Programación de *sockets*

Xavier Perramon Tornil
Enric Peig Olivé

P03/75064/00978

UOC
www.uoc.edu

Índice

Introducción

Este módulo didáctico contiene una descripción detallada de la llamada *interfaz de programación de sockets*. Dicha interfaz facilita el desarrollo de aplicaciones basadas en los protocolos de transporte TCP y UDP, que se han estudiado en otros módulos. Diseñada en un inicio para el sistema operativo UNIX, en la actualidad existen versiones de la librería de *sockets* para una gran variedad de plataformas y sistemas operativos.

Consultad los protocolos TCP y UDP en los apartados 7 y 8 del módulo didáctico "TCP/IP: los protocolos de la red Internet" de esta asignatura.

La interfaz de programación de *sockets* está formada por una serie de llamadas al sistema, funciones auxiliares de soporte y definiciones de tipos de datos para representar las variables con que trabaja, tales como direcciones IP y números de puerto. En este módulo se describen todos estos elementos de la interfaz con suficiente detalle para poderlo utilizar como manual de referencia.

Para poder seguir este módulo, son necesarios ciertos conocimientos o bien del lenguaje de programación C, o bien del lenguaje Java. Asimismo, es conveniente, pero no imprescindible, tener algunos conocimientos de sistemas operativos en general y de UNIX en particular.

Objetivos

Los objetivos principales que debéis lograr con este módulo didáctico son los siguientes:

1. Saber situar la interfaz de programación de *sockets* en el contexto de los protocolos TCP, UDP e IP, de las aplicaciones Internet y del sistema operativo UNIX.

2. Conocer con detalle las llamadas al sistema que forman la interfaz de los *sockets* y los parámetros de los mismos, si se opta por trabajar con el lenguaje C, o conocer detalladamente las clases de Java que proporcionan toda la funcionalidad de los *sockets*.

3. Ver cómo la interfaz de programación de *sockets* puede utilizarse para diferentes tipos de aplicaciones: comunicación de procesos en un mismo ordenador, o en ordenadores conectados por medio de una red, y aplicaciones basadas en protocolos de transporte fiables o en datagramas no fiables.

4. Poder escribir aplicaciones basadas en *sockets*, tanto programas cliente como programas servidor.

5. Conocer las principales funciones auxiliares que incorpora la interfaz de los *sockets* (o, si procede, los métodos de las clases Java), tales como las que permiten traducir nombres a direcciones o las que sirven para modificar el comportamiento de los *sockets* por medio de sus opciones.

1. Qué son los *sockets*

1.1. Visión general

Al principio de la década de los ochenta, una parte importante de los ordenadores conectados por la red ARPANET y a su sucesora, la red ARPA Internet, sobre las que se desarrollaron los protocolos IP, TCP y UDP, utilizaba el sistema operativo UNIX y, más concretamente, la variante de la Universidad de Berkeley (BSD UNIX). A partir de la versión BSD 4.2, se incorporó a este sistema operativo una serie de llamadas en el ámbito del núcleo (o *kernel*) para facilitar el diseño de aplicaciones que utilizaran los protocolos de ARPANET. Dichas llamadas, junto con algunas funciones auxiliares de la librería estándar de UNIX, forman la llamada *interfaz de programación de sockets*.

Un *socket* o conector es un punto de acceso a los servicios de comunicación en el ámbito de transporte. Cada *socket* tiene asociada una dirección que lo identifica. Conociendo esta última, se puede establecer una comunicación con un *socket* para que actúe como extremo de un canal bidireccional.

Existen otras interfaces de programación de los protocolos TCP y UDP, tales como la TLI, pero la de los *sockets* es probablemente la que más se utiliza en la actualidad. Además de haberse incorporado a la especificación "System V Release 4 UNIX", la interfaz de programación de los *sockets* se ha llevado a diferentes sistemas operativos distintos o se ha adaptado a los mismos.

Interfaz TLI

Para acceder a los protocolos de transporte, la interfaz TLI utiliza el mecanismo de los flujos de datos(*streams*), propio de la variante de UNIX llamada *System V*.

TLI es la sigla de *Transport Layer Interface*.

1.2. Relación entre los *sockets* y el sistema operativo UNIX

Dado su origen, la interfaz de programación de los *sockets* está íntimamente vinculada al sistema operativo UNIX. Algunas de las llamadas proporcionadas por la interfaz son directas al núcleo del sistema, y otras constituyen funciones integradas en la librería estándar del lenguaje de programación más utilizado en UNIX, el lenguaje C (desde el punto de vista del programador, las llamadas al núcleo también se ven como si fueran funciones de la librería C).

En la actualidad, hay disponibles otros lenguajes de programación que proporcionan acceso a las llamadas del sistema para trabajar con *sockets*, como el lenguaje interpretado Perl*, o el lenguaje Java. En este módulo veremos cómo se trabaja con *sockets* en dos de estos lenguajes: C y Java.

*** De hecho, el intérprete de Perl está escrito en C.**

El motivo de utilizar C es evidente: como se ha comentado, la relación entre *sockets* y C es bastante íntima. Por lo que respecta a Java, su uso está cada vez más extendido, principalmente en el desarrollo de interfaces de usuario. Y el motivo de explicar los dos es ofrecer al alumno la opción de elegir el lenguaje en que se sienta más cómodo a la hora de realizar aplicaciones con *sockets*.

En el modelo de programación de los *sockets* se sigue la filosofía general de UNIX de tratar los dispositivos y los mecanismos de entrada/salida como ficheros. Excepto la operación de abrir o crear un *socket*, que es especial, las otras funciones básicas aplicables a ficheros (leer datos, escribirlos, cerrar) también son directamente aplicables a los *sockets*. Sin embargo, como es natural, existen otras funciones que son específicas de estos últimos.

De hecho, la interfaz de los *sockets* proporciona un método general de comunicación entre procesos. Una de las posibilidades que ofrece este método es comunicar procesos que se ejecutan en un mismo ordenador, de manera similar a las *pipes* (que también se representan por medio de descriptores de ficheros). La aplicación más habitual, sin embargo, es utilizar los *sockets* como canal de comunicación entre procesos que pueden correr en ordenadores diferentes, conectados mediante una red.

Consultad las *pipes* en el subapartado 5.1 del módulo didáctico "Los dispositivos de entrada/salida" de la asignatura *Sistemas operativos I*.

1.3. Direcciones de los *sockets* y su clasificación

Como hemos visto, cada *socket* tiene asociada una **dirección** que sirve para identificarlo y para que se le puedan enviar datos desde otro *socket*.

En numerosas ocasiones, también se habla del **nombre de un *socket***, que no es más que su dirección.

Asimismo, hemos visto que puede haber diferentes clases de *sockets*, básicamente los que comunican procesos de un mismo sistema y los que permiten la comunicación entre sistemas diferentes. La distinción se efectúa por medio de los nombres o direcciones, puesto que a cada clase de *sockets* le corresponde un **espacio de nombres** diferente.

De este modo, existen dos espacios de nombres básicos que se pueden especificar para un *socket* y que son los siguientes:

- El **espacio de nombres de ficheros**: se utiliza en los *sockets* que deben comunicar un proceso con otro del mismo sistema.

- El **espacio de nombres Internet**: se utiliza en los *sockets* que deben comunicar un proceso con otro de cualquier sistema. En casos particulares, puede ser el mismo sistema, aunque, por norma general, será otro sistema al que esté conectado por medio de una red.

Por otro lado, la clase de un *socket* también determina los protocolos de comunicación que se pueden utilizar. Por consiguiente, cuando se especifica a qué espacio de nombres pertenece un *socket*, se indica al mismo tiempo un conjunto de posibles protocolos, de los que deberá seleccionarse uno para poder establecer la comunicación con otro *socket* (obviamente, para que dos *sockets* se puedan comunicar deben utilizar el mismo protocolo).

Por ello, en ocasiones se utiliza la expresión *familia de protocolos* como sinónimo de *espacio de nombres*. Y en algunas situaciones también se utiliza el término *dominio* para referirse a un espacio de nombres*.

*** El término *dominio* tiene acepciones diferentes en otros contextos y podría dar lugar a confusiones.**

En definitiva, podemos decir que los conceptos clase de *socket*, espacio de nombres, dominio (referido a nombres de *sockets*) y familia de protocolos se pueden considerar equivalentes.

Asimismo, a cada espacio de nombres le corresponde un formato de direcciones; así pues, también existe una asociación obvia entre ambos conceptos.

1.3.1. El espacio de nombres de ficheros

En el espacio de nombres de ficheros (denominado también *dominio UNIX*) las direcciones de los *sockets* simplemente constituyen nombres de ficheros. Es decir, a cada *socket* de esta clase le corresponde un fichero y, más concretamente, en el caso del sistema operativo UNIX, una entrada de directorio de tipo *socket* en el sistema de ficheros. Ello hace que sólo se puedan utilizar los *sockets* de este espacio de nombres para comunicar procesos que se ejecuten en el mismo ordenador.

Entradas de directorio

Las entradas de directorio en el sistema de ficheros UNIX, además del tipo *socket*, pueden ser de tipo fichero regular, directorio, enlace simbólico (*link*), *pipe* o dispositivo de bloques o de carecteres.

