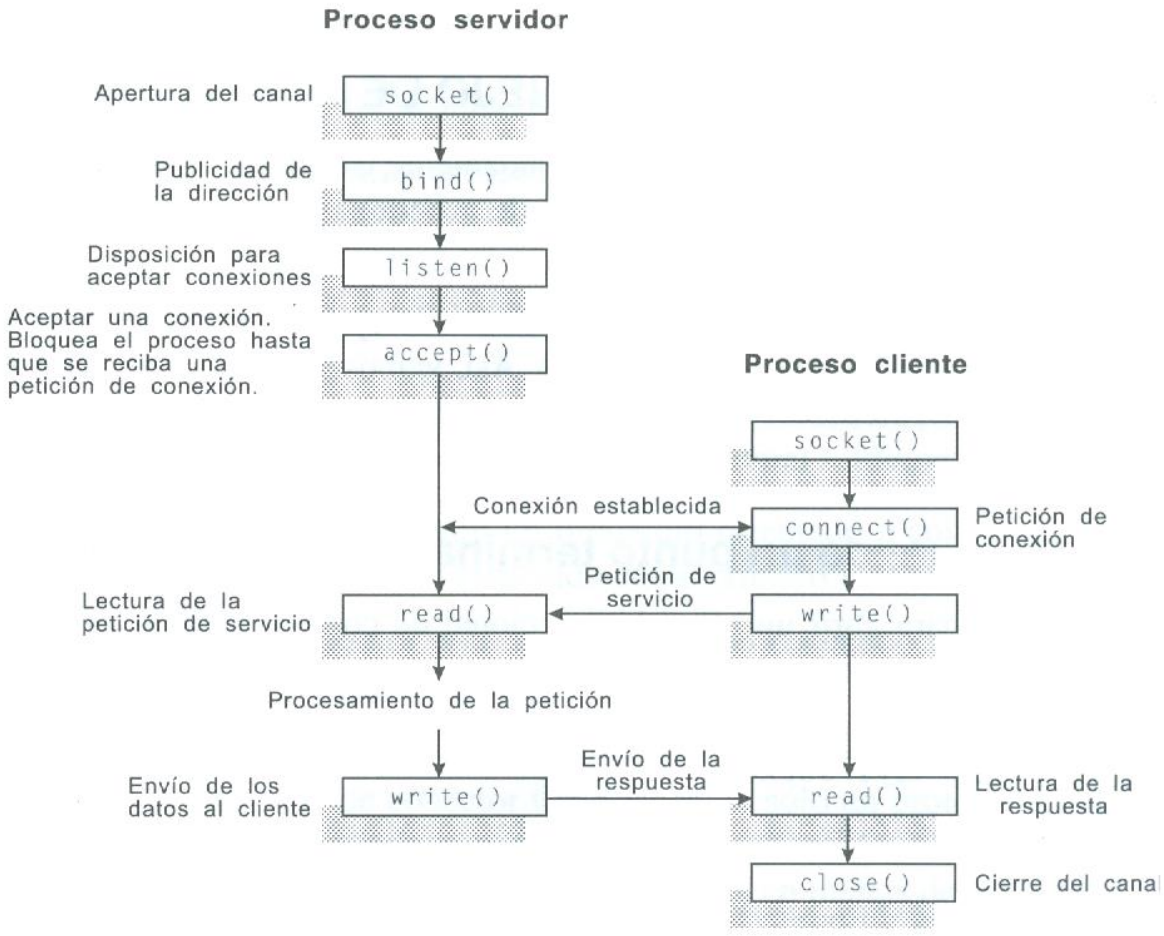
1. Abrir el canal de comunicaciones e informar a la red tanto de la dirección a la que responderá como de su disposición para aceptar peticiones de servicio.
2. Esperar a que un cliente le pida servicio en la dirección que él tiene declarada.
3. Cuando recibe una petición de servicio, si es un servidor interactivo, atenderá al cliente. Los servidores interactivos se suelen implementar cuando la respuesta que necesita el cliente es sencilla e implica poco tiempo de proceso.

   Si el servidor concurrente, creará un proceso mediante `fork` para que le dé servicio al cliente.
4. Volver al punto 2 para esperar nuevas peticiones de servicio.

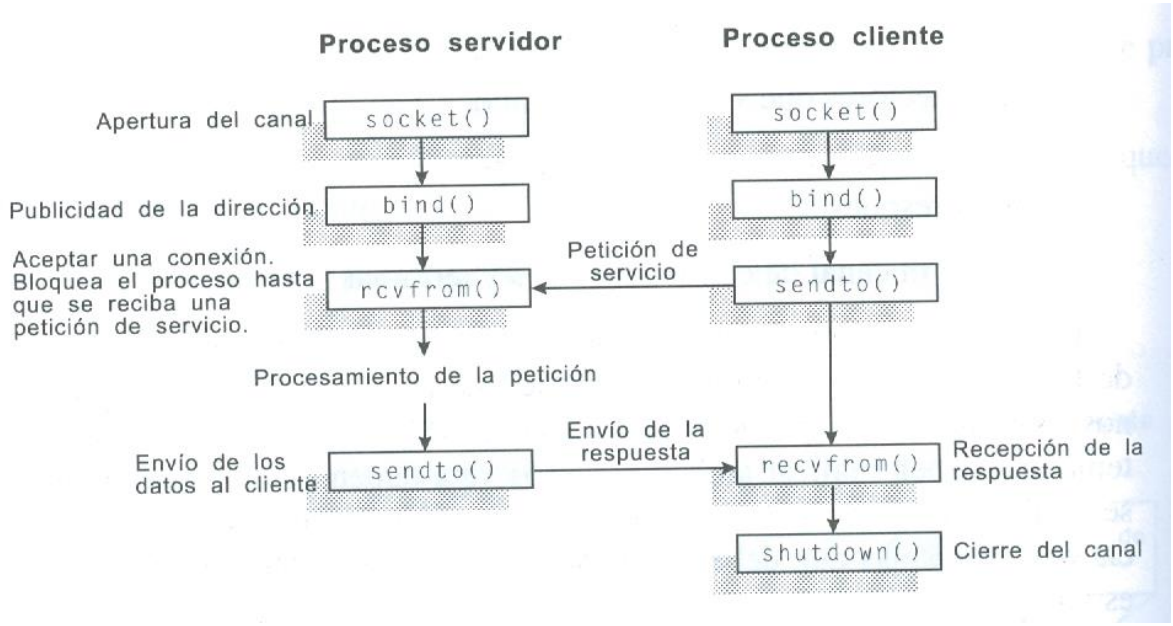
El programa cliente, por su parte, llevará a cabo a las siguientes acciones:

1. Abrir el canal de comunicaciones y conectarse a la dirección de red atendida por el servidor. Naturalmente, esta dirección debe ser conocida por el cliente y responderá al esquema de generación de direcciones de la familia de conectores que se esté empleando.
2. Enviar al servidor un mensaje de petición de servicio y esperar hasta recibir la respuesta.
3. Cerrar el canal de comunicaciones y terminar la ejecución.

Los esquemas anteriores son aplicables de forma general, independientemente de la interfaz con la capa de transporte que empleen el servidor y el cliente. Si nos fijamos en los servicios del sistema 4.3BSD, la secuencia de llamadas al sistema que deben realizar ambos procesos va a depender del tipo de conexión que se establezca entre ellos. En la siguiente figura podemos ver las llamadas y la secuencia de estas que deben realizar el servidor y el cliente cuando la conexión es virtual.



En la siguiente figura vemos la secuencia de llamadas necesarias para comunicar servidor y cliente cuando no hay conexión y el envío de datos es mediante datagramas.



10.2 Llamadas para el manejo de Conectores

A continuación, vamos a explicar detalladamente cada una de las llamadas al sistema que intervienen en la codificación de programas servidores y clientes que se comunican mediante conectores.

Hay que hacer notar que la interfaz de Berkeley se ha modificado para sea muy similar a la del sistema de archivos. Así, vamos a trabajar con descriptores de conectores y de archivos, y algunas de las llamadas estudiadas para manejar archivos estarán disponibles también para conectores.

10.2.1 Apertura de un punto Terminal de un Canal – socket

La llamada para abrir un canal bidireccional de comunicaciones es socket y se declara como sigue:

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int af, int type, int protocol);
```

`socket` crea un punto terminal para conectarse a un canal y devuelve un descriptor. El descriptor del conector devuelto se usará en llamadas posteriores a funciones de la interfaz.

`af` (address family) especifica la familia de conectores o familia de direcciones que se desea emplear. Las distintas están definidas en el archivo de cabecera `<sys/socket.h>` y dependerán del fabricante del sistema y de la configuración del hardware. Las dos familias siguientes suelen estar presentes en todos los sistemas:

Valores	Significado
AF_UNIX	Protocolos internos UNIX. Es la familia de conectores empleada para comunicar procesos que se ejecutan en una misma computadora. Esta familia no requiere que esté presente un hardware especial de red, puesto que en realidad no realiza acceso a ninguna red.
AF_INET	Protocolos Internet. Es la familia de conectores que se comunican mediante protocolos, tales como TCP (Transmission Control Protocol), desarrollado por la Universidad de California en Berkeley para DARPA (Defense Advance Research Projects Agency) o UDP (User Datagram Protocol).
AF_CCITT	Norma X.25 del CCITT.
AF_NS	Protocolo NS de Xerox.

Más adelante vamos a trabajar sólo con las familias `AF_UNIX` y `AF_INET`.

El argumento `type` indica la semántica de la comunicación para el conector. Puede ser:

Valores	Significado
SOCK_STREAM	Conector con un protocolo orientado a conexión. Esto es lo que hemos estudiado como circuito virtual.
SOCK_DGRAM	Conector con un protocolo no orientado a conexión o datagrama.

`protocol` especifica el protocolo particular que se va a usar con el conector. Normalmente, cada tipo de conector tiene asociado sólo un protocolo, pero si hubiera más de uno, se especificaría mediante este argumento. `protocol` puede valer 0, en cuyo caso la elección del protocolo se deja en manos del sistema.

Si la llamada se ejecuta satisfactoriamente, devolverá un descriptor de archivo válido. En caso contrario, devolverá -1 y en `errno` estará codificado el error producido.

10.2.2 Nombre de un conector – bind

La llamada `bind` se utiliza para unir un conector con una dirección de red determinada. Su declaración es la siguiente:

