Programación de aplicaciones distribuidas usando *sockets*

1^a Versión: José Luis Díaz. Octubre 1994.
2^a Versión: José Ramón Arias. Nov. 1998, Oct. 1999, Oct. 2000.

Índice general

1. Programación de aplicaciones distribuidas usando *sockets* 2

1.1. Introducción . . . 2
1.2. Implementación en el sistema operativo . . . 2
1.3. Conceptos de diseño . . . 3
1.3.1. Dirección final genérica . . . 5
1.3.2. Utilización del API para TCP/IP . . . 5
1.4. Programación con el interfaz de *sockets* . . . 6
1.5. Uso de los *sockets* . . . 7
1.5.1. Creación del *socket*: `socket()` . . . 8
1.5.2. Asignar una dirección al *socket*: `bind()` . . . 10
1.5.3. Conexión en *sockets* tipo *stream* . . . 15
1.5.4. Conexión en *sockets* tipo *datagram* . . . 19
1.5.5. Transferencia de información en *sockets* tipo *stream*: `write()` y `read()` . . . 19
1.5.6. Transferencia de información en *sockets* tipo *datagram*: `sendto()` y `recvfrom()` . . . 21
1.5.7. Cerrar la conexión: `close()` y `shutdown()` . . . 22
1.6. Ejemplo con *sockets* tipo *stream* . . . 24
1.6.1. Servidor . . . 24
1.6.2. Cliente . . . 25
1.7. Desarrollo de aplicaciones. Tipos de servidores . . . 27
1.7.1. Concurrencia real . . . 28
1.7.2. Concurrencia aparente . . . 31
1.8. Otras funciones interesantes . . . 37
1.8.1. Obtener el IP de un nodo: `gethostbyname()` . . . 38
1.8.2. Obtener el nombre de un nodo: `gethostbyaddr()` . . . 39
1.8.3. Obtener servicios y puertos por su nombre . . . 39
1.9. Análisis del interfaz de `sockets` . . . 41

Capítulo 1

Programación de aplicaciones distribuidas usando *sockets*

1.1. Introducción

En 1981 en la Universidad de California en Berkeley diseñaron para su sistema operativo, el BSD Unix, un interfaz para permitir a los programas acceder y comunicarse a través de una red de comunicaciones.

Ese interfaz, conocido como el *interfaz de sockets*, se incorporó a la versión 4.1 del sistema operativo. Como ese sistema operativo fue adoptado por varios fabricantes de estaciones de trabajo como Sun Microsystems Inc., Tektronix Inc. o Digital Equipment Corp., el interfaz de *sockets* estuvo disponible en gran cantidad de máquinas. El interfaz fue tan ampliamente aceptado que se convirtió en un estándar *de facto*.

El UNIX System V usa en su lugar el Transport Level Interface (TLI). No obstante el uso de los *sockets* está tan extendido que es habitual que cualquier implementación de UNIX disponga de este interfaz por compatibilidad con el BSD.

1.2. Implementación en el sistema operativo

Una de las ideas iniciales en el diseño de los *sockets* era utilizar las funciones que suministra el sistema operativo Unix siempre que fuera posible y, añadir nuevas llamadas al sistema si era difícil encajar los requerimientos de la programación en red con las funciones existentes en el operativo. (Consultar [1] para una explicación más detallada).

En Unix, cuando una aplicación desea realizar operaciones de entrada/salida, llama a la función `open` para crear un descriptor de fichero que se usará luego para acceder al fichero. El sistema operativo implementa los descriptores de ficheros como un array de punteros a estructuras internas de datos. Para cada proceso, el sistema mantiene una tabla de descriptores

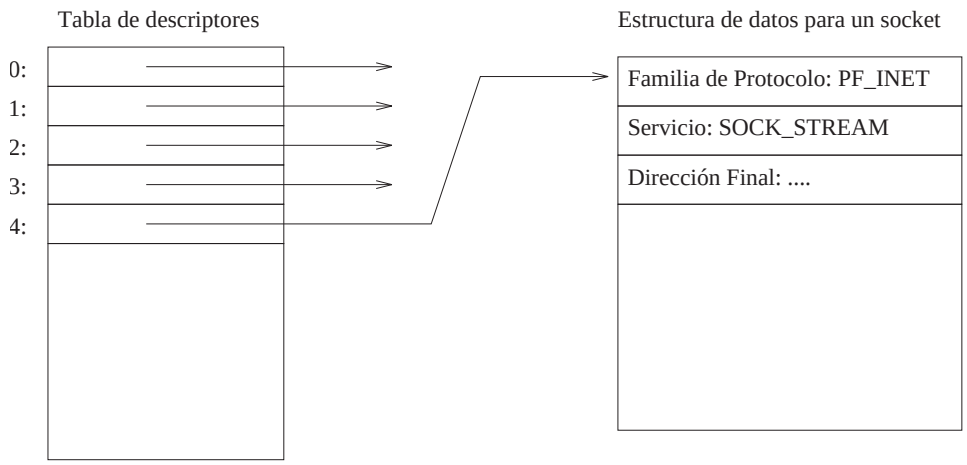

Figura 1.1: Estructura de datos de un *socket* en el sistema operativo

de ficheros separada. Cuando un proceso abre un fichero, el sistema coloca un puntero a las estructuras de datos internas de ese fichero en la tabla de descriptores de ficheros de ese proceso y le devuelve al proceso el índice del puntero en esa tabla. Ese índice es el *descriptor del fichero* y es lo único que el proceso tiene que recordar cuando quiere acceder al fichero.

El interfaz de los *sockets* añade una nueva abstracción para la comunicación a través de la red, el ***socket***. Cada *socket* activo se identifica por un entero denominado su ***descriptor de socket***. El sistema operativo Unix coloca los descriptores de *sockets* en la misma tabla de descriptores que los ficheros. De esta manera, una aplicación no puede tener un descriptor de fichero con el mismo valor que un descriptor de *socket*.

Para crear un descriptor de *socket* se le añade al sistema operativo una nueva llamada al sistema: la función `socket`. Con esta función se crea un *socket*. Cuando un proceso crea un *socket*, el sistema coloca un puntero a las estructuras internas de ese *socket* en la tabla de descriptores de ficheros de ese proceso y le devuelve al proceso el índice de esa tabla. Ese índice es el descriptor de *socket*. Para rellenar los detalles específicos del *socket* será necesario realizar llamadas a otras funciones del API.

En la figura 1.1 podemos ver una representación simplista de un *socket* en el interior del sistema operativo.

1.3. Conceptos de diseño

Un *socket*, desde el punto de vista funcional, se define como un punto terminal al que pueden *"enchufarse"* dos procesos para comunicar entre sí. Para que dos procesos pudieran comunicar hubo que dotar al sistema de una serie de funciones que permitieran a esos procesos acceder a los dispositivos

de red. Cuando se consideró cómo añadir funciones al sistema operativo para suministrar acceso a las comunicaciones, surgieron dos posibilidades:

- Definir funciones que soportaran específicamente el protocolo TCP/IP.
- Definir funciones que soportaran cualquier tipo de protocolo de comunicaciones y parametrizarlas cuando se quisiera utilizar TCP/IP.

En el momento del desarrollo del interfaz, el protocolo TCP/IP no estaba tan ampliamente divulgado como ahora y existían otras posibilidades para establecer comunicaciones entre dos máquinas. Por esta razón, los diseñadores optaron por la segunda opción: mantener la generalidad del interfaz.[1]

Al desvincular el interfaz de *sockets* de un protocolo de comunicaciones determinado, se hará necesario especificar ese protocolo cuando se usen esas funciones. De esta manera, cada vez que se quiera utilizar el interfaz de *sockets*, será necesario especificar:

Familia de Protocolo. Hay que indicar qué tipo de protocolo se va a utilizar para realizar las distintas comunicaciones. Los protocolos TCP/IP constituyen una única familia representada por la constante `PF_INET`. En el caso de comunicaciones entre procesos en la misma máquina usando el sistema de ficheros, tendríamos la familia de protocolos identificada como `PF_UNIX`.

Tipo de servicio. El interfaz permite seleccionar el tipo de servicio que se desea siempre y cuando el protocolo seleccionado sea capaz de suministrar distintos tipos de servicio. Aquí por *tipo de servicio* nos estamos refiriendo a cosas como *comunicación orientada a la conexión* o bien a *comunicación orientada a los datagramas*.

Familia de direcciones finales. Cada familia de protocolos especifica la *dirección final* de una comunicación de una manera distinta. La dirección final de una comunicación es el *"punto"* a través del cual un proceso envía o recibe datos. Por ejemplo en IP, una dirección final se especifica usando la dirección IP de la máquina y el número de puerto de protocolo que usará el programa. En el caso de usar la familia de protocolos UNIX la dirección final será el nombre de un fichero. El interfaz permite que las direcciones finales se puedan expresar con distintos formatos *aún dentro de la propia familia de protocolos*. Cada una de esas posibles representaciones correspondería a una *familia de direcciones*.

[1] Quizás la incorporación del interfaz de *sockets* y de todas las aplicaciones que se desarrollaron sobre dicho interfaz haya sido una de las principales causas de la enorme popularidad de TCP/IP en estos días. La otra quizás sea que era gratis.

1.3.1. Dirección final genérica

Para gestionar esta múltiple variedad de direcciones finales posibles[2], el interfaz define un formato de dirección final generalizada. En ese formato, una dirección final consta de dos elementos:

1. Una constante que identifica la familia de direcciones a utilizar.
2. La representación de la dirección final en esa familia de direcciones concreta.

En la práctica, ésta dirección final genérica que define el interfaz de los *sockets* se convierte en una estructura C con dos campos:

```
struct sockaddr {
    u_short sa_family; /* Familia de la dirección */
    char    sa_data[14]; /* Dirección final especifica
                         para esa familia */
};
```

Todas las funciones del interfaz de *sockets* que necesiten recibir como parámetro una dirección final, recibirán un puntero a una estructura de tipo `sockaddr`. En la práctica no todas las familias de direcciones encajan en esa estructura porque no todas usan 14 bytes para representar una dirección final. En esos casos es necesario utilizar otras estructuras más adecuadas para la familia de direcciones con la que se trabaje.

1.3.2. Utilización del API para TCP/IP

Vamos a ver con más detenimiento como se particularizan las generalizaciones del interfaz de *sockets* para el protocolo TCP/IP.

En el caso de la familia de protocolos el valor a indicar, como ya hemos visto, será la constante predefinida `PF_INET` (*"Protocol Family Internet"*).

En cuanto a la especificación del tipo de servicio deseado, los protocolos de la familia Internet solo admiten dos tipos de servicio: el orientado a conexión (TCP) y el orientado a los datagramas (UDP). Para identificar ambos tipos de servicio cuando se crea un *socket* se usan las constantes predefinidas `SOCK_STREAM` y `SOCK_DGRAM`, respectivamente.

Para el caso de la familia de direcciones, dentro de los protocolos de Internet sólo se utiliza un método para expresar las direcciones finales de una comunicación. Este mecanismo de direccionamiento identifica una dirección final por el par:

[2]Hay funciones en el interfaz, como `bind` que tienen que manejar una dirección final sin tener ni idea de cuál es el protocolo utilizado. Actualmente, el problema se resolvería con un tipo de dato `void *`, pero en el momento en el que se desarrolló el interface este tipo de dato no existía.