El mecanismo de comunicación por medio de esta clase de *sockets* es similar al de las *pipes* en UNIX. La diferencia principal reside en que en un *socket* la comunicación es bidireccional, mientras que en una *pipe* sólo se puede escribir por uno de los extremos y leer por el otro.

Como en este espacio de nombres los *sockets* residen en el sistema de ficheros, para asignarles un nombre es preciso considerar los requisitos impuestos por el sistema: es necesario tener permiso de escritura en el directorio en que se quiere crear el *socket*, no puede haber otra entrada (de cualquier tipo: fichero, directorio, *socket*) con el mismo nombre, etc. En UNIX es habitual crear los *sockets* de esta clase en el directorio /tmp.

Un ejemplo...

... de aplicación UNIX que utiliza los *sockets* del dominio de ficheros es el sistema de ventanas X, que en muchas implementaciones los utiliza para comunicar a los clientes con el servidor cuando corren en el mismo ordenador.

1.3.2. El espacio de nombres Internet

El espacio de nombres Internet se denomina de este modo por la familia de protocolos que se utilizan en la comunicación entre los *sockets* (básicamente TCP y UDP). La dirección de un *socket* en este espacio de nombres consta, además del protocolo concreto que debe utilizarse, de los dos componentes siguientes:

- La **dirección de red del nodo**, es decir, la dirección Internet de la máquina correspondiente.

  Las direcciones Internet de las máquinas se representan por medio de los números de 32 bits (4 *bytes*) utilizados en el protocolo IP. Como puede haber máquinas conectadas a más de una red IP, una máquina puede disponer de múltiples direcciones Internet; sin embargo, a cada dirección debe corresponderle una sola máquina.

Consultad las direcciones utilizadas en el protocolo IP en el subapartado 3.1 del módulo didáctico "TCP/IP: los protocolos de la red Internet" de esta asignatura.

- Un **número de puerto**. Sirve para distinguir los *sockets* de una máquina determinada (para un protocolo determinado).

  Tanto en TCP como en UDP, los números de puerto son cantidades de 16 bits, de manera que su valor puede estar comprendido entre 1 y 65.535 (en muchos sistemas, el número 0 no corresponde a un puerto válido, sino que representa un puerto indeterminado).

La gran mayoría de las aplicaciones que utilizan *sockets* siguen el modelo cliente/servidor. En este último, una de las partes, la que solicita un determinado servicio (el cliente), inicia la comunicación, mientras que la otra parte, la que proporciona el servicio (el servidor), espera que le lleguen las peticiones de servicio y las responde.

El modelo cliente/servidor se explica detalladamente en el capítulo "El modelo cliente-servidor" del módulo didáctico "Aplicaciones Internet" de esta asignatura.

Socket servidor

Cuando se crea un *socket* que debe actuar como destino de una comunicación (*socket* servidor), es importante asignarle una dirección bien determinada, puesto que los *sockets* de origen (clientes) la necesitarán para poder establecer la conexión. Por norma general, la parte correspondiente a la dirección del servidor ya estará fijada y, por tanto, será preciso preocuparse simplemente por el número de puerto.

A la hora de asignar un número de puerto a un *socket*, es conveniente considerar que en el sistema operativo UNIX se encuentran definidos dos rangos de puertos: el correspondiente a los **puertos reservados**, con números entre el 1 y el 1.023, y el resto de los puertos, con números entre el 1.024 y el 65.535. Un proceso cualquiera por lo general no puede asignar un número de puerto reservado a un *socket*; el sistema sólo permite hacer esto último a los procesos con privilegios de superusuario. El motivo de esta restricción es que los puertos reservados se utilizan habitualmente para los servicios estándar como, por ejemplo, `telnet`, `ftp`, `rlogin`, `rsh`, `finger`, etc.

Si un proceso asignara a un *socket* uno de estos puertos, podría recibir conexiones de otros procesos que creerían que se conectaban al proceso servidor "oficial" y enviarles información falsa o recibir información secreta*.

*** Por ejemplo, contraseñas (*passwords*).**

La tabla siguiente muestra los números de puerto TCP asignados a algunos de los servicios Internet oficiales:

Servicio	Puerto
`ftp`	21
`telnet`	23
`smtp (correo)`	25
`finger`	79
`http (WWW)`	80
`nntp (noticias)`	119
`rlogin`	513
`rsh`	514

Socket cliente

Cuando se crea un *socket* para utilizarlo como origen de una comunicación (*socket* cliente), su número de puerto suele ser irrelevante. Lo que es más habitual es dejar que el sistema asigne uno de manera automática, puesto que la dirección del *socket* cliente sólo la necesita el *socket* servidor para saber a dónde debe enviar los datos intercambiados.

Existen algunos casos en los que es necesario asignar un número de puerto específico a un *socket* cliente. Salvo situaciones especiales, por norma general sólo será preciso asignar un número de puerto a un *socket* cuando deba actuar como servidor.

Las aplicaciones `rsh` y `rlogin` constituyen dos ejemplos de estos casos especiales. Estas últimas permiten que un usuario acceda a un sistema remoto sin necesidad de identificarse explícitamente*. Los servidores correspondientes basan la autenticación en la identidad del usuario en el sistema cliente y simplemente comprueban que esté en una lista de usuarios autorizados. Por consiguiente, es capital que la identidad que se envía al sistema remoto no se pueda falsificar con facilidad. Por ello, estos servidores sólo permiten el acceso sin identificación cuando la conexión proviene de un puerto reservado. De este modo, existe cierta garantía de que el proceso que pide la conexión disponga de suficientes privilegios para que se pueda confiar en la información que proporciona, es decir, en la identidad del usuario en cuyo nombre solicita el acceso.

*** Una manera de identificarse sería con un nombre y una contraseña.**

Se debe tener en cuenta que, si, al asignar un número de puerto a un servidor, se elige un número cualquiera, existe la posibilidad de que este último ya esté asignado a un *socket* cliente al que el sistema haya proporcionado, como hemos visto con anterioridad, una dirección automática. Para evitar esto, muchos sistemas dividen el rango de los números de puerto no reservados en dos subrangos y garantizan que las direcciones generadas automáticamente siempre estén en el primer subrango. De este modo, para que no haya conflictos, sólo es preciso elegir para los servidores no estándar números de puerto que estén en el segundo subrango.

De hecho, los protocolos de transporte no impiden que dos *sockets* de un mismo sistema utilicen el mismo número de puerto, siempre que no se intenten comunicar con el mismo *socket* remoto. Sin embargo, por norma general, los sistemas no permiten reutilizar números de puerto; es decir, asignar a un *socket* un número que ya esté ocupado, salvo que se utilice una opción específica de los *sockets* con esta finalidad.

1.4. Estilos de comunicación

Al establecer un canal de comunicación entre dos procesos por medio de un *socket*, uno de los parámetros que se debe especificar es el **estilo** que se utilizará en esta comunicación. Básicamente, existen dos estilos de comunicación posibles (llamados también *tipos de sockets*):

- El estilo secuencia de *bytes*.
- El estilo datagrama.

1.4.1. Estilo secuencia de *bytes*

En los *sockets* que utilizan el estilo secuencia de bytes, los datos se transmiten de extremo a extremo como una corriente o flujo ordenado de bytes.

Asimismo, este estilo se puede llamar *orientado a conexión*, puesto que la comunicación consiste en establecer previamente una conexión con el *socket* remoto y después utilizarla para intercambiar los datos. Cuando se utiliza el espacio de nombres Internet, este estilo se corresponde con el protocolo TCP.

El protocolo de transporte utilizado en el estilo secuencia de bytes debe garantizar que los datos lleguen al otro extremo en el mismo orden en que se han enviado y sin pérdidas ni repeticiones.

Es posible que el protocolo permita la transmisión de datos **fuera de banda**. Por norma general, estos últimos se utilizan para enviar información urgente, puesto que se supone que el destinatario, cuando los reciba, los procesará antes que los datos normales que tenga pendientes de leer.

1.4.2. Estilo datagrama

En los *sockets* que utilizan el estilo datagrama, cada vez que el emisor escribe datos en los mismos se transmite un paquete individual o datagrama con dichos datos, y cada vez que el destinatario quiere leer estos últimos recibe como máximo un datagrama.

Todos los datagramas contienen la dirección del *socket* al que van dirigidos, de manera que se pueden direccionar independientemente. Ello hace que algún datagrama pueda llegar a su destino después de otro que se ha enviado más tarde y, por tanto, no hay garantías sobre el orden en que se recibirán los datos en el otro extremo. Asimismo, puede suceder que algún datagrama se pierda o llegue duplicado, mientras que los que han seguido otro camino lleguen correctamente.

Este estilo también se denomina *no orientado a conexión*, puesto que, por el hecho de utilizar paquetes direccionados individualmente a su destino, no es preciso establecer ninguna conexión previa con el *socket* remoto. Cuando se utiliza el espacio de nombres Internet, dicho estilo se corresponde con el protocolo UDP.

Con el uso de datagramas no se garantiza la llegada de todos los paquetes, ni su orden, ni su unicidad.