```
#include <sys/socket.h>
/* Solo para la familia AF_UNIX. */
#include <sys/un.h>
/* Sólo para la familia AF_INET. */
#include <sys/netinet.h>
int bind(int sfd, const void *addr, int addrlen);
```

Cuando se crea un conector con la llamada `socket`, se le asigna una familia de direcciones, pero no una dirección particular. `bind` hace que el conector cuyo descriptor es `sfd` se una a la dirección del conector especificada en la estructura apuntada por `addr`. `addrlen` indica el tamaño de la dirección.

Hay que indicar que el formato de la dirección depende de la familia del conector, por lo que habrá que utilizar la estructura adecuada. Para la familia `AF_UNIX` se debe usar la estructura `struct sockaddr_un` y para la familia `AF_INET` la estructura `sockaddr_in`. El valor de `addrlen` se puede calcular con el operador `sizeof`.

La semántica utilizada en la unión del conector con la dirección depende de la familia. Así, por ejemplo, cuando se une un conector `AF_UNIX` a una ruta (por ejemplo, `/tmp/misocket`), se crea un archivo con ese nombre. Cuando se cierra el conector, el archivo sigue existiendo hasta que se dé la orden de borrarlo. Si se une un conector `AF_INET` a una dirección, el campo `sin_port` puede contener el número de puerto ó 0. Si valor 0, el sistema le asigna un número de puerto libre.

Si la llamada funciona correctamente, devuelve el valor 0. En caso contrario devuelve -1.

10.2.3 Disponibilidad para recibir peticiones de servicio – listen

Cuando se abre un conector orientado a conexión, el programa servidor indica que está disponible para recibir peticiones de conexión mediante la llamada a `listen`, que se declara como sigue:

```
#include <sys/socket.h>
int listen(int sfd, int backlog);
```

La llamada a `listen` la suele ejecutar el proceso servidor después de las llamadas a `socket` y `bind`. `listen` habilita una cola asociada al conector descrito por `sfd`. Esta cola se va a encargar de alojar peticiones de conexión procedentes de los procesos de cliente. La longitud de la cola es la especificada en el argumento `backlog`. Para que la llamada a listen tenga sentido, el conector debe ser del tipo `SOCK_STREAM` (conector orientado a conexión).

La cola de conexiones es importante para los servidores de tipo interactivo, porque mientras están atendiendo a un cliente pueden llegar peticiones procedentes de otros clientes. En los de tipo concurrente, el único instante en el que el servidor no atiende la recepción de peticiones es el que dedica a la llamada `fork`. En estos casos también es importante la cola de conexiones.

Si la llamada a listen funciona correctamente, devuelve el valor 0. En caso contrario devuelve -1.

10.2.4 Petición de Conexión – connect

Para que un proceso cliente inicie una conexión con un servidor a través de un conector, es necesario que haga una llamada a `connect`. Esta función se declara de la siguiente forma:

```
#include <sys/socket.h>
/* Solo para la familia AF_UNIX. */
#include <sys/un.h>
/* Sólo para la familia AF_INET. */
#include <sys/netinet.h>
int connect(int sfd, const void *addr, int addrlen);
```

`sfd` es el descriptor del conector que da acceso al canal y `addr` es un apuntador a una estructura que contiene la dirección del conector remoto al que queremos conectarnos. `addrlen` es el

tamaño en bytes de la dirección. La estructura de la dirección dependerá de la familia de conectores con la que estemos trabajando.

Si el conector es del tipo `SOCK_DGRAM`, `connect` especifica la dirección del conector remoto al que se le van a enviar los mensajes, pero no se conecta con él. La llamada devuelve el control inmediatamente. Además, a través del conector sólo se podrán recibir mensajes procedentes de la dirección especificada.

Si el conector es del tipo `SOCK_STREAM`, `connect` intenta contactar con la computadora remota con objeto de realiza una conexión entre el conector remoto y el conector local especificado por `sfd`. La llamada normalmente permanece bloqueada hasta que la conexión se completa. Por el contrario, si la conexión no se puede realizar de forma inmediata, pero el conector tiene activo el modo de acceso `O_NDELAY` (activado a través de una llamada a `fcntl`), la llamada a `connect` devuelve el control inmediatamente indicando que se ha producido un error de conexión.

Si la conexión se realiza satisfactoriamente, la llamada devuelve el valor 0; en caso contrario, devolverá -1 y en `errno` estará el código del error producido.

10.2.5 Aceptación de una conexión – accept

Los procesos servidores van a leer peticiones de servicio mediante la llamada `accept`. La declaración de esta llamada se muestra a continuación:

```
#include <sys/socket.h>
int accept(int sfd, void *addr, int *addrlen);
```

Esta llamada se usa con conectores orientados a conexión, como el tipo `SOCK_STREAM`. El argumente `sfd` es un descriptor del conector creado por una llamada previa socket y unido a una dirección mediante `bind`. `accept` extrae la primera petición de conexión que hay en la cola de peticiones pendientes creada con una llamada previa a `listen`. Una vez extraída la petición de la conexión, `accept` crea un nuevo conector con las mismas propiedades que `sfd` y reserva un nuevo descriptor de archivo (`nsfd`) para él.

Si no hay peticiones de conexión pendientes y el conector no tiene activo el modo de acceso no bloqueante (`O_NDELAY`), `accept` permanece bloqueada hasta que reciba una nueva petición de conexión. Si el nodo no bloqueante está activado, `accept` devolverá el control inmediatamente e indicará que se ha producido un error.

El conector original (`sfd`) permanece abierto y puede aceptar nuevas conexiones; sin embargo, el conector recién creado (`nsfd`) no puede usarse para aceptar más conexiones. La llamada `select` puede usarse para determinar si un conector tiene pendiente alguna conexión.

El argumento `addr` debe apuntar a una estructura local con la dirección del conector. La llamada `accept` rellenará esa estructura con la dirección del conector remoto que pide la conexión. El formato de la estructura de dirección dependerá del tipo de conector que se esté manejando. El argumento `addrlen` debe ser un apuntador a `int`. Inicialmente, debe contener el tamaño en bytes de la estructura de dirección. La función sobrescribirá en `addrlen` el tamaño real de la dirección leída de la cola de conexiones.

Si la llamada funciona correctamente, devolverá un número entero no negativo que se debe interpretar como un descriptor del conector aceptado. En caso de error, devolverá el valor -1.

10.2.6 Lectura o Recepción de Mensajes de un Conector

Una vez que el canal de comunicación entre los procesos servidor y cliente está correctamente inicializado y ambos procesos disponen de un conector con el canal, contamos con 5 llamadas al sistema para leer datos/mensajes de un conector y otras 5 llamadas para escribir datos/mensajes en él.

Las llamadas para leer datos de un conector son: `read`, `readv`, `recv`, `recvfrom` y `recvmsg`. `read` no la vemos a describir, porque ya vimos cuál era su interfaz al manejar el sistema de archivos. De cara al manejo de conectores, la llamada `read` se comporta de la misma forma, pero con la salvedad de que el descriptor de archivos que utiliza es en realidad un descriptor del conector. Esta es una de las ventajas que tiene la interfaz del 4.3BSD, que las llamadas para el manejo de archivos siguen siendo válidas para el manejo de conectores.

La llamada `readv` es una generalización de `read` y puede utilizarse para leer datos de cualquier descriptor, ya sea archivo o conector. Su declaración es la siguiente:

```
#include <sys/uio.h>
ssize_t readv(int filedes, const struct iovec *iov, size_t iovcnt);
```

`readv` va a leer datos procedentes del archivo descriptor por `filedes` y los va a distribuir entre las distintas zonas de memoria intermedia especificadas por el arreglo `iov`. El total de elemento de `iov` viene indicado por el argumento `iovcnt`. Así, `iov` va a describir un total de `iovcnt` memorias intermedias, que son: `iov[0]`, `iov[1]`, `iov[2]`, …, `iov[iovcnt – 1]`.

Cada elemento del arreglo `iov` es del tipo `struct iovec`, que se define con los siguientes campos:

```
struct iovec {
      caddr_t iov_base; /* Dirección inicial de la memoria intermedia. */
      int iov_len;      /* Tamaño, en bytes, de la memoria intermadia. */
};
```

`readv` irá rellenando las distintas memorias intermedias, empezando por la primera, a medida que lee datos del archivo. Si llega al final del archivo o completa los bloques de memoria, termina su ejecución, devolviendo el número total de bytes leídos.

Si bien `read` y `readv` puede leer de cualquier descriptor (descriptor de archivos o de conector), las llamadas `recv`, `recvfrom` y `recvmsg` sólo pueden leer de un conector. Las declaraciones de estas funciones quedan como sigue:

```
#include <sys/socket.h>
int recv(int sfd, void *buf, int len, int flags);
int recvfrom(int sfd, void *buf, int len, int flags, void *from,
                  int fromlen);
int recvmsg(int sfd, struct msghdr msg[], int flags);
```

En todas las llamadas, `sfd` es el descriptor del conector del qué se leen los datos, `buf` es el apuntador a la memoria intermedia donde se van a escribir los datos leídos y `len` es el número máximo de bytes que se pueden escribir en la memoria referencia por `buf`.

En los conectores del tipo `SOCK_STREAM`, las llamadas pueden usarse sólo después de haber establecido una conexión previa vía la llamada a `connect`. En los conectores no orientados a conexión, `SOCK_DGRAM`, las llamadas pueden usarse tanto si se ha establecido una conexión como si no.

`recvfrom` trabaja de la misma forma que `recv`, pero con la ventaja de que puede devolver la dirección del conector desde el que se enviaron los datos. En el caso de conectores datagrama en los que se ha establecido una conexión, la dirección devuelta por `recvfrom` siempre va a ser la misma. Para conectores del tipo `stream`, `recvfrom` devuelve el mismo tipo de información que `recv` y no va a devolver la dirección del conector origen.

Si `from` es un apuntador distinto a `NULL`, la dirección del conector origen de los datos se escribirá en la estructura apuntado por `from`. `fromlen` es un parámetro de entrada/salida. Al llamar a la función, debe contener el tamaño de la estructura apuntada por `from`, y a la salida de la función, contendrá el tamaño real de la dirección leída.

Para conectores basados en paso de mensajes, los conectores datagrama, cada mensaje se debe leer en su totalidad en una sola operación. Si el mensaje es demasiado largo para caber en la memoria intermedia indica, será truncado. En los conectores no basados en el paso de mensajes, caso de conectores `stream`, los datos se devuelven a medida que están disponibles y no se maneja para nada el concepto de tamaño de un mensaje.

La llamada `recvmsg` es una generalización de las dos anteriores. La diferencia entre ellas es que los datos leídos pueden distribuirse entre varios bloques de memoria. El argumento `msg` es un arreglo de cabeceras de mensaje, cada una de las cuales tiene la siguiente estructura:

```
struct msghdr {
      caddr_t msg_name;             /* Dirección, opcional. */
      int msg_namelen;              /* Tamaño del campo msg_name. */
      struct iovec *msg_iov;        /* Arreglo de bloques de memoria sobre
                                       los que distribuir los datos
                                       leídos. */
      int msg_iovlen;               /* Número de elementos del arreglo. */
      caddr_t msg_accrights;        /* Derechos de acceso. */
      int msg_accrightslen;         /* Tamaño de msg_accrights. */
};
```

Esta estructura se emplea tanto para leer mensajes de un conector como para escribirlos en él. Los campos `msg_name` y `msg_namelen` se usan para conectores no orientados a conexión cuando nos interesa conocer la dirección del conector origen o destino del mensaje. Si no necesitamos conocer esa dirección, el campo `msg_name` puede ser un apuntador a `NULL`. `msg_iov` es el arreglo de bloques de memoria sobre los que distribuir los datos leídos, o de los que recoger los datos que se van a enviar. `msg_accrights` es la memoria intermedia sobre la que recoger los derechos de acceso enviados junto con los mensajes. Este campo sólo se utiliza con los conectores

de la familia AF_UNIX cuando los procesos se intercambian descriptores de archivo. El campo msg_accrights puede ser un apuntador a NULL, no se leen del conector los derechos de acceso.

Para las tres llamadas, el argumento flags tiene el mismo significado, y sus valores posibles son el valor 0 o una combinación de los bits siguientes:

Valores	Significado
MSG_PEEK	Cualquier dato leído del conector es tratado como si la lectura no se hubiese llevado a cabo, por lo que la siguiente operación de la lectura va a leer los mismos datos. Esta opción se puede usar para ver si hay datos disponibles en el conector.
MSG_OOB	Se utiliza para leer datos que van fuera de banda. Estos son datos a los que se les da un carácter de urgente, por lo que tienen preferencia en el envío y recepción.

En los conectores de la familia AF_UNIX no pueden estar activos ninguno de los bits anteriores.

Las tres funciones estudiadas devuelven el total de bytes leídos del conector.

10.2.7 Estructura o envío de mensajes a un conector

Las llamadas para escribir datos en un conector son: write, writev, send, sendto y sendmsg. write no la va vamos a describir, porque ya vimos es su interfaz al manejar el sistema de archivos. De cara al manejo de conectores, la llamada write se comporta de la misma manera, pero con la salvedad de que el descriptor del archivo que utiliza es en realidad un descriptor de conector.

La llamada writev es una generalización de write y puede utilizarse para escribir datos en cualquier descriptor, ya sea archivo o conector. Su declaración es la siguiente:

```
#include <sys/uio.h>
ssize_t writev(int filedes, const struct iovec *iov, size_t iovcnt);
```

writev va a escribir datos en el archivo descrito por filedes y los va a tomar de los distintos bloques de memoria especificados en el arreglo iov. El significado del arreglo iov y del argumento iovcnt es el mismo que vimos para la llamada readv.

Si bien write y writev pueden escribir en cualquier descriptor (descriptor de archivo o conector), las llamadas send, sendto y sendmsg sólo puede escribir en un conector. Las declaraciones de estas funciones quedan como sigue:

```
#include <sys/socket.h>
int send(int sfd, void *buf, int len, int flags);
int sendto(int sfd, void *buf, int len, int flags, void *to, int tolen);
int sendmsg(int sfd, struct msghdr msg[], int flags);
```

En todas las llamadas, sfd es el descriptor del conector en que se van a escribir los datos, buf es un apuntador al buffer donde están los datos que se desean enviar y len es el número de bytes que hay en el buffer.

En los conectores orientados a conexión, `SOCK_STREAM`, las llamadas pueden usarse sólo después de haber establecido una conexión previa vía llamada a `connect`. En los conectores no orientados a conexión, `SOCK_DGRAM`, debe usarse la función `sendto`, a menos que se especifique una dirección de destino con una llamada a `connect`. Si ya se ha especificado una dirección de destino, al llamar a `sendto` se va a producir un error si el parámetro `to` contiene una dirección. La dirección del conector destino está en la posición de memoria apuntada por `to` y su tamaño es `tolen`.

En los conectores del tipo datagrama para los que no se ha definido una dirección mediante la llamada correspondiente a `bind`, si se envía un mensaje con `sendto`, el sistema va a elegir de forma automática una dirección local, pero no hay garantía de que esa dirección siga siendo la misma en sucesivas llamadas a `sendto`.

La llamada `sendmsg` es una generación de las dos anteriores. La diferencia con ellas es que los datos a enviar pueden proceder de varios bloques de memoria. El argumento `msg` es un arreglo de cabeceras de mensaje, cada una de las cuales tiene una estructura como la definida para la función `recvmsg` (`struct msghdr`).

Para las tres llamadas, el argumento `flag` tiene el mismo significado y sus valores posibles son 0 o `MSG_OOB` (enviar el mensaje fuera de banda). En los conectores de la familia `AF_UNIX` no se pueden enviar mensajes fuera de banda.

Las tres funciones estudiadas devuelven el total de bytes escritos en el conector.

10.2.8 Cierre de una conexión – close

Una vez que un proceso no necesita realizar más accesos a un conector, puede desconectarse del mismo. Para ello, y aprovechando que un conector es tratado sintácticamente como si fuera un archivo, podemos usar la llamada `close`. Esta llamada va a cerrar el conector en sus dos sentidos (servidor-cliente y cliente-servidor). Veremos más adelante que hay otra llamada que tiene un control más fino a la hora de cerrar y permite deshabilitar uno o los dos sentidos del conector.

10.3 Ejemplos de servidores y clientes

En este apartado vamos a ver una serie de programas escritos para mostrar la forma que tiene tanto los servidores como los clientes que manejan las distintas familias de conectores y sus protocolos asociados.

Los ejemplos van a ser programa costos que no tienen pretensiones de aplicación, pero ilustran muy bien la estructura básica de un programa que trabaja con la red. Sobre todo, dejan muy claro cuál es la secuencia de llamadas necesarias para conseguir establecer la comunicación. Por otro lado, vamos a aprovechas para introducir cuáles son los convenios que utilizan los distintos protocolos para formas las direcciones de los nodos. Veremos también algunas funciones útiles para manejar estas direcciones.

Vamos a ver ejemplos escritos para dos familias de conectores: `AF_UNIX` y `AF_INET`. El motivo para elegir estas dos es que la primera está presenta en todas las instalaciones 4.3BSD, ya que no requiere hardware de red adicional, y la segunda es la familia más utilizada para redes de la norma 802.

En todos los ejemplos, tanto el servidor como el cliente se van a justar a una misma funcionalidad. El cliente enviará cadenas de caracteres al servidor y el servidor va a devolver esa misma cadena, pero acompañada del nombre que recibe la computadora donde se ejecuta, el nombre y versión del sistema operativo, el PID del proceso que dio servicio al cliente, y la fecha y hora en la que se prestó servicio al cliente.

Para leer estos últimos datos se utilizarán las llamadas uname y time. La llamada uname se declara como sigue:

```
#include <sys/utsname.h>
int uname(struct utsname *uname);
```

uname devuelve, en la estructura apuntada por name, información referente al nombre y versión del sistema operativo. La definición de struct utsname se encuentra en el archivo cabecera <sys/utsname.h>.

```
struct utsname {
      char sysname[9];        /* Nombre del sistema. */
      char nodename[9];       /* Nombre del nodo. */
      char release[9];        /* Edición del sistema. */
      char versión[9];        /* Versión del sistema. */
      char machine[9];        /* Tipo de computadora en la que se ejecuta
                                 el sistema. */
};
```

10.3.1 Ejemplos con conectores de la familia AF_UNIX

Los primeros ejemplos van a manejar conectores de la familia AF_UNI. Esta es una familia que se utiliza para comunicar procesos que se ejecutan en una misma computadora, por lo que no necesitan trabajar con la red.

Las direcciones que maneja esta familia tienen la misma forma que las rutas de los archivos, pero la transferencia de los datos no se hace a través del sistema de archivos, sino a través de memoria.

Vamos a plantear dos parejas de clientes-servidores, la primera trabajará con un protocolo orientada a conexión (stream), mientras que la segunda lo hará con un protocolo no orientado a conexión (datagrama).

10.3.1.1 Cliente-Servidor con conectores AF_UNIX del tipo stream

El archivo cabecera scomun.h tiene declaraciones de tipos y funciones que van a ser usados por todos los programa de manejo de conectores que veamos en este apartado y en los siguientes. El contenido de este archivo es el siguiente:

```
/**
ARCHIVO: scomun.h
DESCRIPCION:
      Declaracion de tipo y funciones utilizados por los programas
      de ejemplo de manejo de conectores.
**/
#ifndef SCOMUN_H
#define SCOMUN_H
```

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

#define CR 10 /* Carrie return */

extern char *nombre_programa;

/* Estructura que almacenará la información sobre el sistema. */
struct informacion {
	char nodo[9];
	char sistema[9];
	char version[9];
	int pid;
	char fecha[25];
};

/* Funciones. */
struct informacion leer_informacion();

/* Constantes de ayuda a la depuracion. */
#define DEP __FILE__, __LINE__

#endif
```