<dirección IP de la máquina, número de puerto de protocolo>

Este tipo de dirección final se identificará con la constante predefinida `AF_INET`[3](*"Address Family Internet"*). Para esta familia de direcciones (`AF_INET`) tendremos que usar una versión modificada de la estructura `sockaddr` denominada `sockaddr_in`, la cual tiene el siguiente aspecto:

```
struct sockaddr_in {
       short       sin_family;
       u_short     sin_port;
       struct in_addr sin_addr; /* En algunas máquinas es
                                   de tipo u_long */
       char        sin_zero[8];
};
```

El uso de esta estructura lo podremos ver en detalle en la sección 1.5.2.

1.4. Programación con el interfaz de *sockets*

En este documento sólo veremos como se programa usando los *sockets* que se comunican a través de Internet usando el protocolo TCP/IP. Otros *sockets* se comunican a través del kernel de UNIX, pero éstos sólo se usan para procesos corriendo en la misma máquina y no los tendremos en cuenta en este documento.

Para usar los *sockets* en un programa[4] escrito en lenguaje C, lo único que hay que hacer es abrir uno y escribir o leer en él mediante las funciones adecuadas que enseguida veremos. Estas funciones están en la librería `libsocket.a`, con la que tendremos que enlazar nuestros programas.

Las declaraciones de las funciones y las constantes predefinidas necesarias para trabajar con *sockets* se hallan en los ficheros:

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

La comunicación entre dos *sockets* depende del tipo al que pertenezcan estos *sockets*. Ambos deben tener el mismo tipo para que la comunicación sea posible. Estos tipos pueden ser:

[3]Aquí se produce una de las confusiones más típicas al trabajar con los *sockets*. El valor de la constante `PF_INET` y el de la constante `AF_INET` coinciden, por lo que muchos programadores las utilizan indistintamente. Aunque el programa funcione correctamente no es conveniente mantener el error.

[4]Tenemos una descripción más detallada y extensa en [1] y en [2].

- *Stream*. Los *sockets* de este tipo, una vez conectados, permanecen unidos hasta que se cierran los extremos. Este canal de comunicación es como un flujo. Los datos que se van volcando en un *socket* llegan directamente al *socket* del otro extremo en el mismo orden en que los hemos enviado. La dirección del otro *socket* sólo hay que suministrarla al hacer la conexión, no en cada envío. Es análogo a una comunicación telefónica en la que la dirección del otro interlocutor (su número de teléfono) se da sólo para iniciar la comunicación. Una vez iniciada, todo lo que hablamos por el teléfono llega automáticamente al otro interlocutor y en el mismo orden en que nosotros lo hemos pronunciado.

  Estos *sockets* son del tipo `SOCK_STREAM` y se comunican mediante el protocolo TCP (Transmission Control Protocol).

- *Datagram*. Los *sockets* de este tipo se reconectan cada vez que necesitamos enviar algo. Por tanto en cada envío hay que volver a especificar la dirección del otro *socket*. Además, no hay garantía de que el receptor reciba los paquetes en el mismo orden en el que los hemos enviado (ni siquiera hay garantía de que lo reciba: el paquete puede perderse en el camino debido a un fallo de la red y el emisor no es informado de ello para que lo reenvíe). Es análogo al sistema de correos: en cada carta es necesario escribir la dirección del destinatario y las cartas no llegan necesariamente en el mismo orden en que las enviamos, además si una carta se pierde, el emisor no es informado de ello por correos.

  Estos *sockets* son del tipo `SOCK_DGRAM` y se comunican mediante el protocolo UDP (User Datagram Protocol).

1.5. Uso de los *sockets*

Para comunicar dos programas escritos en lenguaje C mediante *sockets*, es necesario seguir los siguientes pasos, que tenemos ilustrados en las figuras 1.2 y 1.3:

1. Creación del *socket* (mediante la función `socket()`).

2. Asignar una dirección final al *socket*, una dirección a la que pueda referirse el otro interlocutor. Esto se hace con la función `bind()`.

3. En los *sockets* tipo *stream*, es necesario conectar con otro *socket* cuya dirección debemos conocer. Esto se logra ejecutando la función `connect()` para el proceso que actuará de cliente y las funciones `listen()` y `accept()` para el proceso que actuará como servidor.

4. Comunicarse. Esta es la parte más sencilla. En los *sockets* tipo *stream*, basta usar `write()` para volcar datos en el *socket* que el otro extremo

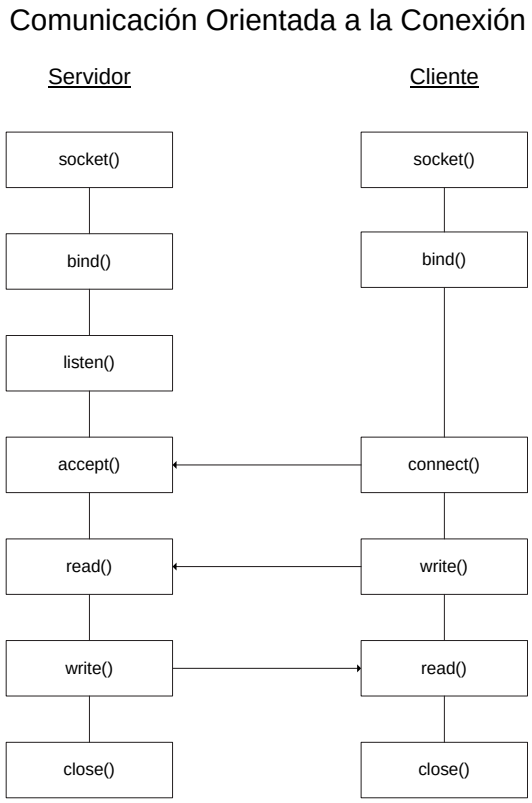

Figura 1.2: Secuencia de llamadas a funciones para una comunicación tipo *stream*

puede leer mediante la función `read()`, y a la inversa. En los *sockets* tipo *datagram* (Figura 1.3), el envío se realiza con la función `sendto()` y la recepción se realiza con `recvfrom()`. Ambas deben especificar siempre la dirección final del otro interlocutor.

5. Finalmente, cuando ya la comunicación se da por finalizada, ambos interlocutores deben cerrar el *socket* mediante la función `close()` o `shutdown()`.

A continuación veremos con un poco más de detalle estos pasos.

1.5.1. Creación del *socket*: `socket()`

La llamada a `socket()` es de la forma siguiente:

```
s = socket(fam_pro, tipo_ser, protocolo);
```

Siendo:

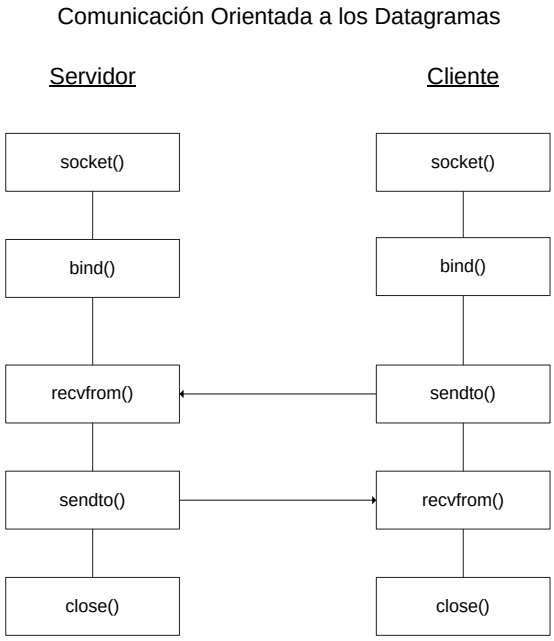

Figura 1.3: Secuencia de llamadas a funciones para una comunicación tipo *datagrama*

`s` Valor devuelto por la función. Ha de ser de tipo entero. Un valor negativo significa que ha ocurrido un error. Esto puede suceder por falta de memoria o por un valor erróneo en los parámetros de la función. En caso contrario, `s` será el descriptor del *socket*, descriptor que necesitaremos para las funciones siguientes.

`fam_pro` Aquí se especifica la familia de protocolos que se utilizará en la comunicación, es decir, si va a ser para comunicar procesos UNIX en la misma máquina o para comunicar procesos en diferentes (o la misma) máquinas a través de Internet. En el primer caso, este parámetro debe tomar el valor `PF_UNIX`, y en el segundo, el valor `PF_INET`. Estas constantes están predefinidas en el fichero `<sys/socket.h>`.

`tipo_ser` Aquí se especifica el tipo de servicio deseado, o dicho de otra forma, si el *socket* será de tipo *stream* (constante `SOCK_STREAM`) o de tipo *datagram* (`SOCK_DGRAM`). También existe una tercera posibilidad y es el tipo *raw* (`SOCK_RAW`, este último no se estudiará aquí).

`protocolo` En este campo se especifica que tipo de protocolo concreto se desea para el tipo de servicio especificado, en el caso de que la familia de protocolos escogida disponga de más de un protocolo para cada tipo de servicio. En el caso de Internet (IP), sólo se ofrece un protocolo para el servicio `stream` (TCP) y un protocolo para el servicio `datagram`

(UDP), por lo que lo mejor es poner un cero que significa que el sistema elija el protocolo más adecuado para el tipo solicitado.

Por ejemplo, para crear un *socket* de tipo *stream* para una comunicación a través de la red, la llamada sería:

```
s = socket( PF_INET, SOCK_STREAM, 0 );
```

Si la llamada a `socket()` ha retornado el valor `-1` significa que algo ha ido mal. En esta caso podemos comprobar la variable `errno` para saber que ha ocurrido. Los valores posibles son:

`EACCES` Se deniega el permiso para crear el *socket* del tipo solicitado.

`EMFILE` La tabla de descriptores del proceso está llena.

`ENOMEM` No hay suficiente memoria disponible para el usuario.

`EPROTONOSUPPORT` El protocolo especificado o el servicio solicitado no están soportados.

1.5.2. Asignar una dirección al *socket*: `bind()`

Aunque con `socket()` ya se ha creado un *socket*, lo único que hemos hecho ha sido crear las estructuras internas necesarias para mantener una comunicación. Ahora será necesario ir rellenando esa estructura interna. Lo primero, será asignar una dirección final local a este *socket* recién creado. Si no le asignamos una dirección final local, ningún otro proceso podrá conectarse a él, ya que como veremos, la función para conectar requerirá la dirección final del *socket* al que conectarse.

Un *socket* queda completamente definido cuando se suministran los tres parámetros
`<protocolo, dirección final local, dirección final remota>`
No pueden existir dos *sockets* con estos tres parámetros iguales. Particularizando para el caso de Internet que es el que nos interesa, las direcciones finales consisten en las direcciones IP de las dos máquinas conectadas más el número de un puerto de protocolo en cada máquina. Por lo tanto, un *socket* se nos convierte en la 5-tupla:

`<protocolo, dirección IP máquina local, número puerto de protocolo local, dirección IP máquina remota, número puerto de protocolo remoto>`.