La ventaja de este estilo de comunicación radica en que requiere muchos menos recursos que el estilo secuencia de bytes (no se necesitan números de secuencia en las cabeceras, *buffers* y ventanas de transmisión y recepción, confirmaciones –*acknowledgements*–, retransmisiones, etc.).

Los datagramas son apropiados cuando no es indispensable la fiabilidad total en la recepción de los datos, por ejemplo cuando basta con un mecanismo sencillo de recuperación de los errores (como puede ser reenviar un paquete si no se ha recibido ninguna respuesta después de cierto tiempo). En este caso, los mecanismos de corrección, si existen, se implementan en el nivel de aplicación.

En cambio, cuando se necesita asegurar la recepción ordenada de los datos sin pérdidas ni repeticiones, no es conveniente que la aplicación utilice datagramas y se intente recuperar de los errores por sí misma, sino que es más eficiente encargar esta tarea a los niveles inferiores (es decir, utilizar un protocolo de transporte que garantice la fiabilidad).

1.5. La comunicación entre *sockets*

Para poder establecer una comunicación entre dos *sockets*, las dos partes involucradas, cliente y servidor, deben efectuar una serie de operaciones, antes y después de llevar a cabo el intercambio de información.

Las operaciones que debe efectuar el servidor son las siguientes:

- Crear un *socket* en el espacio de nombres y con el protocolo deseados.
- Asignar una dirección al *socket*.
- Dejar el *socket* preparado para recibir conexiones y crear una cola en la que se irán guardando las peticiones que le lleguen (sólo en los protocolos orientados a conexión). Esta operación se conoce como *establecer una conexión pasiva*.
- Aceptar una petición de conexión de la cola o esperar hasta que llegue una si no hay ninguna (sólo en los protocolos orientados a conexión).
- Intercambiar datos con el cliente que ha solicitado la comunicación leyendo los datos y escribiéndolos en el *socket*.
- Cerrar el *socket* cuando haya finalizado el intercambio.

Por otro lado, las operaciones que debe efectuar el cliente son las siguientes:

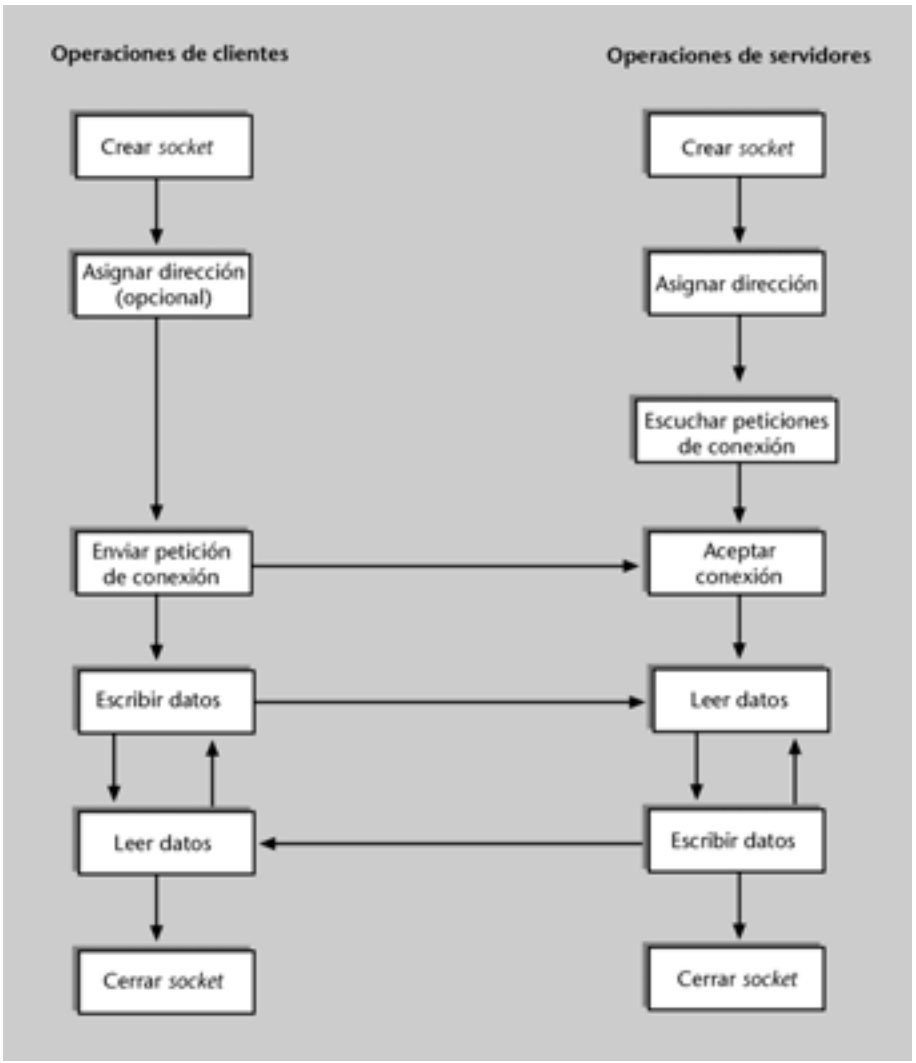
- Crear un *socket* en el espacio de nombres y con el protocolo deseados.
- Asignar una dirección al *socket* (si la familia de protocolos no lo requiere, como en el caso de los protocolos Internet, este paso es opcional).
- Enviar una petición de conexión al servidor (sólo en los protocolos orientados a conexión). Esta operación se conoce como *establecer una conexión activa*.

- Intercambiar datos con el servidor leyéndolos y escribiéndolos en el *socket*.

- Cerrar el *socket* cuando haya finalizado el intercambio.

La figura siguiente recoge las operaciones correspondientes a los clientes y a los servidores (en el caso de los protocolos orientados a conexión):

El capítulo 2 de este módulo muestra cómo se realizan estas operaciones en lenguaje C y el capítulo 3, en Java.



2. *Sockets* con lenguaje C

A continuación, veremos con detalle todas las operaciones que hemos descrito en el apartado anterior y las llamadas correspondientes proporcionadas por la interfaz de programación de los *sockets* en UNIX, agrupadas en tres categorías: las comunes a servidores y clientes, las propias de los servidores y las propias de los clientes.

2.1. Operaciones comunes a servidores y clientes

2.1.1. Crear un *socket*

El prototipo de la llamadas básica para crear un *socket* está declarado en el fichero cabecera `<sys/socket.h>`:

El fichero `<sys/socket.h>`

En algunos sistemas, es necesario haber incluido el fichero cabecera `<sys/types.h>` antes de incluir el `<sys/socket.h>`.

```
int socket (int espacio, int estilo, int protocolo);
```

Los parámetros que utiliza esta llamada son los siguientes:

1) El parámetro **`espacio`** especifica el espacio de nombres o familia de protocolos correspondiente al *socket*. El mismo fichero cabecera `<sys/socket.h>` define las constantes siguientes para utilizarlas como valor de este parámetro:

- `PF_FILE`: representa la familia de protocolos del espacio de nombres de ficheros.
- `PF_UNIX`: es un sinónimo de `PF_FILE` definido por compatibilidad con versiones antiguas de la librería de *sockets*.
- `PF_INET`: representa la familia de protocolos Internet.
- `PF_NS, PF_ISO, PF_CCITT, PF_IMPLINK, PF_ROUTE`, etc., representan otras familias de protocolos, no tan utilizadas como las anteriores, al menos por medio de *sockets*, y que, por tanto, no estudiaremos aquí: son los protocolos Network Software de Xerox, protocolos OSI, etc. Muchos sistemas, aunque proporcionan las definiciones de estos símbolos, no tienen implementado el acceso a los protocolos respectivos por medio de la interfaz de los *sockets* (si se intenta crear un *socket* de alguna de estas familias, la llamada retorna error).

2) El parámetro **`estilo`** especifica el estilo de comunicación que se quiere utilizar en el *socket*. El fichero cabecera `<sys/socket.h>` proporciona las constantes siguientes para el valor de dicho parámetro:

- `SOCK_STREAM`: representa el estilo de comunicación secuencia de bytes.

- `SOCK_DGRAM`: representa el estilo de comunicación datagrama.

- `SOCK_RAW`: representa un estilo con acceso directo al nivel de red que permite enviar datagramas arbitrarios (una aplicación que utilice este estilo puede definir, por ejemplo, su formato de cabeceras de los paquetes).

- `SOCK_SEQPACKET` (datagramas transferidos de manera fiable), `SOCK_RDM` (*reliably delivered message*, también basado en datagramas), etc., representan estilos de comunicación que tampoco estudiaremos aquí.

SOCK_RAW

En UNIX se permite la creación de *sockets* de este tipo sólo en los procesos con privilegios de superusuario.

3) El parámetro **`protocolo`** especifica el protocolo concreto que se quiere utilizar en el *socket* por medio del número asociado a dicho protocolo.

Dada una familia de protocolos (primer parámetro), el estilo de comunicación (segundo parámetro) determina qué subconjunto de los mismos se puede utilizar en un *socket*. Dentro de este subconjunto, a cada protocolo le corresponde un número preestablecido que se especifica por medio del parámetro **`protocolo`**.