En `scomun.c` hay funciones que van a usar lo programa de ejemplo que veremos a continuación. Son funciones no relacionadas con el manejo de conectores.

```
/**
ARCHIVO: scomun.c
DESCRIPCION:
	Este archivo contiene funciones que van a ser usadas por los
	programas que traba
**/

#include "scomun.h"

char *nombre_programa;

/**
FUNCION: leer_informacion
DESCRIPCION:
	Esta función leer datos referentes al proceso servidor para
	enviárselos al cliente.
**/
struct informacion leer_informacion() {
	struct informacion info;
	struct utsname sistema;
	time_t fecha;

	uname(&sistema);
	strncpy(info.nodo, sistema.nodename, 9);
	strncpy(info.sistema, sistema.sysname, 9);
	strncpy(info.version, sistema.release, 9);
```

```
	info.pid = getpid();
	time(&fecha);
	strncpy(info.fecha, (char *) asctime(localtime(&fecha)), 24);
	info.fecha[24] = 0;
	return info;
}

/**
FUNCION: error
DESCRIPCION:
	Función para imprimir mensajes de error.
**/
void error(char *archivo, int linea, char *mensaje) {
	fprintf(stderr, "%s: (%s - %d). %s - %s\n", nombre_programa, archivo,
linea, mensaje, strerror(errno));
}
```

El archivo de cabecera `u_addr.h` contiene las declaraciones utilizadas por los programas de ejemplo que manejan conectores `AF_UNIX` del tipo `STREAM`. Fíjate en la longitud que tienen las rutas utilizadas como soporte para formas las direcciones de clientes y servidores. Aunque las direcciones pueden llegar a tener 118 bytes de longitud. Sin embargo, en las implementaciones actuales, las rutas no pueden tener más de 14 caracteres. El contenido de `u_addr.h` es el siguiente:

```
/**
ARCHIVO: u_addr.h
DESCRIPCION:
	Archivo cabecera para los ejemplos de servidores y clientes que
	emplean la familia de conectores AF_UNIX.
**/

#ifndef UNIX_ADDR_H
#define UNIX_ADDR_H

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

#define UNIX_STR_ADDR "u_str_socket"
#define UNIX_DG_ADDR "u_dg_socket"
#define UNIX_DG_TMP "u_dgtmp.XXXXXX"

#endif
```

En el archivo `str_com.c` están las funciones que necesitan todos los servidores y clientes que se comunican mediante conectores del tipo `STREAM` (ya sean de la familia `AF_UNIX` o `AF_INET`).

```
/**
ARCHIVO: str_com.c
DESCRIPCION:
	Este archivo contiene funciones que van a ser usadas por los
	programas que trabajan con conectores orientados a conexión.
**/

#include "scomun.h"
```

```
#include <stdlib.h>

/**
FUNCION: escribir_n
DESCRIPCION:
	Función utilizada para escribir en un archivo referenciado
	por el descriptor fd. Esta funcion es una interfaz con "write"
	y es necesaria porque la escritura de bloques de datos es muy
	grandes en un conector no se puede realizar mediante una sola
	llamada a "write".
**/
int escribir_n(int fd, char *buf, int nbytes) {
	int escritos, resto = nbytes;

	while (resto > 0) {
		escritos = write(fd, buf, resto);
		if (escritos == -1) {
			return -1;
		}
		resto -= escritos;
		buf += escritos;
	}
	return (nbytes - resto);
}

/**
FUNCION: leer_linea
DESCRIPCION:
	Función para leer líneas de un archivo de entrada.
**/
int leer_linea(int fd, char *buf, int nbytes) {
	int n, nb;
	char b;

	for (n = 1; n < nbytes; n++) {
		nb = read(fd, &b, 1);
		if (nb == -1) {
			return -1;
		}
		if (b == '\n') {
			/* salto de línea */
			break;
		}
		if (nb == 0) {
			if (n == 1) {
				return 0;
			} else {
				break;
			}
		}
		*buf++ = b;
	}
	*buf = 0;
	return n;
}

#define MAXLINEA 256

/**
```

```
FUNCION: recibir_mensaje_str
DESCRIPCION:
	Función ejecutada por el proceso servidor para dar servicio
	a sus clientes. Esta funcion la van a ejecutar los servidores
	que trabajen con conectores de tipo STREAM.
**/
void recibir_mensaje_str(int sfd) {
	int nbytes;
	char linea[MAXLINEA], mensaje[MAXLINEA];
	struct informacion est;
	int salida = 0;

	while (!salida) {
		/* Lectura de los mensajes procedentes del cliente. */
		nbytes = leer_linea(sfd, linea, MAXLINEA);
		if (nbytes == 0) {
			return;
		} else if (nbytes == -1) {
			error(DEP, "leer_linea");
			exit(-1);
		}
		fprintf(stdout, "Mensaje recibido: %s\n", linea);
		if (strcmp(linea, "EOT") == 0) {
			salida = 1;
			fprintf(stdout, "entré al if - salida = %d\n", salida);
		}
		/* Creando el mensaje para el cliente. */
		est = leer_informacion();
		sprintf(mensaje, "%s(%s, %s), PID(%d), %24s, %s\n",
			est.nodo, est.sistema, est.version, est.pid,
			est.fecha, linea);
		/* Envío del mensaje al cliente. */
		nbytes = strlen(mensaje);
		if (escribir_n(sfd, mensaje, nbytes) != nbytes) {
			error(DEP, "escribir_n");
			exit (-1);
		}
		fprintf(stdout, "Mensaje devuelto: %s\n", mensaje);
	}
}

/**
FUNCION: limpia_cadena
DESCRIPCION:
	Función ejecutada después de leer de la entrada estándar. Se
	ejecuta delimitar la información que se acaba de leer.
**/
void limpia_cadena(char *cadena) {
	char *p = strchr(cadena, '\n');
	if (p) {
		*p = '\0';
	}
}

/**
FUNCION: enviar_mensaje_str
DESCRIPCION:
	Función ejecutada por los procesos clientres para pedir
	servicios a su servidor. Esta función la van a ejecutar los
```

```
	clientres que trabajen con conectores del tipo STREAM.
**/
void enviar_mensaje_str(int sfd) {
	int nbytes, salida = 0;;
	char linea[MAXLINEA];

	/* Leer las lineas de la entrada estándar. */
	while (!salida) {
		fprintf(stdout, "<=="); fflush(stdout);
		fgets(linea, MAXLINEA, stdin);
		limpia_cadena(linea);
		if (strcmp(linea, "EOT") == 0) {
			salida = 1;
		}
		strcat(linea, "\n");

		/* Escribir cada linea en el conector para
		enviárselo al servidor. */
		if (escribir_n(sfd, linea, strlen(linea)) == -1) {
			error (DEP, "escribir_n");
			exit(-1);
		}

		/* Leer la respuesta procedente del servidor. */
		nbytes = leer_linea(sfd, linea, MAXLINEA);
		if (nbytes == -1) {
			error(DEP, "leer_linea");
			exit(-1);
		}

		/* Sacar el mensaje por el archivo estándar de
		salida. */
		fprintf(stdout, "==> %s\n", linea); fflush(stdout);
	}
}
```

El programa servidor que vamos a ver es `u_str_se.c`. Este servidor es del tipo concurrente. Su código es el siguiente:

```
/**
ARCHIVO: u_str_se.c
DESCRIPCION:
	Programa servidor de ejemplo. Este servidor se comunica a través
	de conectores de la familia AF_UNIX y del tipo SOCK_STREAM.
**/

#include "scomun.h"
#include "u_addr.h"
#include <stdlib.h>

int main(int argc, char *argv[]) {
	int sfd, nsfd, pid;
	struct sockaddr_un ser_addr, cli_addr;
	int ser_addr_len, cli_addr_len;

	nombre_programa = argv[0];

	/* Apertura de un conector orientado a conexión de la familia
```

```
	AF_UNIX.*/
	if ((sfd = socket(AF_UNIX, SOCK_STREAM, 0)) == -1) {
		error(DEP, "abrir conector");
		return -1;
	}

	/* Publicar la dirección del servidor. */
	unlink(UNIX_STR_ADDR);
	ser_addr.sun_family = AF_UNIX;
	strcpy(ser_addr.sun_path, UNIX_STR_ADDR);
	ser_addr_len = strlen(ser_addr.sun_path) +
			sizeof(ser_addr.sun_family);
	if (bind(sfd, (struct sockaddr *) &ser_addr, ser_addr_len) == -1) {
		error(DEP, "bind");
		return -1;
	}

	/* Declaración de una fila con 5 elementos para peticiones de
	conexión. */
	listen(sfd, 5);

	/* Ciclo de lectura de peticiones de conexión. */
	while (1) {
		cli_addr_len = sizeof(cli_addr);
		if ((nsfd = accept(sfd, (struct sockaddr *) &cli_addr,
&cli_addr_len)) == -1) {
			error(DEP, "accept");
			return -1;
		}

		/* Creación de un proceso hijo para atender al cliente. */
		if ((pid = fork()) == -1) {
			error(DEP, "fork");
		} else if (pid == 0) {
			/* Código del proceso hijo. */
			close(sfd);
			recibir_mensaje_str(nsfd);
			close(nsfd);
			return 0;
		}
		/* Código del proceso padre. */
		close(nsfd);
		return 0;
	}
}
```

El programa cliente se encuentra en el archivo `u_str_cl.c` y su código es el siguiente:

```
/**
ARCHIVO: u_str_cl.c
DESCRIPCION:
	Programa cliente de ejemplo. Este cliente se comunica a través de
	conectores de la familia AF_UNIX y del tipo SOCK_STREAM.
**/

#include "scomun.h"
#include "u_addr.h"
#include <stdlib.h>
```

```
int main(int argc, char *argv[]) {
	int sfd;
	struct sockaddr_un ser_addr;
	int ser_addr_len;

	nombre_programa = argv[0];

	/* Apertura de un conector orientado a conexión de la familia
	AF_UNIX. */
	if ((sfd = socket(AF_UNIX, SOCK_STREAM, 0)) == -1) {
		error(DEP, "apertura del conector");
		return -1;
	}

	/* Petición de conexión con el servidor. */
	ser_addr.sun_family = AF_UNIX;
	strcpy(ser_addr.sun_path, UNIX_STR_ADDR);
	ser_addr_len = strlen(ser_addr.sun_path) +
			sizeof(ser_addr.sun_family);
	if (connect(sfd, (struct sockaddr *) &ser_addr, ser_addr_len) == -1) {
		error(DEP, "conexión");
		return -1;
	}

	enviar_mensaje_str(sfd);
	close(sfd);
	return 0;
}
```

10.3.1.2 Cliente-Servidor con conectores AF_UNIX del tipo datagrama

Antes de implementar ejemplos con conectores no orientada a conexión hay que recordar que el protocolo de los datagramas no garantiza la distribución de los mensajes que se envían. Por otro lado, tampoco se garantiza que la secuencia de recepción de los mensajes coincida con la secuencia de la emisión de estos. Lo único que tenemos garantizado es que los mensajes van a viajar por la red sin fraccionarse, por lo que cuando se recibe un mensaje se tiene la garantía de que está íntegro. Para poder apreciar el orden en el que se distribuyen los menajes, los vamos a acompañar de un número secuencia que indica el orden en que fueron generados por el cliente. Este número ya constituye en sí un protocolo rudimentario que puede servir para reconstruir la verdadera secuencia que deben seguir los mensajes, si bien en estos ejemplos sólo lo utilizaremos a título de información.