Hasta que los cinco parámetros no estén fijados, la comunicación no es posible.

Como hemos visto, la función `socket()` crea un *socket*, y le asigna un `protocolo`, pero quedan sin asignar los restantes cuatro parámetros. Como

veremos enseguida, la función `bind()` asignará los parámetros `<dirección IP local, puerto protocolo local>`, y finalmente la función `connect()` (o bien `accept`) asignará los parámetros `<dirección IP remota, puerto protocolo remoto>`, con lo que el *socket* quedará completamente definido y podrá comenzar la comunicación.

La sintaxis de `bind()` es la siguiente:

```
retcod=bind(s, dir_fin, lon_dir_fin);
```

donde:

`s` es el descriptor del *socket* que queremos usar. Ha de ser un valor devuelto por la función `socket()`.

`dir_fin` es la dirección local y puerto local que queremos asignar al *socket*. Este parámetro es un puntero a una estructura de tipo `sockaddr`. Esta estructura en realidad no suele usarse tal cual, sino que en su lugar se usa una estructura de tipo `sockaddr_in` para las conexiones de la familia `AF_INET` o bien una estructura de tipo `sockaddr_un` para las conexiones de la familia `AF_UNIX`. Veremos con un ejemplo qué campos tienen estas estructuras y qué valores deben asignarse.

`lon_dir_fin` es el tamaño en bytes del parámetro anterior. Ya que como hemos dicho, el parámetro anterior puede ser de varios tipos diferentes según la familia del *socket*, es necesario para la función `bind()` conocer el tamaño de ese parámetro, ya que lo que ella recibe es tan sólo el puntero. Lo habitual es usar la macro `sizeof` para averiguar este tercer parámetro.

`retcod` Código de retorno de la función. Si el valor retornado es 0, la operación ha completado con éxito. Si el valor retornado es `-1` significa que ha ocurrido un error.

Cuando la llamada a `bind()` devuelve un valor negativo se puede consultar la variable `errno` para comprobar los motivos del fallo. Las razones pueden ser:

`EBADF` No se ha especificado un descriptor válido.

`ENOTSOCK` No se ha especificado un *socket*. Se ha usado un descriptor de fichero.

`EACCES` La dirección especificada está protegida y el programa no tiene acceso a ella.

`EADDRINUSE` La dirección especificada ya está en uso.

`EADDRNOTAVAIL` La dirección especificada no está disponible en la máquina (p. ej. una dirección IP que no existe en la máquina local).

`EINVAL` El *socket* ya tiene una dirección asignada.

La estructura `sockaddr_in`

La función `bind()` recibe como segundo parámetro un puntero a una estructura de tipo `struct sockaddr`. En la práctica lo que se le pasa es un puntero a una estructura `sockaddr_in`, por lo que es necesario hacer un *cast* al tipo `struct sockaddr`.

El tipo `struct sockaddr_in` está definido en `<netinet/in.h>` y es una adaptación de la dirección final genérica vista en 1.3.1 al caso de direcciones finales Internet. Su aspecto es como sigue:

```
struct sockaddr_in {
       short       sin_family;
       u_short     sin_port;
       struct in_addr sin_addr; /* En algunas máquinas es de tipo
                                   u_long */
       char        sin_zero[8];
};
```

donde:

`sin_family` es la familia del *socket*, `AF_INET` en nuestro caso, ya que estamos interesados sólo en *sockets* que comuniquen a través de Internet.

`sin_port` es el puerto de protocolo local. Este es un número que podemos elegir libremente, siempre que no esté en uso ese mismo puerto por otro proceso. Los servicios de red del UNIX (telnet, ftp, etc...) utilizan números de puerto bajos. Si usamos números de puerto por encima de 1000 no tendremos conflictos (salvo con otros usuarios que estén haciendo programas de red y hayan elegido el mismo puerto). Se puede especificar el valor cero para el puerto, con lo que el sistema elegirá el primero que quede libre. Esto no es una buena idea, ya que el otro interlocutor debe conocer el número del puerto para hacer la conexión.

`sin_addr` Ésta, a su vez, es una estructura pero bastante sencilla. Tan sólo:

```
struct in_addr {
       unsigned long s_addr;
}
```

siendo `s_addr` la dirección IP de nuestra máquina. Lo normal es poner aquí la constante predefinida `INADDR_ANY`, ya que una misma máquina puede tener varias direcciones IP (si está conectada a varias redes simultáneamente). Al poner `INADDR_ANY` estamos dejando indeterminado el dato de `<dirección local>` con lo que nuestro programa podrá aceptar conexiones de todas las redes a las que pertenezca la máquina. Lo habitual es que cada máquina pertenezca sólo a una red, por lo que poner `INADDR_ANY` es igual a poner la dirección IP de la máquina en cuestión, con la ventaja de que no necesitamos conocer esta dirección, y además el programa es más portable, ya que el mismo código fuente compilado en otra máquina (con otra dirección IP) funcionaría igual. No obstante, si queremos, podemos dar esta dirección IP directamente, aunque corremos el riesgo de generar problemas de enrutamiento en aquellas máquinas que están conectadas a varios interfaces de red.

El tipo del campo `s_addr` es un entero largo (4 bytes) en el que estos 4 bytes son los 4 números de una dirección IP. La función `inet_addr()` nos ayuda a traducir de la forma habitual en que se escribe la dirección (por ejemplo 156.34.41.70) al tipo requerido para el campo `s_addr`. Por ejemplo:

```
minodo.sin_addr.s_addr = inet_addr("156.34.41.70");
```

En los casos en que no sepamos la dirección IP sino su nombre Internet (por ejemplo `sirio.atc.uniovi.es`) tendremos de utilizar la función `gethostbyname()` de la que veremos un ejemplo en la página 38.

`sin_zero` Es un campo de relleno que no utilizaremos.

Un ejemplo:

```
#include<sys/types.h>
#include<netinet/in.h>
#include<sys/socket.h>
...
int  s;
struct sockaddr_in local;
...
    s = socket(PF_INET, SOCK_STREAM, 0);
    ... /* Comprobacion de errores */
    local.sin_family=AF_INET;
    local.sin_port=htons(15001);
    local.sin_addr.s_addr = htonl(INADDR_ANY);
    bind(s, (struct sockaddr *) &local, sizeof local);
...
```

Observar las siguientes peculiaridades:

- Es necesario hacer un *cast* en el segundo parámetro de `bind()`.
- El tercer parámetro de `bind()` es el tamaño en bytes del segundo parámetro. Lo más cómodo es usar la macro `sizeof` que nos devuelve este valor.
- No hemos hecho comprobación de errores. Habría que comprobar si el valor devuelto por `bind()` es negativo, en cuyo caso habría un error.
- El puerto que hemos usado (17001) lo hemos elegido arbitrariamente, pero deberá ser conocido también por el otro interlocutor.
- Las funciones `htons()` y `htonl()` se usan para traducir cantidades de más de 1 byte entre el formato utilizado por nuestra computadora y el utilizado por la red. Existen otras dos funciones para la traducción inversa. El significado de todas estas funciones es:

  `htons()` Host to Network Short. Convierte un valor corto (2 bytes) del formato usado por el host (nuestra máquina) al usado por la red.

  `htonl()` Host to Network Long. Convierte un valor largo (4 bytes) del formato usado por el host (nuestra máquina) al usado por la red.

  `ntohs()` Network to Host Short. Convierte un valor corto (2 bytes) del formato usado por la red al usado por el host (nuestra máquina).

  `ntohl()` Network to Host Long. Convierte un valor largo (4 bytes) del formato usado por la red al usado por el host (nuestra máquina).

  Es necesario hacer estas conversiones porque cada máquina puede tener su propio criterio para ordenar los bytes de estas cantidades (*Little Endian* o *Big Endian*), pero el criterio de la red es único, en concreto *Big Endian*.

La nueva estructura de direcciones

Actualmente se está siguiendo un proceso de sustitución de la versión actual del protocolo de red IPv4 hacia la versión IPv6. En esta nueva versión, las direcciones de las máquinas ya no se representan por 32 bits, sino que se utilizan 128 bits. La estrucura que acomoda las direcciones debe, por tanto, modificarse también para adaptarse a este cambio.

La nueva estrucura se denomina `sockaddr_in6` y tiene los siguientes campos:

```
struct sockaddr_in6 {
       uint8_t     sin6_len;  /* longitud de la estructura */
       short       sin6_family; /* AF_INET6  */
       u_short     sin6_port;
       uint32_t    sin6_flowinfo; /* Etiqueta de prioridad y flujo */
       struct in6_addr sin6_addr; /* Dirección de máquina IPv6 (128 bits) */
};
```

1.5.3. Conexión en *sockets* tipo *stream*

Esta operación se hace en los *sockets* de tipo *stream*. En los de tipo *datagram* no es necesario conectar para poder empezar a enviar ya que en estos *sockets* no hay una conexión mantenida.

La conexión se realiza de forma asimétrica. Es decir, el servidor y el cliente no realizan las mismas operaciones para conectarse. Generalmente, el servidor efectúa una llamada a la función `listen()` que le deja preparado para admitir conexiones. Pone el *socket* en modo "pasivo". Lo usual es que a continuación llame a la función `accept()` para bloquearse hasta que un cliente "llame a su puerto". En ese momento los cinco parámetros del *socket* estarán completos: `protocolo`, `dirección local` y `puerto local` son los que se hayan asignado con `bind()`, mientras que `dirección remota` y `puerto remoto` serán los del nodo que acaba de conectarse. A partir de este momento puede comenzar la transferencia de datos mediante llamadas a `read()` y `write()`.

En el cliente, en cambio, tras la llamada a `bind()` (que fija los parámetros locales) se llama a la función `connect()` que intenta fijar los parámetros remotos, siempre que la máquina remota acepte la conexión. Si no la acepta, puede deberse a que no hay un servidor atendiendo en el puerto indicado o que el *socket* no es del tipo especificado.

Veamos más en detalle estas funciones:

El servidor: `listen()` y `accept()`

Después de ejecutar `bind()` es necesario llamar a `listen()` para que el *socket* sea marcado por el sistema como listo para recibir. Esta llamada tiene una función doble:

1. Pone el *socket* en modo *"pasivo"* a la espera de conexiones.
2. Fija el tamaño máximo de la cola de peticiones para ese *socket*.

La sintaxis es:

```
retcod=listen(s, num_cola);
```

donde

`s` es el descriptor del *socket* (valor devuelto por `socket()`).

`num_cola` es el máximo número de peticiones de conexión que pueden esperar en la cola. Lo normal es que el servidor esté en un bucle aceptando conexiones de diferentes clientes. Si hay una conexión ya establecida cuando otro cliente intenta conectar, este segundo cliente es puesto en una cola, y la próxima vez que el servidor llame a `accept()` se establecerá la conexión con este cliente. Si el número de clientes en la cola es igual a `num_cola`, y un nuevo cliente intenta conectarse, será rechazado[5]. El tamaño máximo que puede tener la cola de peticiones vendrá determinado para cada sistema por el valor de la constante `SOMAXCONN`[6].

`retcod` Es el valor retornado por la función. Un valor 0 indica que se ha completado con éxito. El valor `-1` indica que ha ocurrido algún error.