Cuando, para una familia y un estilo dados, sólo hay un protocolo definido, simplemente es necesario proporcionar el número 0 como valor del tercer parámetro. Cuando hay más de uno, el número 0 representa el protocolo por defecto o el más utilizado.

Protocolo 0

El espacio de nombres de ficheros sólo soporta un único protocolo para cada estilo de comunicación y se selecciona con el número 0. Y, en el espacio de nombres Internet, el número 0 selecciona el protocolo TCP para el estilo secuencia de bytes y el protocolo UDP para el estilo datagrama.

Ejemplo de función para crear un *socket* TCP

```
int crear_socket_TCP(void)
{
return socket(PF_INET, SOCK_STREAM, 0);
}
```

Ejemplo de función para crear un *socket* UDP

```
int crear_socket_UDP(void)
{
return socket(PF_INET, SOCK_DGRAM, 0);
}
```

El valor retornado por la llamada `socket` es un entero que, si no es negativo, representa el descriptor de fichero asociado al *socket* creado. Si vale –1, indica que se ha producido un error y no se ha podido crear el *socket*.

Cuando el número retornado represente un descriptor de fichero, será preciso guardarlo para poder especificar sobre qué *socket* se quieren efectuar las operaciones posteriores. Este número pertenece al mismo espacio de descriptores que los correspondientes a ficheros o dispositivos que un proceso haya abierto, por ejemplo, con la llamada `open`. Es decir, el descriptor asociado a un *socket* creado por un proceso nunca coincidirá con el de un fichero abierto por el mismo proceso. Asimismo, muchas de las llamadas al sistema que admiten un descriptor de fichero también funcionan cuando el descriptor representa un *socket*: `read`, `write`, `close`, etc.

Los *sockets* creados con la llamada `socket` en un inicio no tienen ninguna dirección asignada, ni están conectados a ningún otro *socket*. Hay otra llamada que permite la creación de una pareja de *sockets* ya conectados entre sí. Su prototipo, también declarado en el fichero cabecera `<sys/socket.h>`, es el siguiente:

```
int socketpair(int espacio, int estilo,
          int protocolo, int vs[2]);
```

Los tres primeros parámetros, **`espacio`**, **`estilo`** y **`protocolo`**, son análogos a los de la llamada `socket`. Es preciso considerar, sin embargo, que muchos sistemas sólo soportan el espacio de nombres de ficheros en la llamada `socketpair`.

El parámetro **`vs`** debe ser la dirección de un vector de enteros que disponga de al menos dos elementos. Si la llamada tiene éxito, retorna 0 y llena los dos primeros elementos del vector `vs` (`vs[0]` y `vs[1]`) con los descriptores de los *sockets* creados. Estos dos *sockets* no tendrán dirección o nombre; sin embargo, ya estarán conectados el uno con el otro y, por tanto, preparados para empezar a intercambiar datos (si conviene, siempre se le puede asignar una dirección, aunque, por norma general, no es necesario). Si la llamada no tiene éxito, indica que se ha producido un error retornando el valor `-1`.

Así pues, se puede crear una pareja de *sockets* con una sola llamada; sin embargo, ambos estarán necesariamente en el mismo sistema y, asimismo, corresponderán al mismo proceso. Por tanto, puede parecer que esta llamada no es demasiado útil, sobre todo si se tiene en cuenta que el objetivo principal de los *sockets* es la comunicación entre procesos. Sin embargo, existe un tipo de aplicación en el que es conveniente el uso de estas parejas de *sockets*: la comunicación entre un proceso padre y un proceso hijo.

Aprovechando que, en UNIX, cuando un proceso crea un hijo, este último hereda sus descriptores abiertos, el proceso padre suele crear en primer lugar los *sockets* con la llamada `socketpair` y, después, genera el proceso hijo con la llamada `fork`. Entonces, para comunicarse entre sí, el padre utiliza uno de los dos descriptores y el hijo, el otro,y así se ahorran la asignación de direcciones y el establecimiento de la conexión.

Cuando retornan con error, tanto `socket` como `socketpair` utilizan el método habitual en la interfaz de llamadas al sistema desde C para indicar su causa: asignar a una variable global denominada **`errno`** un entero que representa un código de error. La declaración de dicha variable y las definiciones de las constantes que representan sus valores posibles están contenidas en el fichero cabecera `<errno.h>`.

Los códigos de error concretos pueden variar de un sistema a otro; sin embargo, existe una serie de causas típicas por las que pueden fallar las llamadas `socket` y `socketpair`. Los valores que, por norma general, encontraremos

en la variable `errno` (según las constantes definidas en el fichero `<errno.h>`), aparte de los que representan errores por memoria insuficiente o exceso de descriptores abiertos, son los siguientes:

- `EAFNOSUPPORT`: la librería de *sockets* no soporta el espacio de nombres especificado.
- `EPROTONOSUPPORT`: la librería de *sockets*, para el espacio de nombres especificado, no soporta la combinación de estilo de comunicación y número de protocolo.
- `EACCESS`: el proceso no tiene privilegios para crear un *socket* con el estilo y/o el protocolo especificados.
- `EOPNOTSUPP` (sólo para la llamada `socketpair`): el protocolo especificado no soporta la creación de parejas de *sockets*.

2.1.2. Asignar dirección a un *socket*

Antes de estudiar la llamada que permite asignar direcciones a los *sockets*, veremos cómo se representan estas direcciones en la interfaz de programación.

Formatos de las direcciones

El fichero cabecera `<sys/socket.h>` proporciona la definición del tipo `struct sockaddr`:

```
struct sockaddr
{
   short int sa_family;
   char sa_data[...];
};
```

Este tipo representa las direcciones de los *sockets* en general; sin embargo, por norma general, no se utiliza, puesto que los que se utilizan son los tipos específicos de cada espacio de nombres.

El tipo `struct sockaddr` contiene dos campos:

- El campo **`sa_family`** indica el espacio de nombres o familia de protocolos y, por tanto, el formato de la dirección representada. El fichero cabecera `<sys/socket.h>` define las constantes que es preciso adoptar como valor de este campo, que se corresponden una a una con las constantes definidas para las familias de protocolos:

– `F_FILE`: representa el formato de direcciones del espacio de nombres de ficheros `PF_FILE`.

– AF_UNIX: es un sinónimo de AF_FILE (de la misma manera que PF_UNIX lo es de PF_FILE).

– AF_INET: representa el formato de direcciones del espacio de nombres Internet PF_INET.

– AF_UNSPEC: es un valor especial que no representa ningún formato de dirección y que se utiliza en ciertos casos muy concretos (por analogía, en algunos sistemas se define el símbolo PF_UNSPEC, pero no tiene ningún uso).

AF_ y PF_

Para todos los otros símbolos que empiezan por PF_ se encuentra definido el símbolo correspondiente, que empieza por AF_. Lo que es más habitual es que los valores de las constantes que empiezan por AF_ sean idénticos a los de las respectivas constantes que empiezan por PF_.

- El campo **sa_data** es un contenedor genérico de la dirección que se quiere representar. La longitud de este último es irrelevante porque los programas nunca trabajan directamente con variables del tipo struct sockaddr, sino con punteros que apuntan a este tipo; es decir, con el tipo propio del espacio de nombres correspondiente, que determina cómo está estructurada la información de la dirección (qué campos contiene, etc.).

Los tipos correspondientes a cada espacio de nombres son los siguientes:

a) En el espacio de nombres de ficheros, o dominio UNIX, se utiliza el tipo struct sockaddr_un, definido en el fichero cabecera <sys/un.h> de la manera siguiente:

```
struct sockaddr_un
{
   short int sun_family;
   char sun_path[UNIX_PATH_MAX];
};
```

UNIX_PATH_MAX

La constante UNIX_PATH_MAX determina la longitud máxima del campo sun_path en la dirección de un *socket* UNIX, y su valor en muchos sistemas, por motivos históricos, es 108.

– El campo **sun_family** sirve para identificar el formato de la dirección, y su valor debe ser AF_FILE.

– El campo **sun_path** es una cadena de caracteres cuyo valor es el nombre del *socket*.

b) En el espacio de nombres Internet, se utiliza el tipo struct sockaddr_in, definido en el fichero cabecera <netinet/in.h> de la manera siguiente:

```
struct sockaddr_in
{
   short int sin_family;
   struct in_addr sin_addr;
   unsigned short int sin_port;
};
```

El fichero <netinet/in.h>

Como en el caso del fichero <sys/socket>, en algunos sistemas es necesario haber incluido el fichero <sys/types.h> antes de incluir el <netinet/in.h>.

– El campo **`sin_family`** sirve para identificar el formato de la dirección, y su valor debe ser `AF_INET`.

– El campo **`sin_addr`** se utiliza para especificar la dirección Internet del servidor representada con el tipo **`struct in_addr`**, que se define en el mismo fichero `<netinet/in.h>` de la manera siguiente:

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

Es decir, este tipo contiene un campo, **`s_addr`**, que tiene por valor los 4 bytes de la dirección IP almacenados en el orden de la red.

– El campo **`sin_port`** representa el número de puerto y tiene 2 bytes almacenados también en el orden de la red.

`sin_addr`

El motivo por el cual el campo `sin_addr` es una estructura de un solo miembro, en lugar de ser directamente un entero de 4 bytes es histórico: en versiones anteriores, este campo era una unión de otros campos que permitían acceder a los bytes de diferentes maneras.