El programa servidor se compone de dos archivos cabecera, `scomun.h` y `u_addr.h`, y tres archivos fuente, `scomun.c`, `dg_com.c` y `u_dg_se.c`. Los archivos `scomun.h`, y `u_addr_h` y `scomun.c` ya han sido comentados en el apartado anterior por lo que no los vamos a repetir aquí. El módulo `dg_com.c` contiene las funciones que van a usar los servidores y clientes no orientados a conexión, tanto de este ejemplo como de los siguientes. El contenido de este módulo es el siguiente:

```
/**
ARCHIVO: dg_com.c
DESCRIPCION:
	Este archivo contiene funciones que van a ser usadas por los
```

```
	programas que trabajan con conectores orientados no orientados a
	conexión (datagramas).
**/

#include "scomun.h"
#include <stdlib.h>

#define MAXLINEA 255

/**
FUNCION: limpia_cadena
DESCRIPCION:
	Función ejecutada después de leer de la entrada estándar. Se
	ejecuta delimitar la información que se acaba de leer.
**/
void limpia_cadena(char *cadena) {
	char *p = strchr(cadena, '\n');
	if (p) {
		*p = '\0';
	}
}

/**
FUNCION: imprimir_direccion_cliente
DESCRIPCION:
	Presenta por salida estándar la direccion de la computadora a la
	cual está conectado un proceso cliente. El tipo de presentación
	depende de la familia al a que pertenece el conector.
**/
void imprimir_direccion_cliente (struct sockaddr *pcli_addr, int cli_len) {
	struct sockaddr_un *u_addr;
	struct sockaddr_in *i_addr;

	switch (pcli_addr->sa_family) {
		case AF_UNIX:
			u_addr = (struct sockaddr_un *) pcli_addr;
			fprintf(stdout, "Dirección cliente: %14s\n",
				u_addr->sun_path);
			break;
		case AF_INET:
			i_addr = (struct sockaddr_in *) pcli_addr;
			fprintf(stdout, "Nodo del cliente: %s\n",
				(char *) inet_ntoa(i_addr->sin_addr));
			fprintf(stdout, "Puerto del cliente: %d\n",
				i_addr->sin_port);
			break;
		default:
			error(DEP, "imprimir_direccion_cliente");
	}
}

/**
FUNCION: recibir_mensaje_dg
DESCRIPCION:
	Función ejecutada por el proceso servidor para dar servicio a sus
	cliente. Esta función la van a ejecutar los servidores que trabajen
	con conectores del tipo datagrama.
**/
void recibir_mensaje_dg(int sfd, struct sockaddr *pcli_addr,
```

```
	int maxcli_len) {
	int nbytes, cli_len = maxcli_len, nro_mensaje_rec;
	char linea[MAXLINEA], mensaje[MAXLINEA];
	struct informacion info;

	while(1) {
		/* Lecuta de los mensajes procedentes del cliente. */
		nbytes = recvfrom(sfd, mensaje, MAXLINEA, 0, pcli_addr,
			&cli_len);
		if (nbytes == -1) {
			error(DEP, "recibir_mensaje_dg (recvfrom)");
			exit(-1);
		}
		mensaje[nbytes] = 0;
		sscanf(mensaje, "%d %s", &nro_mensaje_rec, linea);
		imprimir_direccion_cliente(pcli_addr, cli_len);
		fprintf(stdout, "Numero mensaje: %d\n", nro_mensaje_rec);
		fprintf(stdout, "Mensaje: %s\n", linea);

		/* Formación del mensaje para el cliente. */
		info = leer_informacion();
		sprintf(mensaje, "%s(%s, %s), PID(%d), %24s, %d %s\n",
			info.nodo, info.sistema, info.version, info.pid,
			info.fecha, nro_mensaje_rec, linea);

		/* Envío del mensaje al cliente. */
		nbytes = strlen(mensaje);
		if (sendto(sfd, mensaje, nbytes, 0, pcli_addr,
			cli_len) != nbytes) {
			error(DEP, "recibir_mensaje_dg (sendto)");
			exit(-1);
		}
		fprintf(stdout, "Mensaje devuelto: %s\n", mensaje);
	}
}

/**
FUNCION: enviar_mensaje_dg
DESCRIPCION:
	Función ejecutada por los procesos cliente para pedir servicios
	a su servidor. Esta función la van a ejecutar los clientes que
	trabajen con conectores del tipo datagrama.
**/
void enviar_mensaje_dg(int sfd, struct sockaddr *pser_addr,
	int maxser_len) {
	int nbytes, nro_mensaje_env = 0;
	char linea[MAXLINEA], mensaje[MAXLINEA];
	int salida = 0;

	/* Leer líneas de entrada estándar. */
	do {
		fprintf(stdout, "<== "); fflush(stdout);
		fgets(linea, MAXLINEA, stdin);
		limpia_cadena(linea);
		fprintf(stdout, "linea: .%s.\n", linea);
		if (strcmp(linea, "EOT") == 0) {
			salida = 1;
		}
		/*Escribir cada linea en el conector para enviárselo al
```

```
            servidor. */
            sprintf(mensaje, "%d %s", ++nro_mensaje_env, linea);
            nbytes = strlen(mensaje);
            if  (sendto(sfd,  mensaje,  nbytes,  0,  pser_addr,  maxser_len)  !=
nbytes) {
                  error(DEP, "enviar_mensaje_dg (sendto)");
                  exit(-1);
            }

            /* Leer la respuesta procedente del servidor. */
            nbytes = recvfrom(sfd, mensaje, MAXLINEA, 0, NULL, NULL);
            if (nbytes == -1) {
                  error(DEP, "enviar_mensaje_dg (recvfrom)");
                  exit(-1);
            }

            /* Sacar el mensaje por la salida estándar. */
            mensaje[nbytes] = 0;
            fprintf(stdout, "==> %s\n", mensaje);
      } while (!salida);
}
```

El programa servidor se encuentra en el archivo `u_dg_se.c` y es del tipo interactivo. Como no hay una conexión con el cliente, al leer los menajes procedentes de éste es necesario conocer su dirección para saber a quién hay que dirigir la respuesta. El listado del servidor se muestra a continuación:

```
/**
ARCHIVO: u_dg_se.c
DESCRIPCION:
      Programa servidor de ejemplo. Este servidor se comunica a través de
      conectores de la familia AF_UNIX y del tipo SOCK_DGRAM.
**/

#include "scomun.h"
#include "u_addr.h"

int main(int argc, char *argv[]) {
      int sfd;
      struct sockaddr_un ser_addr, cli_addr;
      int ser_addr_len;

      nombre_programa = argv[0];

      /* Apertura de un conector orientada a conexión de la familia
      AF_UNIX.*/
      if ((sfd = socket(AF_UNIX, SOCK_DGRAM, 0)) == -1) {
            error(DEP, "abriendo conector");
            return -1;
      }

      /* Publicidad de la dirección del servidor. */
      unlink(UNIX_DG_ADDR);
      ser_addr.sun_family = AF_UNIX;
      strcpy(ser_addr.sun_path, UNIX_DG_ADDR);
      ser_addr_len = strlen(ser_addr.sun_path) +
                  sizeof(ser_addr.sun_family);
      if (bind(sfd, (struct sockaddr *) &ser_addr, ser_addr_len) == -1) {
            error(DEP, "bind");
```

```
            return -1;
      }

      /*Lectura y procesamiento de los mensajes del conector. */
      recibir_mensaje_dg(sfd, &cli_addr, sizeof(cli_addr));
      return 0;
}
```

El programa clienta se encuentra en el archivo `u_dg_cl.c`. Al no haber una conexión con el servidor. Es necesario que cada cliente tenga una dirección distinta para que los menajes se puedan encaminar correctamente dentro de la red. Las direcciones de los conectores de la familia `AF_UNIX` se forman a partir de rutas del sistema de archivos. Esto fuerza a que cada cliente tenga asociada una ruta distinta. Para crear estas rutas podemos utilizar la función estándar `mktemp`, que se declara como sigue:

```
char* mktemp(char *template);
```

`mktemp` reemplaza el contenido de la cadena apuntada por `template` por un nombre de archivo único y devuelve la dirección de `template`. La cadena `template` debe tener la forma de ruta terminada con 6 caracteres X. `mktemp` reemplazará estos caracteres por una letra y el `PID` del proceso actual. La letra se elige para que no se dé repetición con un nombre de archivo ya existente.