Una vez que se ejecuta esta función, el servidor debe bloquearse esperando hasta que un cliente intente la conexión. Esto lo hace ejecutando `accept()`, cuya sintaxis es:

```
n_sock = accept(s, quien, l_quien);
```

donde

`n_sock` es el valor devuelto por la función. Es un descriptor de fichero (tipo `int`) que deberemos utilizar después en las llamadas a `read()` o `write()` para transmitir la información a través del *socket*. Si `accept()` devolviera el valor `-1` se trataría de un error en la conexión.

En realidad, `n_sock` es el descriptor de otro *socket*. Podemos pensar que `s` es el *socket* a través del cual se "escuchan" las peticiones y que una vez aceptadas, `n_sock` es el *socket* por el que se comunicarán los procesos. De hecho, `s` está de nuevo disponible para otra conexión y si, inmediatamente después de `accept()` hiciéramos otro `accept()`, podríamos tener otro cliente enganchado a otro *socket* (son *sockets*

[5] Realmente, como se explica en [3], la cola de espera está formada por otras dos subcolas: (1) la cola de conexiones incompletas y (2) la cola de conexiones completadas. El número indicado en la llamada a `bind` especifica el tamaño total de ambas colas

[6] Existe un ataque a los programas servidores basado en el tamaño de la cola de peticiones. El ataque consiste en enviar grandes cantidades de peticiones de conexión al puerto en el que está escuchando un servidor, *inundar* al servidor con peticiones (*"SYN flooding"*). Al mismo tiempo, para garantizar que esas peticiones llenen la cola de peticiones y hagan que se rechazen nuevas conexiones, el cliente oculta su dirección original (*"IP spoofing"*). De esta manera se evita que el servidor atienda a sus clientes legítimos

diferentes ya que las direcciones y puertos remotos son diferentes). Lógicamente, habría que guardar el valor devuelto por `accept()` en variables diferentes para cada llamada para poder tener acceso a los descriptores de ambos *sockets*.

No obstante, la forma de servir a varios clientes simultáneamente no es haciendo varios `accept()`, ya que esto bloquearía al servidor si no hubiera clientes tratando de conectar, sino usando la función `select()` que veremos en la sección 1.5.3.

`s` es el descriptor del *socket* (valor devuelto por `socket()`) en el que estamos esperando las conexiones.

`quien` es un puntero a una estructura de tipo `sockaddr` en la que se nos devuelve información sobre el cliente que se ha conectado. En concreto, esta estructura suele ser en realidad de tipo `sockaddr_in` cuyos campos ya hemos visto en la página 12, y de la que nos pueden interesar la dirección IP y el número de puerto del cliente.

No hay forma de especificar en la llamada a `accept()` que sólo queremos admitir conexiones con otra dirección dada; `accept()` aceptará conexiones de cualquier proceso que intente conectar con la dirección de nuestro *socket*. El programador puede examinar los valores devueltos en la estructura `quien` y si el nodo conectado no es de su agrado, cerrar de nuevo la conexión.

Si no nos interesa la información sobre quién se ha conectado, podemos pasar `NULL` como segundo parámetro de `accept()`.

`l_quien` es un puntero a entero a través del cual se nos devuelve el tamaño en bytes del parámetro anterior. Pondremos `NULL` si hemos puesto `NULL` en el parámetro anterior.

Veremos un ejemplo de uso en la página 24. Tras la llamada con éxito a esta función, la conexión está establecida. Ya podemos usar `n_sock` con `read()` y `write()`.

En el caso de que `accept()` devolviera el valor -1, tendríamos que comprobar las razones del fallo. Éstas pueden ser:

`EOPNOTSUPP` El *socket* no es del tipo `SOCK_STREAM`.

`EBADF` El primer argumento no especifica un descriptor válido.

`ENOTSOCK` No se ha especificado un *socket*. Se ha usado un descriptor de fichero.

`EWOULDBLOCK` El *socket* está marcado como no bloqueante y no hay conexiones esperando para ser aceptadas.

`ENOMEM` No hay suficiente memoria de usuario disponible para completar la operación.

El cliente: `connect()`

El cliente, una vez ejecutado `bind()`, debe intentar establecer conexión con un servidor. Para ello debe conocer la dirección IP del servidor y el puerto en el cual está escuchando. Si no se conocen estos datos no podrá establecerse la conexión. La dirección IP puede averiguarse mediante la función `gethostbyname()` si se conoce el nombre del nodo en el cual se está ejecutando el servidor. El número de puerto se supone ya conocido bien porque estamos consultando a un servidor "bien conocido" (cuyo número está documentado en manuales) o bien porque estamos conectando con un servidor programado por nosotros mismos.

La sintaxis de `connect()` es:

```
retcod=connect(s, servidor, l_servidor);
```

donde:

`s` es el *socket* que queremos conectar (es decir, el valor devuelto por `socket()`).

`servidor` es un puntero a una estructura del tipo `sockaddr` en cuyos campos se especificará la dirección del otro interlocutor del *socket*. En las comunicaciones a través de Internet este parámetro es en realidad de tipo `(struct sockaddr_in *)`, y contiene la dirección IP y el puerto del interlocutor.

`l_servidor` es el tamaño en bytes del parámetro anterior.

`retcod` Es el código de retorno de la función. Si el valor es cero, significa que todo ha ido con normalidad. Si el valor retornado ha sido `-1`, quiere decir que se ha producido algún error.

El paso más difícil que hay que dar antes de llamar a `connect()` es rellenar correctamente todos los campos de la estructura `sockaddr_in` que hay que pasarle como parámetro. Veamos ahora cómo debe hacerse esto:

`servidor.sin_family` debe contener el valor `AF_INET` para las comunicaciones a través de Internet, que son el caso que nos ocupa.

`servidor.sin_port` debe contener el número del puerto al que queremos conectarnos. Este ha de ser el puerto en el que el servidor está escuchando (al cual se ha "ligado" el servidor en su llamada a `bind()`). El número de puerto debe ser convertido al formato red mediante la función `htons()` (ver página 14).

`servidor.sin_addr.s_addr` Si conocemos la dirección IP de la máquina en la que se halla el servidor, podemos llenar este campo con ella traduciéndolo mediante la función `inet_addr()` (ver página 13). Si sólo conocemos el nombre, debemos llamar a `gethostbyname()` para que nos averigüe la dirección IP. En la página 38 veremos en detalle esta función.

Si `connect()` devuelve un valor negativo quiere decir que se ha producido un error al intentar la conexión. Si se consulta la variable `errno` es posible obtener alguna indicación sobre el motivo. Algunas de las posibles razones para que `connect()` falle son:

`EBADF` El primer argumento no especifica un descriptor válido.

`ENOTSOCK` El primer argumento no es un descriptor de *socket*.

`EAFNOSUPPORT` La familia de direcciones especificada para la conexión remota no puede ser usada con este tipo de *socket*.

`EADDRNOTAVAIL` La dirección final remota no está disponible.

`EISCONN` El *socket* ya está conectado.

`ETIMEDOUT` (Sólo TCP) El protocolo ha sobrepasado el tiempo de espera sin conseguir la conexión.

`ECONNREFUSED` (Sólo TCP) Conexión rechazada por la máquina remota.

`EADDRINUSE` La dirección especificada ya está en uso.

1.5.4. Conexión en *sockets* tipo *datagram*

No existe el concepto de conexión en este tipo de *sockets*. Por tanto en ellos no se deberían utilizar las funciones `listen()`, `accept()` ni `connect()`. Es posible utilizar ésta última función en algunos casos particulares, como se explica en la página 21.

En estos *sockets*, una vez que el cliente y el servidor han ejecutado `bind()` para asignar una dirección final a sus *sockets*, pueden comenzar a enviar y recibir datos a través de ellos. En cada envío o recepción debe indicarse la dirección del otro interlocutor. Esto se hace mediante las funciones `sendto()` y `recvfrom()` que veremos en la sección 1.5.6.

1.5.5. Transferencia de información en *sockets* tipo *stream*: `write()` y `read()`

Una vez que la conexión quedó establecida como se explicó en el punto 1.5.3, tenemos un descriptor de *socket* (`n_sock`) que utilizamos como un

descriptor de fichero para las funciones `read()` y `write()`. Recordemos la sintaxis de estas funciones:

```
retcod=write(socket, buffer, l_buffer);
```

donde:

`socket` es el descriptor de *socket* devuelto por `accept()` en el servidor o el completado a través de `connect()` en el cliente.

`buffer` es un puntero a carácter que apunta a los datos que queremos transmitir.

`l_buffer` es un entero que indica la cantidad de bytes a transmitir.

`retcod` Si la función ha ejecutado correctamente, será el número de bytes transferidos. En caso contrario, devolverá el valor `-1`.

Si el número de bytes enviado es menor al número de bytes que se querían enviar, será necesario volver a escribir en el *socket* a partir del punto en que se haya detenido la transmisión, hasta que hayamos enviado la cantidad de bytes esperada. Si coincide el número de bytes a enviar (`l_buffer`) con el número de bytes devuelto por `write()` significa que los datos han llegado a su destino.

```
retcod=read(socket, buffer, l_buffer);
```

donde:

`socket` es el descriptor del *socket* devuelto por `accept()` en el servidor o por `connect()` en el cliente.

`buffer` es un puntero a carácter que apunta a la zona donde se dejarán los datos leídos del *socket*.

`l_buffer` es el tamaño de la zona que tenemos reservada para leer datos del *socket*. Si llegan más datos de los que caben en nuestro buffer, quedarán a la espera de la siguiente llamada a `read()`.

`retcod` La función devuelve un valor negativo si ocurre un error durante la recepción. Si no hay error devuelve el número de bytes leídos. Si este número es igual a cero, significa que la conexión ha sido cerrada por el otro interlocutor (es como si hubiéramos leído un fin de fichero).

En el caso de que `read()` devuelva el valor cero, la comunicación puede darse por terminada y podemos cerrar el *socket*.

Si el número de bytes leídos es menor al número de bytes esperados será necesario volver a leer del *socket* hasta que hayamos recibido la cantidad de bytes esperada. El protocolo de transmisión (TCP) no respeta los límites entre los registros lógicos que queramos transferir. El protocolo va enviando los bytes según va llenando los buffers del sistema por lo que puede enviar la información en "paquetes" que no encajan con los "pedazos" lógicos de nuestro programa.

1.5.6. Transferencia de información en *sockets* tipo *datagram*: `sendto()` y `recvfrom()`

En este caso debemos usar las funciones `sendto()` y `recvfrom()` cuyas sintaxis son:

```
retcod=sendto(s, buffer, l_buffer, flags, destino, l_destino);
retcod=recvfrom(s, buffer, l_buffer, flags, origen, l_origen);
```

donde:

`s` es el descriptor del *socket* devuelto por `socket()`.

`buffer` es un puntero a la zona de memoria donde están los datos a enviar o donde encontraremos los datos recibidos.

`l_buffer` es la cantidad de bytes a enviar (o el tamaño del buffer de recepción).

`flags` es un entero que indica opciones avanzadas. En nuestro caso podemos dejar simplemente el valor cero.