En las cabeceras de los datagramas IP y de los paquetes TCP y UDP, algunos campos representan números de más de un byte. Estos protocolos establecen un orden canónico para la transmisión de los bytes, el orden de la red: el primer byte es el de más peso, y el último, el de menos peso. La interfaz de programación de los *sockets* requiere que los valores correspondientes a estos campos estén almacenados en la memoria en el orden de la red.

En algunos ordenadores la representación interna en la memoria de los números enteros puede coincidir con el orden de la red; sin embargo, en otros puede ser a la inversa (de menor a mayor peso). Para realizar la conversión entre el orden interno y el de la red, la interfaz de los *sockets* proporciona cuatro funciones, cuyos prototipos se declaran en el fichero cabecera `<netinet/in.h>` de la manera siguiente:

Big/little endian

En ocasiones, de los ordenadores que almacenan los *bytes* de mayor peso a menor, se dice que son *big endian* y, de los que los almacenan de menor peso a mayor, *little endian*.

- `unsigned short int htons(unsigned short int n2_intern);`
- `unsigned long int htonl(unsigned long int n4_intern);`
- `unsigned short int ntohs(unsigned short int n2_red);`
- `unsigned long int ntohl(unsigned long int n4_red);`

Las dos primeras reciben como argumento un número, de 2 y 4 bytes respectivamente, en la representación interna de la máquina (es decir, del ordenador local), y retornan la representación del mismo número en el orden de la red. Las otras dos funciones llevan a cabo la conversión inversa: pasan del orden de la red a la representación interna.

En los ordenadores en que el orden interno de los bytes coincida con el de la red, estas funciones retornarán sus argumentos sin modificar; sin embargo, conviene utilizarlas siempre para garantizar la portabilidad de los programas.

Relleno de una dirección Internet con unos valores concretos

La función siguiente llena una dirección Internet (primer parámetro) con los valores que se le pasan (dirección IP en el segundo parámetro y número de puerto en el tercero):

```
void llenar_dir_internet(struct sockaddr_in *adr,
unsigned long int adr_host, unsigned short int n_port)
{
adr->sin_family = AF_INET;
adr->sin_addr.s_addr = htonl(adr_host);
adr->sin_port = htons(n_port);
}
```

El fichero cabecera `<netinet/in.h>` proporciona, además de las funciones ya mencionadas, las definiciones de constantes siguientes, que se pueden usar como valores especiales del campo `s_addr` de una struct `in_addr`:

- `INADDR_LOOPBACK`: equivale a la dirección IP 127.0.0.1, que corresponde al nombre del ordenador local *(localhost)*. Las comunicaciones con esta dirección por norma general no utilizan el dispositivo lógico de la red, que es innecesario cuando un ordenador se envía datos a sí mismo, sino el llamado *loopback*.

- `INADDR_BROADCAST`: equivale a la dirección IP que es preciso utilizar para enviar mensajes de difusión.

- `INADDR_ANY`: representa una dirección IP indeterminada. Se utiliza en los *sockets* pasivos para indicar que pueden recibir peticiones de conexión destinadas a cualquier dirección IP si el ordenador tiene más de una. Si el ordenador que debe recibir las peticiones tiene una sola dirección IP, también se puede utilizar para representar esta última sin necesidad de especificar su valor.

El mismo fichero `<netinet/in.h>` también define las constantes siguientes para representar valores especiales de los números de puerto:

- `IPPORT_RESERVED`: los números más pequeños de esta constante (1.024) corresponden a puertos reservados.

- `IPPORT_USERRESERVED`: en los sistemas que dividen el rango de los puertos no reservados en dos subrangos, los números inferiores a esta constante* pertenecen al primer subrango, y los mayores o iguales, al segundo. Ello significa que los puertos asignados automáticamente estarán comprendidos entre los valores `IPPORT_RESERVED` e `IPPORT_USERRESERVED - 1`, mientras que los servidores de usuario (no estándar) pueden utilizar los puertos a partir de `IPPORT_USERRESERVED`.

*** El valor de `IPPORT_USERRESERVED` puede variar de un sistema a otro; sin embargo, por norma general suele ser 5.000.**

Vale la pena remarcar que, en todos los tipos que se utilizan para representar direcciones de *sockets*, el primer campo siempre es un `short int` que identifica el formato de la dirección. Así, a partir de su valor se puede saber cómo debe interpretarse el resto de la información.

Asignación de dirección

La llamada para asignar una dirección a un *socket* posee el prototipo siguiente, declarado en el fichero cabecera `<sys/socket.h>`:

```
int bind(int descr, const struct sockaddr *adr,
                  size_t long_adr);
```

El tipo `size_t`

El tipo `size_t`, utilizado en la librería C para representar números de bytes, está definido en el fichero cabecera `<sys/types.h>` y, por lo general, equivale a `unsigned int`.

- El parámetro **`descr`** es el descriptor del *socket*.
- El parámetro **`adr`** es un puntero a la estructura que representa la dirección que debe asignarse.
- El parámetro **`long_adr`** indica cuántos bytes ocupa esta estructura, por si su longitud es variable (como el caso del espacio de nombres de ficheros).

Como ya hemos comentado con anterioridad, para representar la dirección no se suele utilizar una variable del tipo **`struct sockaddr`**. En su lugar, se utiliza una variable del tipo propio del espacio de nombres correspondiente y, habitualmente, se pasa como segundo parámetro de la llamada **`bind`**, a la que se aplica una conversión explícita de tipo **`(cast)`**.

Asignación de un nombre a un *socket* del espacio de nombres de fichero

```
int asignar_dir_fichero(int sock, const char *nombre)
{
   struct sockaddr_un adr;

   adr.sun_family = AF_FILE;
   strcpy(adr.sun_path, nom);
   return bind(sock, (struct sockaddr *)&adr,
      sizeof adr.sun_family + strlen(nom) + 1);
}
```

En este caso, el tercer parámetro de la llamada `bind` se calcula como la suma de la longitud del campo `sun_family` más la longitud efectiva del campo `sun_path` (es decir, el número de caracteres de la cadena `nombre` incluyendo el '`\0`' que marca su final).

Asignación de un nombre a un *socket* del espacio Internet

Esta función asigna una dirección a un *socket* del espacio Internet utilizando la función `asignar_dir_internet`:

Recordad

En el espacio de nombres Internet, por lo general no es preciso asignar ninguna dirección a un *socket* si debe actuar como cliente, puesto que el sistema le asignará una automáticamente cuando la necesite.

```
int asignar_dir_internet(int sock,
   unsigned long int adr_host, unsigned short int n_port)
{
   struct sockaddr_in adr;
```

```
    llenar_dir_internet(&adr, adr_host, n_port);
    return bind(sock, (struct sockaddr *)&adr, sizeof adr);
}
```

El valor retornado por la llamada `bind` es 0 si la dirección se ha podido asignar correctamente. Si se produce algún error, la llamada retorna -1 y asigna a la variable global `errno` un valor que, por norma general, será uno de los siguientes:

- `EBADF`: el primer parámetro no es un descriptor válido.
- `ENOTSOCK`: el primer parámetro es un descriptor; sin embargo, no corresponde a un *socket*.
- `EADDRNOTAVAIL`: no se puede asignar la dirección especificada desde este sistema (por ejemplo, la dirección IP del segundo parámetro no corresponde a ninguna de las del ordenador local).
- `EADDRINUSE`: la dirección especificada ya está asignada a otro *socket*.
- `EACCESS`: el proceso no posee suficientes privilegios para asignar la dirección especificada (en el espacio de ficheros, porque los permisos de los directorios lo prohíben o, en el espacio Internet, porque el número de puerto del segundo parámetro corresponde a un puerto reservado).
- `EINVAL`: el primer parámetro es incorrecto porque corresponde a un *socket* que ya tiene dirección asignada, o bien el tercer parámetro es incorrecto porque no corresponde a una longitud de dirección válida en el espacio de nombres del *socket*.

Además de estos posibles errores, en el espacio de números de ficheros también se pueden dar los errores propios de la creación de ficheros, puesto que el hecho de asignar dirección a un *socket* de este espacio implica crear una entrada de directorio. Entre estos errores, podemos citar, por ejemplo, `ENOENT` (alguno de los componentes del camino de subdirectorios especificado en el nombre no existe) y `ENOTDIR` (alguno de los componentes del camino existe, pero no es un subdirectorio).

2.1.3. Enviar datos

Para enviar datos, podemos utilizar tres llamadas diferentes:

1) La llamada **`write`**. A un *socket* que ya tenga establecida una conexión con otro se le puede aplicar esta llamada para escribir datos; es decir, para enviarlos al *socket* remoto. Si este úlimo es no orientado a conexión, pero tiene definida una dirección de destino por defecto, también se le puede aplicar esta llamada.

En los sistemas UNIX, el prototipo de la llamada `write` está declarado en el fichero cabecera `<unistd.h>` de la manera siguiente:

```
ssize_t write(int descr, const void *datos,
                  size_t longitud);
```

El tipo `ssize_t`

El tipo `ssize_t`, definido también en el fichero cabecera `<sys/types.h>`, es como `size_t`, pero con signo (por norma general, equivale a `int`), y se utiliza en la librería C para valores que pueden representar un número de bytes si son positivos, o un código de error si son negativos.