El código para el programa cliente es el siguiente:

```
/**
ARCHIVO: u_dg_cl.c
DESCRIPCION:
      Programa cliente de ejemplo. Este cliente se comunica a través de
      conectores de la familia AF_UNIX y del tipo SOCK_DGRAM.
**/

#include "scomun.h"
#include "u_addr.h"

int main(int argc, char *argv[]) {
      int sfd;
      struct sockaddr_un ser_addr, cli_addr;
      int ser_addr_len, cli_addr_len;

      nombre_programa = argv[0];

      /* Apertura de un conector no orientado a conexión de la familia
      AF_UNIX. */
      if ((sfd = socket(AF_UNIX, SOCK_DGRAM, 0)) == -1) {
            error(DEP, "apertura del conector");
            return -1;
      }

      /*Formación de la dirección del servidor. */
      ser_addr.sun_family = AF_UNIX;
      strcpy(ser_addr.sun_path, UNIX_DG_ADDR);
      ser_addr_len = strlen(ser_addr.sun_path) + sizeof(ser_addr.sun_family);

      /* Formación y publicidad de la dirección dek cliente. Al tratarse
      de conectores datagrama, cada cliente debe tener una dirección
```

```
        distinta de los demás. Por eso formamos la dirección del cliente
        mediante un nombre de archivo temporal creado con "mktemp". */
        cli_addr.sun_family = AF_UNIX;
        strcpy(cli_addr.sun_path, UNIX_DG_TMP);
        mktemp(cli_addr.sun_path);
        cli_addr_len = strlen(cli_addr.sun_path) + sizeof(cli_addr.sun_family);
        if (bind(sfd, (struct sockaddr *) &cli_addr, cli_addr_len) == -1) {
                error(DEP, "bind");
                return -1;
        }

        /* Comunicación con el servidor. */
        enviar_mensaje_dg(sfd, (struct sockaddr *) &ser_addr, ser_addr_len);
        close(sfd);
        unlink(cli_addr.sun_path);
        return 0;
}
```

10.3.2 Ejemplos con conectores de la familia AF_INET

La familia `AF_INET` soportar los protocolos DARPA para las comunicaciones internet: TCP (Transmission Control Protocol) y UDP (User Datagram Protocol).

TCP es el protocolo orientada a conexión que proporciona a los procesos de usuario un canal fiable y full-duplex. TCP está basado en IP (Internet Protocol), que es un protocolo de más bajo nivel. Aunque a veces se habla de TCP/IP como si se tratase de un solo protocolo, en el fondo no es correcto, ya que UDP también está construido sobre IP.

UDP es un protocolo no orientado a conexión, por lo que no hay garantía de que los mensajes (datagramas) siempre alcancen su destino. Si queremos realizar comunicaciones fiables sobre conectores no orientados a conexión, es necesario recurrir a protocolos de más alto nivel, o que el usuario implemente, a partir de protocolos desarrollados por él, los mecanismos que hagan fiable al canal.

Con los conectores `AF_INET` vamos a poder comunicar procesos que se estén ejecutando tanto en la misma computadora como en computadoras distintas. Esto acarrea la necesidad de una red que conecte físicamente las dos computadoras que intervienen en la comunicación. No vamos a entrar en la descripción de las tecnologías de redes, ni en sus topologías, porque está fuera de lugar para nuestra discusión. Para nosotros, una red va a ser un camino de comunicación entre computadoras, y a través de unos protocolos y unos servicios, podemos hacer uso de la misma. Como ya hemos mencionado con anterioridad, a nosotros, como programadores, usuarios de una red, lo único que nos interesa son las llamadas o conjunto de servicios con los que podemos usarla.

Antes de empezar a ver ejemplos, es necesario que aclaremos algunas ideas sobre la forma de direccionar los distintos nodos y conectores dentro de la familia AF_INET. La forma de una dirección de Internet es:

```
struct in_addr {
        u_long s_addr;      /* 32 bits que contiene la identificación de la
                               red y del nodo. */
};
```

```
struct sockaddr_in {
       short sin_family;               /* AF_INET. */
       u_short sin_port;               /* 16 bits con el número de puerto. */
       struct in_addr sin_addr;  /* 32 bits con la identificación de la
                                       red y del nodo. */
       char sin_zero[8];               /* 8 bytes no usados. */
};
```

En la estructura `sockaddr_in`, el campo `sin_family` debe valer `AF_INET`, por ser `AF_INET` la familia para la que tiene sentido una dirección de la estructura anterior.

El campo `sin_addr` contiene la dirección del nodo origen o destino del mensaje que circula por la red. Este campo es de 32 bits agrupados en 4 bytes. Estos 4 bytes se suelen representar de cara al usuario en la forma `ddd.ddd.ddd.ddd` donde `ddd` representa un número decimal que varía en el rango de 0-255. En el archivo de configuración `/etc/hosts` hay una relación de todos los nodos de que consta la red y de sus direcciones asociadas. Cada línea de este archivo tiene la forma:

`ddd.ddd.ddd.ddd nombre1 nombre2 nombre3 … nombren`

Donde `nombre1`, `nombre2`, …, `nombren` son cadenas de caracteres que se utilizan como sinónimos de la dirección `ddd.ddd.ddd.ddd`.

El campo `sin_port` se utiliza para identificar los distintos procesos que en un mismo nodo están haciendo uso de la red. Este campo es necesario, porque la dirección del nodo da acceso al ordenador con el que queremos comunicarnos, pero no específica cuál es el proceso origen o destino de los mensajes. Si en un mismo nodo se están ejecutando varios clientes o servidores, el campo `sin_port` de la dirección específica a cuál de ellos está referido el mensaje.

Esta estructura de dirección tiene aplicación sobre los protocolos TCP y UDP, por lo que los mensajes que se trasmiten a través de la red, además de los datos del usuario, deben contener información de control dedicada a su enrutamiento. Esta información se compone de los campos:

- Protocolo (TCP o UDP).
- Dirección Internet del nodo origen (32 bits).
- Puerto asociado al proceso origen del mensaje (16 bits).
- Dirección Internet del nodo destino (32 bits).
- Puerto asociado al proceso destino del mensaje (16 bits).

Así, un ejemplo de la cabecera de control del mensaje puede ser:

`[tcp, 128.10.12.02, 1200, 128.12.11, 1350]`

Antes de pasar a ver los ejemplos, vamos a estudiar un conjunto de funciones estándar que nos ayudarán a manejar las direcciones de nodo, los números de puerto y las posibles incompatibilidades entre distintas computadoras.

10.3.2.1 Funciones para la conversión de direcciones

La entrada `inet(3N)` del manual de UNIX describe un conjunto de funciones estándar para manejar direcciones Internet. En los ejemplos vamos a usar `inet_addr` e `inet_ntoa`, que se declaran como sigue:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
unsigned long inet_addr(const char *cp);
char* inet_ntoa(struct in_addr in);
```

`inet_addr` traduce la cadena de caracteres apuntada por `cp`, que contiene una dirección de nodo en el formato decimal estándar para el usuario, al formato binario definido por la estructura `in_addr`. El ejemplo siguiente muestra la traducción:

```
struct in_addr direccion;
direccion.s_addr = inet_addr("192.12.10.03");
```

`inet_toa` realiza la operación inversa de `inet_addr`. Transforma una dirección de nodo con el formato binario `struct in_addr` al formato decimal de usuario. El apuntador devuelto referencia una cadena de caracteres local a la función.

10.3.2.2 Reserva de puertos

A la hora de asignar un número de puerto a un proceso, tenemos dos posibilidades:

- Asignar un número fijo predeterminado, en cuyo caso hay que tener cuidado con los puertos que tiene reservados el sistema. Este tipo de reserva la suele emplear los servidores que necesitan una dirección con un puerto perfectamente conocido por sus clientes.
- Pedir al sistema que le reserve al proceso alguno de los puestos que se encuentren libres en ese momento. Esto se consigue poniendo el número de puerto 0 antes de hacer la llamada a `bind` para dar publicidad a una dirección.

En el caso de que el usuario fije un número de puerto, hay que tener presente que para los protocolos Internet (TCP y UDP) los números comprendidos entre 1 y 1023 están reservados por el sistema, y por lo tanto ningún proceso puede utilizarlos. Los puertos reservados son utilizados por los servidores que se ejecutan con privilegios de superusuario. Así, los puestos del intervalo [1, 255] son utilizados por las aplicaciones Internet estándar, tales como `FTP`, `TELNET`, `TFTP`, etc. Los puertos de los intervalos [256, 511] están reservados para futuras aplicaciones Internet. Por último, los puertos del intervalo [512, 1023] no están reservados para aplicaciones estándar, sino para servidores de usuarios que se ejecuten con privilegios de superusuario. Si queremos utilizar alguno de estos puertos, previamente hay que reservarlo con una llamada `rresvport`. Esta función se define como sigue:

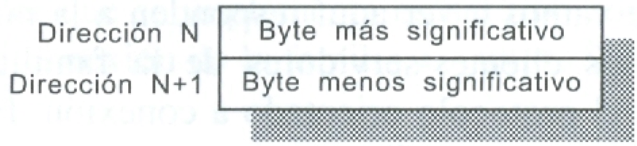
```
int rresvport(int *port);
```

Esta función crea un conector de la familia `AF_INET` y del tipo `SOCK_STREAM`, y reserva un puerto privilegiado para él. La función devuelve un descriptor del conector creado, y a través de la zona de memoria apuntada por `port`, el número del puerto reservado. El puerto reservado es el primero que se encuentra libre en el intervalo [512, 1023]. La búsqueda de puerto libre se inicia con el número 1023 y continúa en orden decreciente hasta 512. Si no se encuentra ningún puerto libre, la función devuelve el valor de -1.

Los puertos libres para el usuario se encuentran en el intervalo [1024, 65535]. De éstos, el sistema gestiona de forma automática los que se encuentran en [1024, 5000]. Por esto es conveniente que al fijar un número de puerto desde un cliente o un servidor, éste sea superior a 5000, para asegurar que no interferimos con ninguna operación de reserva llevada a cabo por el sistema.

10.3.2.3 Orden de los bytes

Al comunicar dos procesos que se ejecutan en computadoras diferentes se presenta un problema que no se da en las comunicaciones sobre una misma computadora. Se trata del orden con el que se almacenan en memoria las palabras *multibyte*. Se han acuñado los términos *big endian* y *little endian* para referirse a las dos formas que hay de tratar palabras *multibyte*. Tomemos como ejemplo una palabra de 2 bytes: las computadoras que trabajan con formato *big endian* almacenan el byte más significativo en la posición baja de memoria, y el menos significativo en la posición alta. El formato *little endian* es justo el contrario del anterior.



a) Formato *big endian*.

b) Formato *little endian*.

El formato que emplean los protocolos TCP y UDP es *big endian*, pero la computadora que los soporta puede manejar cualquier tipo de formato. Para solventar las potenciales diferencias de comunicación entre una arquitectura de computadoras y los protocolos Internet, en la entrada `byteorder(3N)` del manual de UNIX hay definidas 4 funciones estándar de conversión de formato. Éstas son: `htonl`, `htons`, `ntoht` y `ntohs`, y sus declaraciones son las siguientes:

```
#include <sys/types.h>
#include <netinet/in.h>
unsigned long htonl(unsigned long hostlong);
unsigned short htons(unsigned short hostshort);
unsigned long ntohl(unsigned long netlong);
unsigned short ntohs(unsigned short netshort);
```

- `htonl` convierte datos `unsigned long` de formato que maneja la computadora al formato que maneja la red.
- `htons` convierte datos `unsigned short` del formato que maneja la computadora al formato que maneja la red.
- `ntohl` convierte datos `unsigned long` del formato que maneja la red al formato que maneja la computadora.

- ntohs convierte datos unsigned short del formato que maneja la red al formato que maneja la computadora.

10.3.2.4 Cliente-Servidor con conectores AF_INET del tipo Stream

Los ejemplos que vamos a ver aquí responder a la misma funcionalidad que ya hemos visto en los clientes –servidores de la familia AF_UNIX. En la familia AF_INET, TCP es el protocolo orientado a conexión. Por eso, el programa servidor se llamada i_tcp_se.c y el cliente i_tcp_cl.c.

El programa servidor se compone de los archivos cabecera scomun.h, i_addr.h, y de los módulos fuente s_comun.c, str_com.c e i_tcp_se.c. De todos estos archivos, sólo i_addr.h e i_tcp_se.c son nuevos.

i_addr.h contiene las declaraciones de direcciones necesarias para establecer conexiones y para enviar datagramas. El sistema en el que se han implementado los ejemplos consta de 5 computadoras conectadas a red y sus direcciones Internet, así como sus nombres, figuran en el archivo /etc/hosts visto anteriormente. Este primer par de programa cliente-servidor está codificado para que el cliente sólo se puede ejecutar en el nodo de la dirección 192.0.0.5, tal y como muestran las constantes del archivo i_addr_h:

```
/**
ARCHIVO: i_addr.h
DESCRIPCION:
	Archivo cabecera para los ejemplos de servidores y clientes que
	emplean la familia de conectores AF_INET
**/

#ifndef I_ADDR_H
#define I_ADDR_H

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

#define PUERTO_SERVIDOR_TCP 7000
#define PUERTO_SERVIDOR_UDP 7000
#define DIRECCION_NODO_SERVIDOR "127.0.0.1"

#endif
```

El programa servidor que se implementa es del tipo concurrente, al igual que el servidor orientado a conexión de la familia AF_UNIX. Su código se puede ver a continuación:

```
/**
ARCHIVO: i_tcp_se.c
DESCRIPCION:
	Programa servidor ejemplo. Este servidor se comunica a través de
	conectores de la familia AF_INET y del tipo SOCK_STREAM.
**/

#include "scomun.h"
#include "i_addr.h"
```

```
int main(int argc, char *argv[]) {
	int sfd, nsfd, pid;
	struct sockaddr_in ser_addr, cli_addr;
	int cli_addr_len;

	nombre_programa = argv[0];

	/* Apertura de un conector orientado a conexión de la familia
	   AF_INET. */
	if ((sfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
		error(DEP, "abriendo socket");
		return -1;
	}

	/* Publicidad de la dirección del servidor. */
	ser_addr.sin_family = AF_INET;
	ser_addr.sin_addr.s_addr = inet_addr(DIRECCION_NODO_SERVIDOR);
	ser_addr.sin_port = htons(PUERTO_SERVIDOR_TCP);
	if (bind(sfd, (struct sockaddr *) &ser_addr,
		sizeof(ser_addr)) == -1) {
		error(DEP, "bind");
		return -1;
	}

	/* Declaración de un cola de 5 elementos para peticiones de
	   conexiones. */
	listen(sfd, 5);

	while(1) {
		cli_addr_len = sizeof(cli_addr);
		if ((nsfd = accept(sfd, (struct sockaddr *) &cli_addr,
			&cli_addr_len)) == -1) {
			error(DEP, "accept");
			return -1;
		}
		if ((pid = fork()) == -1) {
			error(DEP, "fork");
			return -1;
		} else if (pid == 0) {
			/* Código del proceso hijo. */
			close(sfd);
			recibir_mensaje_str(nsfd);
			close(nsfd);
			return 0;
		}
		/* Código del proceso padre. */
		close(nsfd);
	}
}
```

El código del programa cliente está en el archivo `i_tcp_cl.c`:

```
/**
ARCHIVO: i_tcp_cl.c
DESCRIPCION:
	Programa cliente de ejemplo. Este cliente se comunica a través de
	conectores de la familia AF_INET y del tipo SOCK_STREAM.
**/
```

```
#include "scomun.h"
#include "i_addr.h"

int main(int argc, char *argv[]) {
	int sfd;
	struct sockaddr_in ser_addr;

	nombre_programa = argv[0];

	/* Apertura de un conector oriendado a conexión de la familia
	AF_INET. */
	if ((sfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
		error(DEP, "apertura del conector");
		return -1;
	}

	/* Petición de conexión con el servidor. */
	ser_addr.sin_family = AF_INET;
	ser_addr.sin_addr.s_addr = inet_addr(DIRECCION_NODO_SERVIDOR);
	ser_addr.sin_port = htons(PUERTO_SERVIDOR_TCP);
	if (connect(sfd, (struct sockaddr *) &ser_addr,
		sizeof(ser_addr)) == -1) {
		error(DEP, "conexion");
		return -1;
	}

	/* Comunicación con el servidor. */
	enviar_mensaje_str(sfd);
	close(sfd);
	return 0;
}
```

10.3.2.5 Cliente-Servidor con conectores AF_INET del tipo datagrama

En la familia `AF_INET`, el envío de datagramas se realiza con el protocolo UDP. Los programas cliente y servidor van a ser `i_udp.cl.c` e `i_udp_se.c`, respectivamente. Ambos se van a apoyar en los archivos de cabecera `scomun.h` e `i_addr.h`, así como en los módulos de código `scomun.c` y `dg_com.c`.

El listado del programa servidor lo podemos ver a continuación:

```
/**
ARCHIVO: i_udp_se.c
DESCRIPCION:
	Programa servidor de ejemplo. Este servidor se comunica a través de
	conectores de la familia AF_UNIX y del tipo SOCK_DGRAM.
**/

#include "scomun.h"
#include "i_addr.h"

int main(int argc, char *argv[]) {
	int sfd;
	struct sockaddr_in ser_addr, cli_addr;

	nombre_programa = argv[0];

	/* APertura de un conector no orientado a conexión de la familia
```

```
	AF_INET. */
	if ((sfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
		error(DEP, "abriendo conector");
		return -1;
	}

	/* Publicidad de la dirección del servidor. */
	ser_addr.sin_family = AF_INET;
	ser_addr.sin_addr.s_addr = inet_addr(DIRECCION_NODO_SERVIDOR);
	ser_addr.sin_port = htons(PUERTO_SERVIDOR_UDP);
	if (bind(sfd, (struct sockaddr *) &ser_addr,
		sizeof(ser_addr)) == -1) {
		error(DEP, "bind");
		return -1;
	}

	/* Lectura y procesamiento de los mensajes del conector. */
	recibir_mensaje_dg(sfd, &cli_addr, sizeof(cli_addr));
	close(sfd);
	return 0;
}
```

El programa cliente está disponible en el archivo `i_udp_cl.c`:

```
/**
ARCHIVO: u_udp_cl.c
DESCRIPCION:
	Programa cliente de ejemplo. Este cliente se comunica a través de
	conectores de la familia AF_INET y del tipo SOCK_DGRAM.
**/

#include "scomun.h"
#include "i_addr.h"

int main(int argc, char *argv[]) {
	int sfd;
	struct sockaddr_in ser_addr, cli_addr;
	int ser_addr_len, cli_addr_len;

	nombre_programa = argv[0];

	/* Apertura de un conector no orientado a conexión de la familia
	AF_UNIX. */
	if ((sfd = socket(AF_UNIX, SOCK_DGRAM, 0)) == -1) {
		error(DEP, "apertura del conector");
		return -1;
	}

	/* Formación de la dirección del servidor. */
	ser_addr.sin_family = AF_INET;
	ser_addr.sin_addr.s_addr = inet_addr(DIRECCION_NODO_SERVIDOR);
	ser_addr.sin_port = htons(PUERTO_SERVIDOR_UDP);

	/* Formación y publicidad de la dirección del cleintes. Al
	   tratarse de conectores datagrama, cada cliente debe tener una
	   dirección distinta de los demás. Por eso dejamos que sea el
	   sistema quien elija la dirección automáticamente. */
	cli_addr.sin_family = AF_INET;
	cli_addr.sin_addr.s_addr = htonl(INADDR_ANY);
```

```
        /* El sistema elige el número de puerto. */
        cli_addr.sin_port = htons(0);
        if (bind(sfd, (struct sockaddr *) &cli_addr,
                sizeof(cli_addr)) == -1) {
                error(DEP, "bind");
                return -1;
        }

        /* Comunicación con el servidor. */
        enviar_mensaje_dg(sfd, &ser_addr, sizeof(ser_addr));
        close(sfd);
        return 0;
}
```

Para compilar los cuatro pares de programa de servidores y clientes que hemos visto en este apartado de ejemplos, conviene utilizar el programa `make`. A continuación, podemos ver un archivo `makefile` para llevar a cabo la compilación de los ejemplos:

```
###
# ARCHIVO: makefile
# DESCRIPCIÓN:
#       Archivo con la descripción de los programas que intervienen en la
#       compilación de los distintos programas de ejemplo para conectores.
#
#       Para compilar todas las aplicaciones, escribir:
#          $ make all
#
###

# Servidor-cliente para conectores de la familia AF_UNIX y del tipo
# SOCK_STREAM
u_str_se: makefile u_str_se.o str_com.o scomun.o
        gcc -o u_str_se u_str_se.o str_com.o scomun.o

u_str_cl: makefile u_str_cl.o str_com.o scomun.o
        gcc -o u_str_cl u_str_cl.o str_com.o scomun.o

u_str_se.o:
        gcc -c u_addr.h scomun.h scomun.c u_str_se.c

u_str_cl.o:
        gcc -c u_addr.h scomun.h scomun.c u_str_cl.c

str_com.o:
        gcc -c scomun.h str_com.c

scomun.o:
        gcc -c scomun.h scomun.c

# Servidor-cliente para conectores de la familia AF_UNIX y del tipo
# SOCK_DGRAM
u_dg_se: makefile u_dg_se.o dg_com.o scomun.o
        gcc -o u_dg_se u_dg_se.o dg_com.o scomun.o

u_dg_cl: makefile u_dg_cl.o dg_com.o scomun.o
        gcc -o u_dg_cl u_dg_cl.o dg_com.o scomun.o

u_dg_se.o:
```

```
	gcc -c u_addr.h scomun.h u_dg_se.c

u_dg_cl.o:
	gcc -c u_addr.h scomun.h u_dg_cl.c

dg_com.o:
	gcc -c scomun.h dg_com.c

# Servidor-cliente para conectores de la familia AF_INET y del tipo
# SOCK_STREAM
i_tcp_se: makefile i_tcp_se.o str_com.o scomun.o
	gcc -o i_tcp_se i_tcp_se.o str_com.o scomun.o

i_tcp_cl: makefile i_tcp_cl.o str_com.o scomun.o
	gcc -o i_tcp_cl i_tcp_cl.o str_com.o scomun.o

i_tcp_se.o:
	gcc -c i_addr.h scomun.h i_tcp_se.c

i_tcp_cl.o:
	gcc -c i_addr.h scomun.h i_tcp_cl.c

# Servidor-cliente para conectores de la familia AF_INET y del tipo
# SOCK_DGRAM
i_udp_se: makefile i_udp_se.o dg_com.o scomun.o
	gcc -o i_udp_se i_udp_se.o dg_com.o scomun.o

i_udp_cl: makefile i_udp_cl.o dg_com.o scomun.o
	gcc -o i_udp_cl i_udp_cl.o dg_com.o scomun.o

i_udp_se.o:
	gcc -c i_addr.h scomun.h i_udp_se.c

i_udp_cl.o:
	gcc -c i_addr.h scomun.h i_udp_cl.c

# Orden para compilar todas las aplicaciones
all: u_str_se u_str_cl u_dg_se u_dg_cl i_tcp_se i_tcp_cl i_udp_se i_udp_cl
```

La orden para llevar a cabo la compilación es:

```
$ make all
```

10.4 Algunas otras llamadas y funciones

Las llamadas al sistema estudiadas en los apartados anteriores son fundamentales para construir cualquier tipo de programa que se comunique a través de conectores. En este apartado vamos a ver más llamadas y funciones de biblioteca que amplían el repertorio estudiado y que facilitan la labor del programador.

10.4.1 Nombres de un conector –getsockname, getpeername

Para determinar las direcciones de los procesos conectados, se utilizan las llamadas `getsockname` y `getpeername`. Estas llamadas tienen aplicación en conectores orientados a conexión y se declaran de la siguiente forma:

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

```
int getsockname(int sfd, void *addr, int *addrlen);
int getpeername(int sfd, void *addr, int *addrlen);
```

`getsockname` devuelve sobre la estructura apuntada por `addr` la dirección del conector cuyo descriptor coincide con `sfd`. `getpeername` devuelve sobre la estructura apuntado por `addr` la dirección del conector que se encuentra conectado a `sfd`. En ambas llamadas, `addrlen` le indica a la función cuál es el tamaño en bytes de la estructura apuntada por `addr`, y al salir de la función contiene el tamaño real de la dirección.

Actualmente estas dos llamadas no están disponibles para conectores de la familia AF_UNIX, pero en el futuro se corregirá esta situación.

10.4.2 Nombre del nodo actual – gethostname

Para determinar el nombre oficial que tiene un nodo dentro de la red, podemos usar la llamada `gethostname`, que se declara como sigue:

```
#include <unistd.h>
int gethostname(char *hostname, size_t size);
```

`gethostname` devuelve en la cadena apuntada por `hostname` el nombre oficial del nodo que hace la llamada. `size` indica la longitud de la cadena `hostname`. Esta llamada puede implementarse a partir de la llamada `uname`, que es más genérica.

10.4.3 Construcción de tuberías a partir de conectores

En el sistema 4.3BSD, la comunicación clásica mediante tuberías se puede llevar a cabo en base a las facilidades de comunicación que aportan los conectores. La llamada `socketpair` se utiliza para crear un par de conectores que posibilitan, cada uno por separado, una comunicación bidireccional. Esta función se declara como sigue:

```
#include <sys/socket.h>
int socketpair(int family, int type, int protocol, int sockvec[2]);
```

Los parámetros `family`, `type` y `protocol` tiene el mismo significado que se estudió para la llamada `socket`:

- `family`. Familia a la que pertenecerán los conectores creados. Actualmente sólo pueden ser de la familia `AF_UNIX`.
- `type`. Tipo de conector (`SOCK_STREAM`, `SOCK_DGRAM`).
- `protocol`. Protocolo para emplear. Generalmente vale 0.

`sockvec` es un arreglo de dos números `int`, donde se van a devolver los descriptores de los dos conectores creados (similar a la función `pipe`).

Con `family` sólo puede tomar el valor `AF_UNIX`, las dos formas posibles de llamar a `socketpair` son:

```
int par_de_sockets[2];
...
socketpair(AF_UNIX, SOCK_STREAM, 0, par_de_sockets);
```

```
/* o */
socketpair(AF_UNIX, SOCK_DGRAM, 0, par_de_sockets);
```

Si la llamada se ejecuta satisfactoriamente, devolverá el valor 0, en caso contrario, -1.

10.4.4 Cierre de un conector

Sabemos que la llamada `close` puede utilizarse para cerrar un conector; sin embargo, existe otra llamada que da un control más fino a la hora de cerrarlo. Los conectores son mecanismos de comunicación bidireccionales que posibilitan en todo momento una comunicación full-duplex, y en ellos, los datos que fluyen en un sentido son totalmente independientes de los que fluyen en sentido opuesto. La llamada `shutdown` se utiliza para cerrar total o parcialmente un conector. Se declara como sigue:

```
int shutdown(int sfd, int how);
```

`sfd` es el descriptor del conector sobre el que va a actuar la llamada y `how` es el tipo de acción que va a llevar a cabo. Los valores de `how` que tienen sentido son:

Valores	Significado
0	Se deshabilita la recepción de datos del conector.
1	Se deshabilita el envío de datos a través de conector.
2	Se deshabilita el envío y recepción de datos. Es equivalente a cerrar el conector con una llamada a `close`.

10.4.5 Lectura del archivo /etc/hosts

En el archivo `/etc/hosts` se guarda la información sobre las direcciones Internet y los nombres de todos los nodos que hay conectados a una red. Para acceder a esta información podemos usar las funciones estándar en la entrada `gethostent(3N)` del manual de UNIX.

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
struct hostent* gethostent();
struct hostent* gethostbyname(char *name);
struct hostent* gethostbyaddr(const char *addr, int len, int
type);
int sethostname(int stayopen);
int endhostent();
```

`gethostent`, `gethostbyname` y `gethostbyaddr` devuelven un apuntador a una estructura local del tipo `hostent`, que se define en `<netdb.h>` como sigue:

```
struct hostent {
      char* h_name;
      char** h_aliases;
      int h_addrtype;
      int h_length;
      char** h_addr_list;
```

```
};
```

El significado de los distintos campos es el siguiente:

Campo	Significado
h_name	Nombre oficial del nodo.
h_aliases	Arreglo con los nombres alternativos del nodo. Este arreglo termina con un elemento `NULL`.
h_addrtype	Tipo de la dirección que se maneja. Va a ser siempre `AF_INET`.
h_addr_list	Arreglo con las direcciones de red a que responde el nodo. Este arreglo termina con un elemento `NULL`.
h_addr	Primer elemento del arreglo `h_addr_list`.

`gethostent` lee la siguiente línea significativa del archivo `/etc/hosts`. Si es necesario, abre el archivo en modo sólo lectura. La apertura se produce en la primera llamada a la función. Cuando la función llega al final del archivo, devuelve un apuntador a `NULL`.

`sethostname` abre el archivo `/etc/hosts` y sitúa el apuntador de lectura al principio del mismo. Si el indicador `stayopen` es distinto de 0, la base de datos de nodos se cierra después de cada llamada a `gethostent`.

`endhostent` cierra el archivo abierto con `gethostent` o `sethostent`.

`gethostbyname` busca secuencialmente desde el principio de `/etc/hosts` hasta que encuentra un nombre de nodo (oficial o alias) que coincida con el especificado en el apuntador `name`. Si la función llega al final de archivo sin encontrar el nodo buscado, devolverá un apuntador a `NULL`.

`gethostbytaddr` busca secuencialmente desde el principio de `/etc/hosts` hasta que encuentre una dirección de nodo que coincida con la especificada por la variable `addr`. El parámetro `len` es el total de bytes de la dirección calculados mediante la operación `sizeof` (`in_addr`), y `type` debe ser la constante `AF_INET`. Si la función llega al final de archivo sin encontrar el nodo buscado, devolverá un apuntador a `NULL`.

Referencias

Márquez, F. (2004). *Unix Programación Avanzada.* Colombia: Alfa-Omega.

Stevens, W. R. (2008). *Advanced programming in the UNIX environment.* Addison-Wesley.