`destino` es un puntero a una estructura de tipo `sockaddr` que contiene la dirección a donde enviamos los datos. En comunicaciones a través de Internet, esta estructura será en realidad de tipo `sockaddr_in` y contendrá la dirección IP y el puerto de destino. Podemos usar `gethostbyname()` (ver página 38) o `inet_addr()` (página 13) para rellenar el campo IP, lo mismo que para los *sockets* tipo *stream*.

`l_destino` es el tamaño en bytes del parámetro anterior.

`origen` es un puntero a una estructura de tipo `sockaddr` que contiene la dirección de la que provienen los datos. Análogo al parámetro `destino` en la función `sendto()`. Siempre se aceptarán datos de cualquier nodo que los envíe a nuestro puerto. Al retornar la función la estructura contendrá la dirección del nodo del cual han venido los datos y se

podrá realizar algún tipo de control de acceso. No se pueden rechazar mensajes *a priori*.

`l_origen` es un puntero a un entero en el que se devolverá el tamaño en bytes del parámetro anterior.

`retcod` En el caso de que haya habido un error en la llamada a alguna de las funciones, valdrá `-1`. En el caso de que hayan funcionado correctamente, `retcod` será el número de bytes recibidos o enviados correctamente.

Ambas funciones retornan un valor negativo en caso de error, aunque los errores no se informan de forma síncrona. Es decir, si `sendto()` nos devuelve el valor -1, es que ha habido un error en el envío pero no necesariamente en la llamada que acabamos de hacer. Para entenderlo mejor, pensemos por ejemplo en un cliente que envía una serie de mensajes a distintos servidores y uno de ellos no se está ejecutando. El sistema de la máquina del cliente recibirá una información de que el mensaje no se ha podido repartir, pero ese error no se podrá transmitir al programa cliente ya que el sistema no tiene forma de pasar la dirección de la máquina que falló al proceso. Desde el punto de vista del cliente, el error pudo ocurrir en cualquier llamada a `sendto()`. En caso de error, tampoco se puede afirmar que los datos no hayan llegado correctamente a su destino: el error pudo haber sido local.

Si `sendto()` retorna el valor `-1` y la variable `errno` contiene el valor `EMSGSIZE`, significa que el tamaño del mensaje a enviar es demasiado grande. En las comunicaciones orientadas a los *datagramas* (sin conexión) los mensajes se envían uno en cada datagrama.

Es posible realizar la transferencia de información en *sockets* de tipo *datagram* usando las funciones `write()` y `read()`. En este caso se debe hacer una llamada previa a `connect()`. Esta llamada no establecerá ningún tipo de conexión, simplemente "rellenará" los valores de la dirección final de destino que se usará con la función `write()` o la dirección final de origen desde la que admitirá mensajes `read()`. Cada vez que se use esta función se enviarán los datos a la dirección especificada en `connect()`. Cuando se "conecta" un *socket* de tipo datagrama, ese *socket* no se podrá utilizar para enviar información a destinos distintos ni se recibirán datos de orígenes distintos al indicado en la llamada a `connect()`. Como ventaja, los *sockets* "conectados" sí reciben mensajes de error puesto que ahora, si un mensaje no llega a su destino, sí se tiene perfectamente localizados los parámetros de la comunicación.

1.5.7. Cerrar la conexión: `close()` y `shutdown()`

En los *sockets* tipo *stream* cuando los interlocutores no van a intercambiar más información conviene cerrar la conexión. Esto se hace mediante la función `close()`. Observar que el servidor cerrará el *socket* por el cual "dialogaban" (el que llamamos `n_sock`), pero no cerrará el *socket* por el que recibe

las peticiones de conexión (el que llamamos `s`) si quiere seguir aceptando más clientes.

Cuando un proceso se "mata" (recibe una señal `SIGTERM`) o cuando sale normalmente con una llamada a `exit()`, todos los *sockets* se cierran automáticamente por el sistema operativo.

Si cuando cerramos un *socket* aún quedaban en él datos por transmitir, el sistema operativo nos garantiza que los transmitirá todos antes de cerrar. Si tras varios intentos no lo logra (por ejemplo, porque el otro interlocutor ha cerrado ya o porque hay un fallo en la red) cerrará finalmente el *socket* y los datos se perderán.

La sintaxis de `close()` es muy sencilla:

```
retcod=close(socket);
```

donde:

`socket` es el descriptor del *socket* que queremos cerrar. Normalmente se tratará del valor devuelto por `accept()` en el servidor o por `connect()` en el cliente. El *socket* `s` inicial no suele ser cerrado explícitamente, sino que se deja que el sistema lo cierre cuando nuestro proceso muere.

`retcod` Valor de retorno. Si es 0 la operación se ha completado con éxito. Si es `-1` se ha producido algún tipo de error.

Para evitar pérdidas de datos a causa del cierre precipitado de conexiones, muchas veces es mejor cerrar las conexiones de manera parcial usando la función `shutdown()`. Ésta función tiene la siguiente sintaxis:

```
retcod=shutdown(socket, como);
```

donde:

socket Es el descriptor del *socket* que queremos cerrar.

`retcod` Valor de retorno. Si es 0 la operación se ha completado con éxito. Si es `-1` se ha producido algún tipo de error.

como Es un valor entero que indica el modo en el que queremos cerrar la conexión. Tenemos las siguientes posibilidades:

- Como = 0. No se realizarán más lecturas en el *socket*. No se esperan más datos.
- Como = 1. No se realizarán más escrituras en el *socket*. No se enviarán más datos.

- Como = 2. No se realizará ni entradas ni salidas a través del *socket*. Es equivalente a `close()`.

Tanto para `close()` como para `shutdown()` si devuelven un valor negativo significa que se ha producido un error.

Los *sockets* de tipo *datagram*, normalmente no se cierran de forma explícita ya que no tienen una conexión permanentemente establecida.

1.6. Ejemplo con *sockets* tipo *stream*

A continuación planteamos el ejemplo más sencillo posible. El servidor espera a que cualquier cliente se conecte con él. Cuando esto ocurre, lee datos de la conexión, los muestra por pantalla y se desconecta de ese cliente, quedando listo para admitir otro. El cliente simplemente se conecta al servidor y le envía una cadena de texto tras lo cual se desconecta.

El puerto del servidor, que debe ser conocido por el cliente, será el 15002. El servidor corre en la máquina llamada `"sirio"`. Los clientes pueden conectarse a él desde cualquier máquina.

1.6.1. Servidor

```
/*   servidor.c
*/
#include<sys/types.h>
#include<netinet/in.h>
#include<sys/socket.h>
#include<stdio.h>

/* Puerto del servidor, al que debe conectarse el cliente */
/* es un valor elegido arbitrariamente */
#define PUERTO 15002

main()
{
    int s;                      /* El socket de conexion */
    int n_sock;                 /* El socket de datos */
    struct sockaddr_in local;   /* Direccion local */
    struct sockaddr_in cliente; /* Direccion del cliente */
    int    l_cliente;               /* Tamano de la estructura anterior */
    char   buffer[81];          /* Para recibir los datos */
    int    leidos;              /* Numero de datos recibidos */

    /* Creacion del socket */
    s = socket(PF_INET, SOCK_STREAM, 0);
    /* comprobacion de errores */
    if (s<0) {
        perror("creando socket:");
        exit(1);
    }
```

```
    /* Asignacion de direccion al socket. Le asignaremos el puerto
       pero como direccion IP daremos INADDR_ANY */
    local.sin_family=AF_INET;
    local.sin_port=htons(PUERTO);
    local.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(s, (struct sockaddr *) &local, sizeof local)<0) {
        perror("asignando direccion:");
        exit(2);
    }
    /* Empezamos a aceptar conexiones */
    listen(s, SOMAXCONN);
    /* Y entramos en un bucle infinito para servir a los clientes
       que vayan llegando */
    while (1) {
        printf("Esperando nueva conexion...\n");
        l_cliente=sizeof cliente;
        n_sock = accept(s, (struct sockaddr *) &cliente, &l_cliente);
        if (n_sock<0) {
            perror("aceptando:");
        }
        /* Aceptada la conexion, en "cliente" tenemos la direccion
           del cliente que se nos ha conectado. En este caso ignoramos
           esta informacion */
        else {
            /* Inicializar el buffer a ceros */
            memset(buffer, 0, sizeof buffer);
            /* Leemos datos. Cuando "leidos"==0 es que el cliente ha
               cerrado la conexion */
            while((leidos = read(n_sock, &buffer, sizeof buffer))!=0) {
                if (leidos<0)
                    perror("leyendo:");
                else printf("Cadena recibida -> %s\n", buffer);
                /* Volver a poner buffer a ceros */
                memset(buffer, 0, sizeof buffer);
            } /* El cliente ha cerrado */
            printf("Conexion terminada. \n");
            close(n_sock);
        }
    } /* Fin del bucle infinito */
    /* Aqui habria que cerrar el socket s. De todas formas, esta linea
    nunca se ejecutara debido al bucle infinito, asi que dejaremos que
    sea el sistema quien cierre s cuando el proceso muera por un Ctrl-C
    o por un kill */
}
```

1.6.2. Cliente

```
/*   cliente.c
*/
#include<sys/types.h>
#include<netinet/in.h>
#include<sys/socket.h>
#include<stdio.h>
```

```
#include<netdb.h>
/* Puerto del servidor, al que debe conectarse el cliente */
/* es un valor elegido arbitrariamente */
#define PUERTO 15002
#define DATOS "Un, dos, tres... probando, probando . . ."

main()
{
    int s;                      /* El socket de conexion */
    struct sockaddr_in local;   /* Direccion local */
    struct sockaddr_in serv;    /* Direccion del servidor */
    struct hostent *servidor;   /* para gethostbyname() */

    /* Creacion del socket */
    s = socket(PF_INET, SOCK_STREAM, 0);
    /* comprobacion de errores */
    if (s<0) {
        perror("creando socket:");
        exit(1);
    }
    /* Asignacion de direccion al socket. En el caso del cliente
       no nos importa su numero de puerto local, asi que podemos
       dejar que lo elija el sistema, dandole el valor cero.
       como direccion IP daremos INADDR_ANY */
    local.sin_family=AF_INET;
    local.sin_port=htons(0);
    local.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(s, (struct sockaddr *) &local, sizeof local)<0) {
        perror("asignando direccion:");
        exit(2);
    }
    /* En realidad, no es necesario ejecutar bind() en el cliente
       si intentamos directamente hacer connect() sin haber hecho
       antes bind(), el sistema nos asignara automaticamente una
       direccion local elegida por el */

    /* Ahora vamos a conectar. Necesitamos la direccion IP del servidor
       la cual obtenemos con una llamada a gethosbyname() */
    servidor = gethostbyname("sirio.atc.uniovi.es");
    if (servidor==NULL) {
        fprintf(stderr, "sirio: nodo desconocido.\n");
        exit(2);
    }
    memcpy(&serv.sin_addr.s_addr, servidor->h_addr, servidor->h_length);
    /* Los restantes campos de serv no son problema */
    serv.sin_family = AF_INET;
    serv.sin_port = htons(PUERTO);
    /* Y ahora, conectar */
    if (connect(s, (struct sockaddr *) &serv, sizeof serv)<0) {
        perror("conectando:");
        exit(3);
    }
    /* Si hemos llegado hasta aqui, la conexion esta establecida */
    if (write(s, DATOS, sizeof DATOS)<0) {
```

```
        perror("escribiendo el socket:");
        exit(3);
    }
    /* Todo bien. Podemos cerrar el socket y terminar */
    close(s);
    exit(0);
}
```

1.7. Desarrollo de aplicaciones. Tipos de servidores

En la lección anterior, al hablar de la arquitectura cliente/servidor, se habló de la posibilidad de implementar los servidores de dos formas distintas:

- Servidores con estado, que son aquellos en los que el servidor almacena algún tipo de información sobre anteriores conexiones de un cliente.
- Servidores sin estado, que son aquellos en los que la relación entre el cliente y el servidor depende únicamente de la petición en curso del cliente.