- El primer parámetro, **`descr`**, es un descriptor que, por norma general, puede corresponder a un fichero, un dispositivo, una *pipe*, un *socket*, etc.
- El segundo parámetro, **`datos`**, es la dirección de memoria donde se encuentran los datos que queremos escribir.
- El tercer parámetro, **`longitud`**, indica cuántos *bytes* queremos escribir.

El valor retornado constituye el número de *bytes* que se han escrito en realidad (que, por norma general, será igual al tercer parámetro, pero podría ser menor), salvo que valga –1, lo que significa que se ha producido un error.

2) La llamada **`send`**. Es específica de los *sockets*. Su prototipo está declarado en el fichero cabecera `<sys/socket.h>` de la manera siguiente:

```
int send(int descr, const void *datos, size_t longitud,
                    int flags);
```

Los tres primeros parámetros y el valor retornado tienen el mismo significado que en la llamada `write`.

Los *sockets* a los que se les puede aplicar la llamada `send` también son los mismos que los de la llamada `write`; es decir, los que estén conectados o tengan definido un destino por defecto. La única diferencia entre ambas llamadas es que `send` admite un parámetro adicional, `flags`, que permite especificar ciertas opciones de transmisión. Cuando no se utiliza ninguna de estas opciones, el valor del parámetro `flags` es 0 y entonces la llamada `send` actúa exactamente igual que la llamada `write`.

La lista de opciones se representa con una combinación de bits en la que cada bit igual a 1 indica que se quiere aplicar una opción determinada. El fichero cabecera `<sys/socket.h>` define una serie de símbolos que pueden combinarse directamente con la operación OR binaria (o bien con una suma) para obtener el conjunto de bits deseados. Los símbolos correspondientes a las opciones aplicables a la llamada `send` son los dos siguientes:

- `MSG_OOB`: indica que los datos deben enviarse como datos urgentes; es decir, fuera de banda. Sólo tiene efecto si el protocolo asociado al *socket* soporta la transmisión de datos urgentes*.

*** El protocolo TCP es uno de los que pueden soportar la transmisión urgente de datos.**

- `MSG_DONTROUTE`: indica que no se incluya información de direccionamiento en el mensaje transmitido (esta opción sólo es interesante cuando se trabaja en el nivel de red o se quieren realizar pruebas para obtener diagnósticos).

3) La llamada **`sendto`**. Su prototipo* también está declarado en el fichero cabecera `<sys/socket.h>`:

* Esta llamada también permite enviar datos por medio de un *socket*.

```
int sendto(int descr, const void *datos, size_t longitud,
int flags, const struct sockaddr *adr, size_t long_adr);
```

La llamada `sendto` se diferencia de la `send` en dos parámetros adicionales, que permiten especificar a qué *socket* deben enviarse los datos por medio de un puntero a su dirección (**`adr`**) y el número de *bytes* que esta última ocupa (**`long_adr`**).

Si el *socket* correspondiente a `descr` es no orientado a conexión, esta llamada envía al *socket* indicado por los parámetros `adr` y `long_adr` un datagrama con los datos especificados. Si el *socket* está conectado, estos parámetros se ignoran.

En muchos sistemas, estas tres llamadas (`write`, `send` y `sendto`) por norma general retornan cuando los datos que deben enviarse se han copiado en el *buffer* de transmisión del *socket*. Si el *buffer* está lleno, las llamadas esperan hasta que haya suficiente espacio disponible, salvo que el *socket* trabaje en el llamado *modo no bloqueante*. Por consiguiente, un retorno sin error de cualquiera de estas llamadas no implica necesariamente una recepción correcta en el *socket* remoto.

Modo no bloqueante

Como en cualquier descriptor de fichero en general, el modo de trabajo de los *sockets* por defecto es bloqueante; sin embargo, se puede cambiar a no bloqueante por medio de la llamada fcntl, con la operación `F_SETFL` y el indicador `O_NONBLOCK`.

En los *sockets* no orientados a conexión es posible, incluso, que la dirección de destino no exista. En este caso, la red no sabrá qué hacer con los datagramas una vez han salido de la máquina origen.

Cuando estas llamadas retornan con error, los valores que solemos encontrar en la variable global `errno` son los siguientes:

- `EBADF`: el primer parámetro no es un descriptor válido.
- `ENOTSOCK` (sólo para las llamadas `send` y `sendto`): el primer parámetro es un descriptor, pero no corresponde a un *socket*.
- `EINVAL` (sólo para la llamada `sendto`): el sexto parámetro es incorrecto porque no corresponde a una longitud de dirección válida en el espacio de números del *socket*.
- `ENOTCONN`: el *socket* es orientado a conexión, pero no está conectado.
- `EDESTADDRREQ` (sólo para las llamadas `write` y `send`): el *socket* no tiene destino por defecto.
- `EMSGSIZE`: la longitud de los datos que se precisa enviar es demasiado grande para el protocolo asociado al *socket*.
- `EWOULDBLOCK`*: el *socket* es no bloqueante y los datos no se pueden enviar inmediatamente (la memoria intermedia está llena).

* En muchos sistemas, **`EAGAIN`** es un sinónimo de **`EWOULDBLOCK`**.

- `EPIPE`: el *socket* estaba conectado; sin embargo, ya no hay conexión establecida (por ejemplo, porque se ha cerrado desde el otro extremo). En este caso, el proceso recibe primero una señal `SIGPIPE`. La acción que provoca por defecto la recepción de esta señal por defecto es la culminación del proceso y, por tanto, la llamada no retorna. Si la señal `SIGPIPE` está inhabilitada o la acción se ha reprogramado, y la llamada finalmente retorna, entonces el código de error es `EPIPE`.

- `EINTR`: no se ha completado la operación porque ha llegado una señal no ignorada.

La acción con que un proceso responde a una determinada señal puede cambiarse, por ejemplo, con la función `signal`.

Esta causa de error es propia de las llamadas al sistema bloqueadoras; es decir, las que pueden necesitar cierto tiempo para completarse (puesto que deben esperar hasta que se produzca cierta condición, como que se vacíe el *buffer* de transmisión). Si, mientras una llamada está bloqueada, el proceso recibe una señal no ignorada, la llamada se interrumpe para ejecutar el gestor (*handler*) asociado a esta señal.

Según el sistema, puede ser que, al acabar la ejecución del gestor, la llamada pendiente no continúe, sino que retorne de inmediato con la variable `errno` igual a `EINTR`. Por tanto, conviene que tengamos en cuenta esta situación si en un programa queremos efectuar llamadas que se puedan bloquear y nos interesa tener "capturada" alguna señal.

La función `siginterrupt`

Muchos sistemas proporcionan una función llamada `siginterrupt`, que permite controlar el comportamiento de las llamadas interrumpidas.

Escritura de datos en presencia de señales no ignoradas

Esta función escribe los datos que se le pasan aunque lleguen señales no ignoradas:

```
int escribe_datos(int sock, void *datos, size_t n)
{
   int ret;

   while ((ret = send(sock, datos, n, 0)) < 0 && errno == EINTR);
   return ret;
}
```

2.1.4. Recibir datos

Para recibir datos, podemos utilizar tres llamadas diferentes:

1) La llamada **`read`.** Como cualquier descriptor de fichero en general, los *sockets* no sólo admiten la llamada `write` para enviar datos, sino también la llamada `read` para recibirlos. En UNIX, el prototipo de esta llamada está declarado en el fichero cabecera `<unistd.h>` de la manera siguiente:

```
ssize_t read(int descr, void *datos, size_t longitud);
```

– El parámetro **`descr`** es el descriptor del *socket*.

– El parámetro **`datos`** es un puntero a la zona de memoria en que deben dejarse los datos leídos.

– El parámetro **`longitud`** es el número máximo de *bytes* que queremos leer*.

*** En la zona apuntada por `datos`, deben caber como mínimo `longitud` *bytes*.**

El valor retornado informa del número de *bytes* leídos, que puede ser menor que el tercer parámetro si, de momento, no hay más disponibles. Si el valor retornado es igual a 0, indica que ya no quedan más *bytes* para leer; es decir, si se trata de un fichero, indica que se ha llegado al final de su contenido y, si se trata de un *socket*, que el otro extremo ha cerrado la conexión. Si el valor devuelto es –1, se ha producido algún error en la lectura.

Bucle de lecturas

Si esperamos que el *socket* remoto nos envíe *n* bytes, puede ser que una sola llamadas `read` con el valor *n* como tercer parámetro no sea suficiente para recibirlos todos, puesto que la llamada sólo espera que haya bytes disponibles, pero no necesariamente que haya tantos como se le piden. Ello significa que, para leer exactamente *n* bytes, en general es preciso llevar a cabo un bucle de lecturas, como el presentado en la función siguiente:

```
int leer_n_bytes(int sock, char *buffer, int n)
{
   int leidos, posicion = 0;

   while (posicion < n)
   {
      if ((leidos = read(sock, buffer + posicion,
           n - posicion)) < 0) return -1;
      if (leidos == 0) break;
      posicion += leidos;
   }
   return posicion;
}
```

El número de llamadas `read` necesarias para leer *n* bytes no tiene por qué guardar relación con el número de llamadas `write` que se han efectuado en el *socket* remoto para enviarlas.