Teniendo en cuenta la tecnología involucrada en el desarrollo de aplicaciones distribuidas utilizando el interfaz de *sockets*, podríamos hacer nuevas clasificaciones entre las posibles implementaciones de los servidores. Así, si nos fijamos en el protocolo utilizado para realizar las comunicaciones, podemos clasificar a los servidores como:

Servidores orientados a la conexión Son aquellos en los que la comunicación entre el cliente y el servidor se realiza utilizando el protocolo `TCP`.

Servidores orientados a los datagramas La comunicación entre el cliente y el servidor se realiza utilizando el protocolo `UDP` de `IP`.

Sin embargo, si nos fijamos en la forma en la que el servidor atiende a sucesivos clientes, podemos hacer una clasificación mucho mas interesante:

Servidores iterativos Un servidor decimos que es iterativo cuando atiende a los clientes de uno en uno. Cuando un servidor iterativo está atendiendo a un cliente y se recibe una solicitud de servicio de otro cliente, el sistema mantiene a este último en una cola de espera hasta que el servidor termina de atender al primero de ellos. Los servidores iterativos son adecuados cuando el tiempo de servicio no es elevado. Esto es, cuando el tiempo que emplea un servidor en atender una petición cualquiera no supera un cierto límite dependiente de la aplicación, el sistema y la arquitectura sobre la que se ejecuta el programa servidor.

Servidores concurrentes Decimos que un servidor es concurrente cuando es capaz de atender peticiones de varios clientes simultáneamente. La concurrencia puede ser *aparente* o *real* según haya un único proceso servidor atendiendo a múltiples clientes, que sería el caso de la concurrencia aparente; o bien tengamos varios procesos atendiendo cada uno de ellos a un cliente diferente (concurrencia real).

1.7.1. Concurrencia real

Para conseguir la concurrencia real en nuestro servidor será necesario que, cada vez que se reciba una petición de un cliente, un proceso independiente se encargue de atenderla. Como el coste de crear un nuevo proceso es elevado (en tiempo de ejecución y de recursos del sistema consumidos), este tipo de concurrencia es útil en aquellos casos en los que:

- La respuesta del servidor requiere un trabajo muy significativo de acceso a los dispositivos de entrada/salida de la máquina (una consulta en una base de datos, por ejemplo) lo cual permite que, mientras el servicio a un determinado cliente está bloqueado a la espera de los dispositivos de entrada/salida, el servidor puede atender otro cliente.
- El tiempo de procesamiento entre las peticiones de los distintos clientes varía mucho. Por esta causa no se puede estimar un tiempo de respuesta uniforme para cada petición razón por la que algunos clientes tendrían que esperar excesivamente para ser atendidos o incluso, se perderían sus peticiones.
- El servidor se ejecuta en máquinas multiprocesadoras permitiendo que una instancia del servidor pueda ejecutarse en cada uno de los procesadores del sistema.

El algoritmo básico para construir un servidor concurrente sería de la siguiente forma:

1. El proceso maestro del servidor crea un `socket` y lo inicializa con los parámetros adecuados, colocándolo en modo pasivo (vamos a suponer un servidor orientado a la conexión).
2. Este proceso maestro recibe las peticiones de los clientes llamando a la función `accept()`. Cada vez que llega una petición crea un proceso hijo llamando a la función `fork()` para que se encargue de atender a ese cliente. El proceso maestro retorna a la escucha de nuevas peticiones.
3. El proceso hijo atiende al cliente. Cuando termina, cierra la conexión y termina su ejecución, habitualmente llamando a la función `exit()`.

Este mecanismo puede funcionar debido a que los procesos hijos, cuando son creados, heredan de su proceso padre la tabla de descriptores de ficheros abiertos, por lo que podrán atender al cliente a través de la conexión ya aceptada por el padre.

La gestión correcta de los *sockets* comunes tanto al proceso maestro como al proceso hijo obliga a que el padre cierre inmediatamente el *socket* que ha obtenido como retorno de la llamada a `accept()` para que el proceso hijo lo pueda utilizar en exclusiva. De la misma manera, el proceso hijo debe cerrar el *socket* en el que el proceso maestro escucha por nuevas peticiones.

Procesos errantes Un inconveniente que se genera al crear múltiples procesos hijos que terminan su ejecución llamando a la función `exit()`, es la creación de procesos errantes (*"zombis"*) en la tabla de procesos del sistema.

Estos procesos errantes o zombis se crean cuando un proceso termina su ejecución. Normalmente, cuando un proceso termina su ejecución el sistema envía una señal a su proceso padre para informarle. Si el proceso padre no recibe esa señal, el sistema mantiene una entrada para el proceso hijo en la tabla de procesos del sistema hasta que el proceso padre reciba esa señal.

La mejor forma de actuar para evitar la proliferación de procesos errantes consiste en que el padre suministre al sistema un manejador para esa señal o bien que le indique al sistema que desea ignorar dicha señal. Ambas cosas se pueden conseguir llamando a la función `signal()`[7]. La línea de código que podemos incorporar a nuestros programas para indicale al sistema operativo que el padre desea ignorar este tipo de señales es la siguiente:

```
signal(SIGCHLD, SIG_IGN);
```

Adaptabilidad del código del maestro El mecanismo de lanzar los procesos hijos para atender las distintas peticiones usando la llamada al sistema `fork()` tiene el inconveniente de que el código del proceso maestro y el código del proceso hijo tienen que ser el mismo, ya que `fork` genera en el sistema un proceso idéntico al que realiza la llamada.

Esto es un inconveniente porque, como veremos en la sección 1.7.2, podremos tener servidores que escuchen en múltiples puertos de protocolo y, en cada uno de esos puertos, podemos ofertar un servicio distinto. Si el programa maestro tiene que tener el código de cada uno de los servicios que oferta, o dicho de otra manera, todos los servicios comparten el mismo código fuente, una simple modificación en uno de ellos obligaría a reconstruir el servidor completo.

Para evitar este problema es posible realizar una pequeña modificación al algoritmo simple que hemos visto antes y construir un proceso maestro

[7] Se recomienda consultar la página de ayuda correspondiente a esta función para comprender mejor su funcionamiento.

que simplemente reciba peticiones y llame a la función `fork()`. El proceso hijo identificaría el servicio solicitado y, mediante una llamada a la función `execve()`[8], cargaría desde disco el programa adecuado para servir esa petición. Esto nos permite tener un fichero independiente para cada uno de los servicios ofertados.

El coste de la concurrencia

El algoritmo simple para el servidor concurrente presentado en la sección anterior parece resolver el problema que se presenta al tener que atender a múltiples clientes simultáneamente. Pero esa solución no es gratis en téminos de prestaciones y tiempo de respuesta. Veamos un ejemplo sencillo que aclare el problema:

Supongamos que un determinado servicio ocupa S unidades de tiempo en llevarse a cabo. La versión iterativa del servidor tardaría $2 * S$ unidades de tiempo en atender dos peticiones simultáneas de dos clientes. Si el tiempo que tarda el sistema operativo en crear un nuevo proceso es C, la versión concurrente de ese servidor tardaría al menos $2 * C$ unidades de tiempo en crear los dos procesos hijos que atienden a los dos clientes a lo que habría que añadir el tiempo S de servicio de cada petición.

Si $C > S$ (o muy próximo a ese valor), el tiempo total empleado por el servidor concurrente sería $2 * C + S$ ya que, mientras se crea el segundo servidor el otro atiende al primer cliente. Vemos que debido a la similitud de los valores de C y S no habría ninguna ganancia con la implementación concurrente.

Creación previa de servidores Cuando el tiempo de creación de un nuevo proceso es similar al tiempo de servicio, la solución de ir creando un nuevo servidor para cada petición de un cliente, no es aceptable. En estos casos se puede utilizar otra estrategia que es la creación previa de procesos para crear una cooperativa de servidores.

La idea consiste en que, en vez de crear los procesos hijos según van llegando las sucesivas peticiones de servicio de los clientes, lo que se hace es crear los procesos servidores *antes* de que lleguen esas peticiones, habitualmente en el arranque del servidor. De esta forma, se dispone de una serie de procesos servidores para atender las peticiones de los clientes que se reciban. Los servidores se iran asignando a las peticiones por orden de llegada.

La implementación de esta estrategia no es muy compleja ya que, como hemos comentado anteriormente, un proceso hijo comparte los mismos descriptores de fichero abiertos que su proceso padre y además, el sistema operativo `UNIX` proporciona la mutua exclusión entre todos los procesos que intenten aceptar una conexión sobre el mismo *socket*.

[8] Consultar la ayuda para obtener una descripción detallada de esta función.

Por tanto, el proceso maestro debería crear e inicializar el *socket* de la manera habitual. Antes de realizar la llamada a la función `accept()` para bloquearse a la espera de peticiones, debería crear el número de procesos hijos deseados. Una vez han sido creados, cada proceso hijo debe realizar su propia llamada a `accept()` para bloquearse a la espera de peticiones[9]. Todos los procesos se bloquean a la espera en el mismo puerto. Cuando llega una petición, el sistema operativo desbloqueará a todos los procesos a la vez. El planificador de tareas le dará el control a uno de esos procesos en primer lugar, el cual aceptará la conexión y será el encargado de atender esa petición. El resto de procesos, al intentar aceptar esa conexión, se encuentran con el puerto ya en uso, por lo que se volverán a bloquear a la espera de nuevas conexiones. Cuando un servidor termina de atender a su cliente, debe cerrar el *socket* a través del que se comunicaron (el valor devuelto por `accept()`) y volver a bloquearse a la espera de nuevas peticiones.

La creación previa de procesos debe hacerse con cuidado ya que, aunque se elimina la sobrecarga de la creación de un proceso en el momento de recibir una petición, se añaden sobrecargas al sistema en la gestión de procesos activos y en la gestión del sistema de comunicaciones. No se deben crear grandes cantidades de procesos si no se quiere obtener un efecto contrario del deseado.

1.7.2. Concurrencia aparente

En este tipo de concurrencia, se consigue que un servidor atienda las peticiones de múltiples clientes con únicamente un proceso activo. Este tipo de estrategia permite conjugar los beneficios de tener múltiples procesos servidores pero sin la sobrecarga de crear dichos procesos. A pesar de eso, este tipo de implementación no es aplicable a todas las clases de servidores. Sólamente un cierto tipo de ellos pueden hacer uso de esta técnica.

Negación de Servicio Antes de pasar a detallar la implementación de esta estrategia hay que comentar un problema que afecta a los servidores atendidos por un único proceso: el servidor puede ser anulado por un cliente defectuoso o malicioso. Estos servidores son vulnerables al ataque denominado **Negación de servicio** (*"Denial of Service"*).

[9]Este funcionamiento que acabamos de describir sólo es posible en sistemas operativos descendientes de BSD. En el caso de que tengamos que trabajar con un sistema operativo descendiente del System V (tal como le ocurre al sistema operativo Solaris) los procesos hijos no se podrán bloquear simultáneamente sobre el mismo *socket*.

Esto es así porque estos sistemas operativos implementan la llamada a `accept()` como una función de librería y no como una función interna del kernel. Para solucionar este problema, tenemos que colocar un mecanismo de exclusión mútua alrededor de la llamada a `accept()` en cada proceso hijo, de tal manera que sea sólo uno de ellos en el que esté bloqueado en cada instante. Los demás procesos estarán bloqueados intentando acceder al *socket*.

El problema consiste en lo siguiente. Imaginemos un cliente con mal funcionamiento. Cuando establece una conexión con el servidor, éste la acepta, pero a partir de ese momento el cliente ya no vuelve a solicitar ningún servicio. El servidor se quedaría bloqueado (concretamente en la llamada a `read())` a la espera de recibir datos del cliente. Pero éstos no llegarían.

Otra posibilidad es que el cliente envíe una cantidad mínima de información (1 byte, por ejemplo) y el servidor espere, para siempre, por el resto. Finalmente, también se podría hacer que el cliente enviara una gran cantidad de peticiones, pero no leyera nunca las respuestas. En este caso, al llenarse los *buffers* de escritura del servidor, éste se bloquearía.

Cuando el servidor se implementa utilizando varios procesos, sólo se queda bloqueado el proceso que comunica con ese cliente en particular, pudiendo mantenerse el servicio. En el caso en el que solamente hay un proceso, el servicio se dejaría de suministrar.

Entrada/Salida asíncrona: `select()`

Para implementar la concurrencia aparente tenemos que utilizar la entrada/salida asíncrona. La idea es que un proceso solo lea datos (y por tanto, se bloquee) cuando éstos le llegan al puerto.

A veces es necesario que nuestro programa reciba información desde más de una fuente simultáneamente. Podemos asignar, entoces, un *socket* a cada uno de los orígenes de la información y nuestro programa realizar llamadas sucesivas a `accept()` sobre cada uno de los *sockets* para recibir los datos. El problema de este esquema de funcionamiento es que nuestro programa se queda bloqueado cada vez que se llama a `accept()` sobre un *socket* hasta que reciba algún tipo de conexión por ese *socket*. El programa se quedará paralizado esperando por una información que no llega a través de un determinado camino, mientras que por otro *socket* podemos tener solicitudes de conexión que no podemos atender.

El problema también puede ocurrir cuando tenemos un programa que, además de realizar comunicaciones a través de la red, queremos que sea interactivo y permita que el usuario se comunique con él a través del teclado. Mientras el programa está bloqueado a la espera de mensajes, no puede atender al usuario.

Para evitar este problema tenemos varias técnicas de multiplexado de la entrada/salida. Entre ellas:

- Se puede realizar una llamada a `accept()` no bloqueante. Para ello hay que realizar una llamada a la función del sistema `ioctl()` para indicar que el socket es no bloqueante. El proceso tiene que realizar un bucle de llamadas a `accept()` para comprobar si hay conexiones pendientes. En el caso de que no haya conexiones, habrá que repetir el bucle una y otra vez hasta que llegue una conexión. Este método se denomina *"polling"* o *consulta*.

- Otra posibilidad es una variación de la ya vista en el caso de concurrencia real y consiste en crear un proceso hijo que se bloquee sobre cada uno de los *sockets* y espere por conexiones. Cuando llega una conexión, el proceso hijo pasaría los datos de la conexión al proceso padre para que realice el trabajo.

- La tercera posibilidad sería bloquearse a la espera de conexiones por *cualquiera* de los *sockets*. Cuando llega alguna petición de conexión por *alguno* de los *sockets*, el programa se desbloquea y atiende al cliente. Para realizar este multiplexado de la entrada/salida es necesario utilizar la llamada del API de *sockets* `select()`.

La función `select()` nos permite "escuchar" en varios *sockets* a la vez[10]. De esta manera, la concurrencia viene determinada por los datos: según van llegando datos, el servidor se activa, lee la petición y sirve al cliente, pasando a bloquearse de nuevo a la espera de peticiones. Lógicamente, para que la apariencia de servidor concurrente se mantenga, es necesario que el tiempo de servicio no sea demasiado elevado.

Descripción de `select` Pero, ¿cómo funciona exactamente `select()`? Antes de entrar en los detalles de programación, vamos a ver qué tipos de problemas resuelve está función.

A `select()` le podemos plantear una cuestión como la siguiente: *"Observa los* sockets **`a`**, **`b`** *y* **`c`** *y avisa si se ha recibido alguna conexión por ellos y están listos para leer. Al mismo tiempo observa el* socket **`d`** *y comprueba si está listo para enviar datos. En cualquiera de las dos condiciones (leer o escribir) avisa inmediatamente".* Simultáneamente también podemos determinar la cantidad de tiempo que estamos dispuestos a esperar por alguna de esas condiciones. Esa cantidad de tiempo la especificamos pasándole a `accept()` un parámetro que será una estructura `timeval`. La estructura `timeval` está definida en el fichero `<sys/time.h>` y tiene la siguiente forma:

[10] La función `select()` no sólo se puede utilizar para atender conexiones de red, también se puede usar con cualquier tipo de descriptor de fichero. De esta manera, es posible que, mientras un programa "atiende" a sus conexiones de red, pueda al mismo tiempo atender al usuario a través del teclado. El descriptor de fichero (número entero índice en la tabla de descriptores de ficheros) del teclado lo podemos obtener a partir del nombre de fichero asignado (`stdin`) usando la función `fileno()`. Esta función devuelve el descriptor de fichero asignado a un dispositivo de tipo *stream* del sistema. El prototipo de esta función es el siguiente:

```
int fileno(FILE * stream)
```

siendo `stream` la variable de tipo *stream* inicializada en nuestro programa.

```
struct timeval {
     long tv_sec;       /* segundos */
     long tv_usec;      /* microsegundos */
};
```

Así, por ejemplo, si estamos dispuestos a esperar 2 seg y medio tendríamos que inicializar la estructura de la siguiente manera:

```
struct timeval tiempo;

   tiempo.sec=2;           /* 2 segundos */
   tiempo.usec=500000;     /* 0.5 segundos= 500000 microsegundos */
```

Por otro lado, para indicarle a `select()` los *sockets* que tiene que observar se usan estructuras del tipo `fd_set`. Estas estructuras son arrays de enteros en los que cada bit representa un *socket*. Si, por ejemplo, los enteros son de 32 bits, el bit 0 del primer entero representa al *socket* 0, el bit 1 al *socket* 1, y así sucesivamente hasta el *socket* 31. Para el segundo entero del array, el bit 0 representará al *socket* 32, el 1 el *socket* 33, y así hasta el 63. Todos estos detalles de implementación están ocultos en la estructura `fd_set`.

Para determinar el *socket* que hay que observar, el bit correspondiente a ese *socket* de la estructura `fd_set` se tiene que poner a 1. Para facilitar la tarea de trabajar con los bits de las estructuras `fd_set` se utilizan las siguientes macros:

```
FD_ZERO( &fdset );      /* Inicializa todos los bits a cero */
FD_SET ( fd, &fdset );  /* Pone a 1 el bit indicado por fd */
FD_CLR ( fd, &fdset );  /* Pone a 0 el bit indicado por fd */
FD_ISSET( fd, &fdset ); /* Comprueba si el bit indicado por fd está a 1 */
```

donde:

fdset Es una estructura de tipo `fd_set`.

fd Es un valor entero que representa un descriptor de *socket*.

Vamos a ver ahora como es el prototipo de la función `select()`:

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

retcod=select ( maxfd, lect_fd, esc_fd, exc_fd, tiempo );
```

donde:

maxfd Es un valor entero que indica el número de descriptores que tienen que ser comprobados. Como los descriptores los va asignando el sistema de manera secuencial empezando por 0, el valor de *maxfd* será el valor del máximo descriptor a ser comprobado más 1.

lect_fd, esc_fd y exc_fd Son punteros a estructuras del tipo `fd_set`. *lect_fd* determina los *sockets* que hay que comprobar para saber si están listos para lecturas. El parámetro *esc_fd* determina que *sockets* hay que comprobar para saber si están listos para escrituras. El parámetro *exc_fd* determina los *sockets* que tienen condiciones excepcionales pendientes. Este último tipo de condiciones no las utilizaremos.

tiempo Es un puntero a una estructura del tipo `timeval` en la que se indicará la cantidad de tiempo que se deben comprobar las condiciones antes de desbloquear la llamada y retornar.

`retcod` El valor que retorna `select()` es el número total de descriptores que están listos. Si `select()` retorna a causa de que ha expirado el tiempo asignado por el último parámetro, el valor que devuelve es cero. Si devuelve el valor -1 significa que se ha producido algún error.

Si alguno de los parámetros *lect_fd*, *esc_fd* o *exc_fd* son punteros a NULL, quiere decir que no estamos interesados en esa condición. Este será para nosotros el caso de *exc_fd*.

Si en el último parámetro *tiempo* pasamos un puntero a NULL, estamos solicitando que `select()` consulte los *sockets* de manera permanente hasta que se verifique alguna de las condiciones. De esta manera por ejemplo, podemos usar `select()` para esperar conexiones a través de más de un *socket* sin necesidad de tener que usar `accept()` e ir bloqueándose en cada uno de los *sockets*.

Cuando la función `select()` retorna porque se ha producido alguna de las condiciones en alguno de los *sockets* que está observando, las estructuras `fd_set` se habrán modificado para reflejar cuál de los *sockets* ha sido el afectado. Una vez procesada esa condición, si se desea volver a llamar a la función `select()` será necesario reinicializar correctamente las estructuras `fd_set`.

Vamos a ver un ejemplo sencillo del uso de `select()`:

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

#define PUERTO1 9999
#define PUERTO2 8888
```

```
main()
{
        int sock1,sock2,n_sock,recib, maxfd;
        struct sockaddr_in direc1,direc2;
        fd_set selector;
        struct timeval tiempo;

        /* Creamos un par de sockets */
        if ( ( sock1= socket(PF_INET,SOCK_STREAM, 0)) < 0 ) {
                printf("\n\nError en socket 1\n\n");
                exit();
        }
               if ( ( sock2= socket(PF_INET,SOCK_STREAM, 0)) < 0 ) {
                printf("\n\nError en socket 2\n\n");
                exit();
        }
        direc1.sin_family=AF_INET;
        direc1.sin_port=htons(PUERTO1);
        direc1.sin_addr.s_addr=INADDR_ANY;
        direc2.sin_family=AF_INET;
        direc2.sin_port=htons(PUERTO2);
        direc2.sin_addr.s_addr=INADDR_ANY;
        if ( (bind(sock1, (struct sockaddr *) &direc1, sizeof (direc1))) <0) {
                printf("\nError en bind 1\n");
                exit();
        }
        if ( (bind(sock2, (struct sockaddr *) &direc2, sizeof (direc2))) <0) {
                printf("\nError en bind 2\n");
                exit();
        }
        /* Ponemos los sockets en modo "pasivo" */
        listen(sock1, SOMAXCONN);
        listen(sock2, SOMAXCONN);

        /* A partir de aquí es nuevo para preparar el select */

        /* Se ponen a cero todos los bits de la estructura selector */
        FD_ZERO(&selector);

        /*
           Averigüamos cuál es el mayor de los dos descriptores de
           socket para calcular el número de sockets a comprobar por
           select
        */
        if (sock1 > sock2 )
          maxfd=sock1+1;
        else
          maxfd=sock2+1;

        /*
          Se especifican los sockets que queremos observar a través de
          select
          */
```

```
FD_SET(sock1,&selector);
FD_SET(sock2,&selector);

/* Fijamos un tiempo máximo de espera */
tiempo.tv_sec=22;
tiempo.tv_usec=0;

/*
  Llamamos a select y hacemos que se bloquee hasta que se
  reciba alguna conexión por alguno de los dos sockets o hasta
  que pase el tiempo prefijado.
  */
if (select(maxfd,&selector,(fd_set *) 0, (fd_set *) 0,
           &tiempo) < 0 ) {
  printf("Error en select:");
  exit(-1);
}
/*
   Comprobamos si la conexión se ha recibido a través de sock1
*/
if (FD_ISSET(sock1,&selector) ) {
  /* Si ha sido así, conectamos con el cliente */
  if ( (n_sock=accept(sock1,NULL,NULL)) < 0 ) {
    printf("\n\nError en la conexion 1\n");
    exit();
  }
}
/*
   Comprobamos si la conexión se ha recibido a través de sock2
*/
if (FD_ISSET(sock2,&selector) ) {
  /* Si ha sido así, conectamos con el cliente */
  if ( (n_sock=accept(sock2,NULL,NULL)) < 0 ) {
    printf("\n\nError en la conexion 2\n");
    exit();
  }
}
/* A partir de aqui, es un programa normal de intercambio de
   información usando el interface de sockets... */
```