2) La llamada **`recv`**. Es la llamada específica de los *sockets* para leer datos, análoga a la llamada `send` del caso de la escritura. Su prototipo está declarado en el fichero cabecera `<sys/socket.h>` de la manera siguiente:

```
int recv(int descr, void *datos, size_t longitud,
         int flags);
```

De una manera igual a la de las llamadas `write` y `send`, la diferencia entre `read` y `recv` es simplemente el parámetro adicional `flags`*. Este último se codifica, de la misma manera que en la llamada `send`, como una cadena de bits, y las opciones que se pueden especificar en el caso de la lectura están determinadas por las constantes siguientes:

*** Si el parámetro `flags` vale 0, las llamadass `read` y `recv` son iguales.**

- `MSG_PEEK`: hace que la llamada lea los datos sin borrarlos del *buffer* de recepción del *socket* (por tanto, la llamadas siguiente volverá a leer los mismos datos).

- `MSG_OOB`: hace que la llamada lea datos urgentes enviados fuera de banda, aunque haya datos normales (no urgentes) en el *buffer* de recepción pendientes de lectura. Sin la opción MSG_OOB, la llamada sólo lee datos normales.

MSG_OOB

Esta opción sólo es aplicable al estilo de comunicación secuencia de bytes.

En UNIX, cuando llegan datos urgentes a un *socket*, el proceso correspondiente recibe de inmediato una señal `SIGURG`. Por defecto, los procesos la ignoran; sin embargo, se puede "capturar" para que se ejecute automáticamente una función definida por el usuario que lea los datos urgentes. De este modo, se puede conferir prioridad a los datos urgentes por encima de los datos normales.

Los datos urgentes, pues, se pueden leer fuera de secuencia, pero también es posible saber en qué punto de la secuencia de datos normales se transmitieron, puesto que en este punto se inserta automáticamente una "marca".

Ello es útil cuando los dastos urgentes contienen, por ejemplo, un mensaje de control que significa interrumpir el proceso actual y descartar los datos recibidos hasta aquel momento.

Si una llamada de lectura de datos normales encuentra una marca de datos urgentes, deja de leer y retorna con los datos que hay antes de la marca. Es decir, los datos enviados antes y después de un mensaje urgente no se leerán nunca en una misma llamada.

Detección de la marca

En UNIX, existe una operación de control llamada `SIOCATMARK`, que permite comprobar si la última llamada de lectura ha retornado porque ha llegado a una marca. A continuación, presentamos un ejemplo de uso de esta operación. El prototipo de la llamada `ioctl` está declarado en `<unistd.h>` y el símbolo `SIOCATMARK` se define en `<sys/ioctl.h>`:

```
int marca_encontrada(int sock)
{
   int resultado;

   if (ioctl(sock, SIOCATMARK, &resultado) < 0)
   {
   perror("ioctl(SIOCATMARK)");
   return -1;
   }
   return resultado;
}
```

El resultado retornado será 1 si la última lectura ha encontrado una marca y 0 en caso contrario.

3) La llamada **`recvfrom`**. Lee datos e indica de dónde han venido (opera de la misma manera que la llamada `sendto`). Su prototipo está declarado en el fichero cabecera `<sys/socket.h>` de la manera siguiente:

```
int recvfrom(int descr, void *datos, size_t longitud,
   int flags, struct sockaddr *adr, size_t *long_adr);
```

La diferencia entre `recv` y `recvfrom` es que la segunda llena la estructura apuntada por el parámetro **`adr`** con la dirección del *socket* del que se han recibido los datos (salvo que este parámetro constituya un puntero nulo).

El parámetro **`long_adr`** en un inicio debe apuntar a un valor que indica cuántos bytes puede ocupar como máximo la estructura y, cuando retorna, la llamada cambia este valor al número de bytes que ha llenado. En muchos sistemas, sin embargo, la llamada `recvfrom` no proporciona información sobre la dirección remota en el espacio de nombres de ficheros.

En los *sockets* orientados a conexión (estilo de comunicación secuencia de *bytes*), las llamadas `read`, `recv` y `recvfrom` leen los datos a partir del último *byte* leído por la llamada anterior. En los no orientados a conexión, en cambio, leen sólo un datagrama cada vez y, si la longitud del datagrama recibido es mayor que el parámetro `longitud`, los bytes que sobran se descartan y no hay manera de recuperarlos.

Cada uno de estas tres llamadas puede retornar un número de bytes leídos menor que el solicitado en el parámetro `longitud` si no hay más disponibles en el *buffer* de recepción. Sin embargo, si este último está vacío, las llamadas esperan hasta que haya datos para leer, salvo que el *socket* trabaje en modo no bloqueante.

Cuando estas llamadas retornan con error, los valores habituales de la variable `errno` son los siguientes:

- `EBADF`: el primer parámetro no es un descriptor válido.
- `ENOTSOCK` (sólo para las llamadas `recv` y `recvfrom`): el primer parámetro es un descriptor, pero no corresponde a un *socket*.
- `ENOTCONN`: el *socket* es orientado a conexión, pero no está conectado.
- `EWOULDBLOCK`: el *socket* es no bloqueante y no existen datos disponibles para leerlos (la memoria intermedia está vacía).
- `EINTR`: no se ha completado la operación porque ha llegado una señal no ignorada.

2.1.5. Esperar disponibilidad de datos

La llamada **`select`** en UNIX permite esperar hasta que se puedan leer o escribir datos en algún descriptor de una lista determinada. Esta llamada también es aplicable a los *sockets* y es muy útil cuando, por ejemplo, un proceso debe

recibir datos por diferentes canales, que pueden ser dos o más *sockets*, o un *socket* y la entrada estándar, etc. Con la llamada `select` se puede saber por cuál de estos canales se han recibido datos, y de este modo se evita que el proceso quede bloqueado esperando datos en un descriptor mientras por otro, quizá, van llegando otros sin que se puedan leer.

El prototipo de la llamada `select`, que se detalla a continuación, tradicionalmente estaba declarado, como se indica a continuación, en el fichero cabecera `<sys/types.h>`; sin embargo, el estándar POSIX requiere que aparezca en el fichero `<sys/time.h>`:

```
int select(int n_descr, fd_set *d_lect, fd_set *d_escr,
           fd_set *d_excep, struct timeval *tiempo);
```

Para representar los parámetros, se utilizan los tipos `fd_set` y `struct timeval`. El primero se define en el fichero cabecera `<sys/types.h>` y/o `<sys/time.h>`, según el sistema, y el segundo, en el fichero `<sys/time.h>`.

Veamos estos tipos con detalle:

a) El tipo **`fd_set`** sirve para representar conjuntos de descriptores. Por norma general, se codifica como una máscara de bits en la que los bits puestos a 1 indican qué descriptores pertenecen al conjunto. Sin embargo, el programador no debe preocuparse de la representación interna de estos conjuntos, puesto que los mismos ficheros `<sys/types.h>` y/o `<sys/time.h>` definen las operaciones siguientes para trabajar con variables del tipo `fd_set`:

- `void FD_ZERO(fd_set *conj_d);`. La operación `FD_ZERO` pone a 0 todos los descriptores del conjunto `conj_d`.
- `void FD_SET(int descr, fd_set *conj_d);`. La operación `FD_SET` pone a 1 el descriptor correspondiente al argumento `descr` del conjunto `conj_d`.
- `void FD_CLR(int descr, fd_set *conj_d);`. La operación `FD_CLR` pone a 0 el descriptor correspondiente al argumento `descr` del conjunto `conj_d`.
- `int FD_ISSET(int descr, fd_set *conj_d);`. La operación `FD_ISSET` retorna un valor diferente de 0 si el descriptor correspondiente al argumento `descr` del conjunto `conj_d` está puesto a 1, o retorna 0 si está puesto a 0.

Por norma general, los símbolos `FD_ZERO`, `FD_SET`, `FD_CLR` y `FD_ISSET` se definen como macros.

El fichero `<sys/types.h>` y/o `<sys/time.h>` también define la constante `FD_SETSIZE`, que equivale al número máximo de descriptores que caben en una variable del tipo `fd_set`.

b) El tipo **`struct timeval`** sirve para representar un intervalo de tiempo y se define de la manera siguiente:

```
struct timeval
{
   long int tv_sec;
   long int tv_usec;
};
```

El campo `tv_sec` expresa el número de segundos del intervalo y el campo `tv_usec`, el número de microsegundos.

Una vez explicados los tipos de los parámetros, volvemos al funcionamiento de la llamada `select`.

La llamadas `select` retorna cuando se da alguna de las cuatro condiciones siguientes:

– Se puede llevar a cabo una lectura sin bloqueo. Por ejemplo, cuando existen datos disponibles para leer en alguno de los descriptores de la lista especificada en el segundo parámetro, **`d_lect`**. Asimismo, en el caso de que se haya llegado al final de los datos (en un *socket* conectado, cuando se haya cerrado la conexión).