1.8. Otras funciones interesantes

En esta sección vamos a ver algunas funciones del API de *sockets* que, sin ser absolutamente vitales para realizar una comunicación a través de la red, aqyudan a construir mejores programas.

1.8.1. Obtener el IP de un nodo: `gethostbyname()`

Esta función se encuentra en la librería `libnsl.a`, por lo que habrá que compilar con la opción `-lnsl` en los programas que la utilicen[11]. Su sintaxis es:

```
info = gethostbyname(nombre_nodo);
```

donde

`nombre_nodo` es una cadena de caracteres con el nombre del nodo cuya dirección queremos averiguar. Por ejemplo ‘‘`opalo`’’ o bien ‘‘`src.doc.ic.ac.uk`’’.

`info` es un puntero a una estructura de tipo `hostent` en la cual se halla la información deseada. Esta estructura es bastante complicada, pero sólo nos interesan los campos llamados `h_addr` y `h_length`. El primero de ellos contiene la dirección IP de la máquina cuyo nombre hemos facilitado. El segundo, la longitud en bytes del dato anterior. La propia función `gethostbyname()` se ocupa de reservar espacio para la estructura de tipo `hostent`

Usualmente se copian los bytes contenidos en `info->h_addr` directamente al campo `.sin_addr.s_addr` de la estructura `sockaddr_in` mediante la función `memcpy()`. Un ejemplo aclarará esto:

```
/*
    Ejemplo de gethostbyname()
*/
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<netdb.h>
#include<stdio.h>
...
struct sockaddr_in servidor;
struct hostent *info;
int s, n_sock;
...
servidor.sin_family=AF_INET;
servidor.sin_port=htons(15001);
/* Ahora averiguaremos la direccion IP del nodo "oboe" */
info = gethostbyname("oboe");
if (info==NULL) {
      fprintf(stderr, "oboe: nodo desconocido.\n");
```

[11] Esto es cierto para el caso del sistema operativo SunOS. En otros sistemas no será necesario utilizar esa librería o por el contrario habrá que utilizar otra distinta. Se recomienda utilizar la ayuda del sistema: `man`

```
      exit(1);
    }
/* Copiar la informacion obtenida a la estructura "servidor" */
memcpy(&servidor.sin_addr.s_addr, info->h_addr, info->h_length);
/* Todo esta listo para intentar la conexion */
connect(s, (struct sockaddr *) &servidor, sizeof servidor);
...
```

observar el uso de `memcpy()` para transferir la información. Una mera asignación del tipo:

```
servidor.sin_addr.s_addr = info->h_addr;
```

no funcionaría, ya que se está asignando un puntero a una variable de tipo `long` (y aunque hiciéramos un *cast* se copiaría el valor contenido en el puntero, en lugar del valor apuntado por el puntero).

1.8.2. Obtener el nombre de un nodo: `gethostbyaddr()`

Esta función también se encuentra en la librería `libnsl.a`, por lo que habrá que compilar con la opción `-lnsl` en los programas que la utilicen. Su sintaxis es:

```
nodo = gethostbyaddr(direc, tam, familia);
```

donde

`nodo` es un puntero a una estructura de tipo `hostent` en la cual se halla la información solicitada.

`direc` es un puntero a una estructura de tipo `in_addr` que contiene la dirección de la máquina.

`tam` es el tamaño en bytes del parámetro anterior.

`familia` indica qué familia de direcciones representa *`direc`*. Normalmete será el valor `AF_INET`.

1.8.3. Obtener servicios y puertos por su nombre

Los servicios más comunes se conocen por su nombre (ftp, telnet, etc) mas que por el número de puerto de protocolo asignado. Los distintos nombres de cada servicio se encuentran en el fichero del sistema `/etc/services`. Cuando creamos una aplicación que trabaja con alguno de estos servicios podemos usar directamente el puerto de protocolo en el código del programa, pero en

el caso de dicho servicio cambie de puerto, será necesario recompilar nuestro código.

Del mismo modo, si somos administradores de una red de máquinas, nada nos impide editar el fichero `/etc/services` y añadir una entrada con algún servicio que nosotros hayamos creado y que, a partir de ese momento, podremos referirnos a él por el nombre que le hayamos asignado.

Para acceder a un puerto por su nombre, tenemos la función:

```
listserv = getservbyname (servicio, protocolo);
```

donde

`servicio` es una cadena de caracteres con el nombre del servicio del que queremos obtener su información. Por ejemplo ‘‘`ftp`’’, ‘‘`telnet`’’. Tiene que ser un nombre que aparezca en el fichero `/etc/services`.

`protocolo` es una cadena de caracteres con el protocolo deseado. Algunos servicios están disponibles tanto para `UDP` como para `TCP`. En este campo especificamos el protocolo deseado. En el caso de servicios ofertados bajo un único protocolo se puede poner `NULL`.

`listserv` es un puntero a una estructura de tipo `servent` en la cual se halla la información deseada. Esta estructura tiene los siguientes campos:

```
struct servent {
      char  *s_name;     /* Nombre oficial del servicio */
      char **s_aliases;  /* Lista de alias */
      int    s_port;     /* Número de puerto */
      char  *s_proto;    /* Protocolo a usar */
```

Los campos de la estructura son bastante autoexplicativos. Hay que hacer la aclaración que el campo de puerto ya está inicializado con sus bytes en el orden utilizado en la red, por lo que no será necesario utilizar la función `htons()`.

Además de esta función, tenemos la posibilidad de conocer la información que aparece en el fichero `/etc/services` a partir del número de puerto. Para ello utilizaremos la función:

```
listserv = getservbyport (puerto, protocolo);
```

donde

`puerto` es el número de puerto para el cual estamos buscando la información. Se tiene que pasar a la función en formato de red, por lo que se debería de utilizar siempre la función `htons()`.

`protocolo` es una cadena de caracteres con el protocolo deseado.

`listserv` es un puntero a una estructura de tipo `servent` en la cual se halla la información deseada.

1.9. Análisis del interfaz de `sockets`

Los `sockets` son un interfaz de bajo nivel para programar en red, son algo así como el ensamblador de la red. Un programador debe llevar a cabo las siguientes tareas de forma explícita, las cuales son susceptibles de realizarse con error:

Determinar la dirección del servicio. El usuario tiene que suministrar de manera explícita la dirección del servicio: la dirección IP de la máquina del servidor y el puerto de protocolo en el que está escuchando ese servicio. Al hacerlo de esta manera, el servicio queda "fijado" a esta dirección. Aunque podemos aliviar el conocimiento necesario por parte del usuario usando la función `gethostbyname` (en este caso sólo habría que conocer el nombre de la máquina), si queremos flexibilidad en la ubicación de los servicios necesitaremos algún tipo de mecanismo de localización de los mismos.

Inicializar el socket y conectar con el servidor. Se necesita una cantidad de código no trivial para establecer una conexión con un servidor.

Empaquetado y desempaquetado de la información. Que podamos realizar la transferencia de bytes entre las máquinas no garantiza la transferencia de información. Cuando queremos transmitir estructuras de datos complejas hay que solucionar problemas como:

1. Ordenación de los bytes en memoria dentro de cada máquina.
2. Tamaño de los distintos tipos de datos en cada máquina.
3. Alineación en memoria de los datos para cada tipo de máquina.

Esto significa que tendremos que crear una serie de funciones específicas en nuestra aplicación para solventar estos problemas.

Envío y recepción de mensajes. El código necesario para enviar y recibir mensajes es bastante complejo. El programador tiene que enfrentarse y resolver los problemas derivados de las escrituras cortas ("*short-writes*") o de la transferencia de información a través de chorros de bytes sin respetar los límites de los registros lógicos de nuestra aplicación.

Detección y recuperación de errores. Como en todo programa C, los errores de tipeado son difíciles de localizar. Los *sockets* no tienen un tipo "fuerte" (simplemente son enteros), por lo que un *socket* asociado a una comunicación orientada a la conexión no es sintácticamente distinto de un *socket* asociado a un comunicación orientada a los datagramas. Un compilador no puede detectar el uso incorrecto de un socket. Es por tanto necesario incorporar gran cantidad de código de gestión de errores.

Portabilidad. Aunque los *sockets* son un estándar *de facto*, algunos sistemas no disponen de implementaciones o las implementaciones no son completas. Nuestra aplicación distribuida no sería portable a estos sistemas si la codificamos usando únicamente el interfaz de *sockets*.

Seguridad. El interfaz de *sockets* no incorpora ningún tipo de seguridad. El propio programador de la aplicación tiene que incorporar toda la seguridad que desee.

Bibliografía

[1] Douglas E. Comer and David L. Stevens. *Internetworking with TCP/IP. Client-Server programming and applications.* Prentice Hall, 1993.

[2] W. R. Stevens. *UNIX Network Programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[3] W. R. Stevens. *UNIX Network Programming.* Prentice-Hall, Englewood Cliffs, New Jersey, second edition, 1998.