– Se pueden escribir datos inmediatamente (sin bloqueo) en alguno de los descriptores de la lista especificada en el tercer parámetro, **`d_escr`**.

– Hay una situación especial (una condición excepcional) pendiente de procesar en alguno de los descriptores de la lista especificada en el cuarto parámetro, **`d_excep`**. En el caso de los *sockets*, se considera como una condición excepcional la llegada de datos urgentes.

– Ha transcurrido el tiempo especificado en el quinto parámetro, **`tiempo`**.

El primer parámetro, **`n_descr`**, indica el número máximo de descriptores existentes en cada una de las tres listas. Es decir, sólo se consideran significativos los `n_descr` primeros bits de cada lista (los correspondientes a los descriptores entre 0 y `n_descr - 1`).

La constante `FD_SETSIZE`

Cuando hay más de un *socket* en las listas, y cualquiera puede ser el mayor, es habitual utilizar la constante `FD_SETSIZE` como valor del primer parámetro.

Cuando retorna, la llamada modifica las listas de descriptores y deja puestos a 1 sólo los bits de los descriptores en que se cumplen las condiciones respectivas (posibilidad de lectura o escritura, o presencia de excepciones).

Fuera de tiempo

En algunos sistemas, la llamada `select` también modifica el contenido del quinto parámetro para indicar cuánto tiempo falta para llegar al fuera de tiempo. Por tanto, un programa que deba ser portable no puede presuponer que el contenido del quinto parámetro continuará siendo el mismo después de devolver la llamada.

El valor retornado es el número total de bits que han quedado puestos a 1 en todas las listas, si no se ha producido ningún error, caso en el que el valor retornado es –1. Si es 0, significa que la llamada ha retornado por tiempo agotado (*time out*); es decir que ha transcurrido el tiempo indicado por el último parámetro sin que se diera ninguna de las otras condiciones.

Si alguno de los parámetros `d_lect`, `d_escr`, `d_excep` o `tiempo` es un puntero nulo, la llamada `select` no comprueba la condición correspondiente. Por ejemplo, si `tiempo` es nulo, significa que no hay `tiempo` máximo y, por consiguiente, la llamada espera indefinidamente hasta que se dé alguna de las otras condiciones.

Asimismo, es posible hacer que la llamada `select` retorne de inmediato, si se le pasa en el parámetro `tiempo`, un puntero a una estructura que tenga sus campos a 0. Ello es útil para efectuar una operación de encuesta sobre los descriptores, puesto que, en este caso, la llamada simplemente retorna indicando en cuáles se dan en aquel momento las condiciones especificadas, sin esperar nada.

Cuando la llamada `select` retorna con error, la variable `errno` puede tener alguno de los valores siguientes:

- `EBADF`: algún bit de las listas del segundo parámetro, el tercero y/o el cuarto no corresponde a un descriptor válido.

- `EINVAL`: el contenido del quinto parámetro es incorrecto (es negativo o demasiado grande).

- `EINTR`: no se ha esperado el tiempo indicado por el quinto parámetro porque ha llegado una señal no ignorada.

Reemisión de datos

La función siguiente tiene como parámetros tres descriptores: todos los datos recibidos por el primero, los reenvía al segundo y, simultáneamente, los recibidos por el segundo, los reenvía al tercero. La función retorna cuando se detecta el final de los datos en el primer descriptor o en el segundo.

```
int recibir_enviar(int d1, int d2, int d3)
{
   char buffer[DIM_BUFFER];
   fd_set lista, preparados;
   int n;

   FD_ZERO(&lista);
   FD_SET(d1, &lista);
   FD_SET(d2, &lista);
   while (1)
   {
      preparados = lista;
      if (select(FD_SETSIZE, &preparados, NULL, NULL, NULL) <
0)
               return -1;
      if (FD_ISSET(d1, &preparados))
      {
         if ((n = read(d1, buffer, sizeof buffer)) < 0)
              return -1;
         if (n == 0) break;
         if (write(d2, buffer, n) < 0) return -1;
      }
      if (FD_ISSET(d2, &preparados))
      {
         if ((n = read(d2, buffer, sizeof buffer)) < 0)
              return -1;
```

```
            if (n == 0) break;
            if (write(d3, buffer, n) < 0) return -1;
         }
      }
      return 0;
   }
```

El uso de la llamada `select` permite saber dónde se encuentran datos disponibles. Así, se evita tener que leer en un descriptor que no tiene datos y, por tanto, que el proceso se bloquee si mientras tanto llegan datos por el otro (hemos supuesto, por simplicidad, que las operaciones de escritura siempre se pueden llevar a cabo inmediatamente).

Como caso particular, se puede llamar esta función y pasarle como primer parámetro el valor 0 (el descriptor que por norma general corresponde a la entrada estándar), como segundo el descriptor de un *socket* y, como tercero, el valor 1 (la salida estándar). Con ello, se obtiene un procedimiento que envía por el *socket* los datos que lee del dispositivo de entrada (si no está redireccionado, será el teclado) y que, simultáneamente, escribe por el dispositivo de salida (si no está redireccionado, será la pantalla) los datos que recibe por este mismo *socket*, hasta que encuentra el final de la entrada o de la conexión.

Consultad un ejemplo del uso de la función `recibir_enviar` en el subapartado 2.3.2. de este módulo didáctico.

2.1.6. Cerrar un *socket*

La llamada **`close`** proporciona el mecanismo general para cerrar un descriptor cuando ya no se necesita trabajar más en el mismo y liberar los recursos que el sistema tiene destinados (*buffers* de lectura y escritura, etc.). Una vez cerrado, el argumento de esta llamada deja de ser un descriptor válido.

El prototipo de la llamada `close`, que está declarado en el fichero cabecera `<unistd.h>`, es el siguiente:

```
int close(int descr);
```

El parámetro **`descr`** indica el descriptor que se quiere cerrar, y el valor retornado será 0 para informar de que la operación se ha llevado a cabo correctamente, o –1 para indicar que se ha producido algún error.

Cuando se aplica la llamada `close` a un *socket*, se dan por acabadas las comunicaciones para este último. Si el *socket* estaba conectado, se siguen los pasos establecidos en el protocolo correspondiente para cerrar la conexión. En particular, si se cierra un *socket* que utiliza un protocolo de transmisión fiable, y todavía quedan datos pendientes de enviar, por norma general la llamada no retorna hasta que no se han acabado de enviar todos los datos.

Cierre de un *socket* TCP

Cuando se cierra un *socket* que utiliza el protocolo TCP, puede suceder que la llamada retorne, pero que el número de puerto que tenía asignado el *socket* continúe estando ocupado durante cierto tiempo. Ello significa que la conexión que correspondía a este *socket* todavía está en el estado `time_wait` del protocolo.

Si se cierra un *socket* del espacio de nombres de ficheros, la entrada de directorio correspondiente a este *socket* permanece en el sistema de ficheros. Para eliminarla, es preciso borrarla, por ejemplo, con la llamada `unlink`.

Por otro lado, hay una llamada específica de los *sockets* denominada **`shutdown`**, que permite dar por finalizada la comunicación en cualquiera de los dos cana-

les del intercambio de datos bidireccional. Su prototipo está declarado en el fichero cabecera <sys/socket.h> de la manera siguiente:

```
int shutdown(int descr, int modo);
```

- El parámetro **descr** corresponde al descriptor del *socket*.
- El parámetro **modo** indica cómo debe aplicarse la operación al *socket*:
  - Si vale 0, se da por finalizada la recepción de datos por el *socket* (si llegan más datos, no se leerán).
  - Si vale 1, se da por culminado el envío de datos por el *socket* (si hay datos pendientes de enviar, se descartan).
  - Si vale 2, se dan por acabados tanto la recepción, como el envío de datos.

Los valores de la variable errno siguientes indican errores que se pueden producir en las llamadas close y shutdown:

- EBADF: el primer parámetro no es un descriptor válido.
- ENOTSOCK (sólo para la llamada shutdown): el primer parámetro es un descriptor, pero no corresponde a un *socket*.
- ENOTCONN (sólo para la llamada shutdown): el *socket* es orientado a conexión, pero no está conectado.
- EINTR (sólo para la llamada close): no se ha completado la operación (no se han acabado de transmitir los datos que quedaban pendientes) puesto que ha llegado una señal no ignorada.

2.2. Operaciones propias de los servidores

2.2.1. Preparar un *socket* para recibir conexiones

En los protocolos orientados a conexión, el *socket* servidor debe estar preparado para recibir conexiones de manera pasiva. La manera de conseguirlo es por medio de la llamada **listen**, cuyo prototipo está declarado en el fichero cabecera <sys/socket.h> de la manera siguiente:

```
int listen(int descr, unsigned int n_petic);
```

La llamada listen crea una cola en la que se irán guardando las peticiones de conexión que lleguen destinadas a la dirección del *socket* indicado por el primer parámetro, **descr**. El parámetro **n_petic** determina la longitud de la cola.

En el espacio de nombres Internet,...

... si el *socket* pasado a la llamada listen no tuviera dirección, el sistema le asignaría una (aleatoria) de manera automática y, por tanto, los clientes no podrían saber *a priori* a dónde deberían conectarse.

Como es natural, los clientes deben saber la dirección del *socket* servidor para poderse conectar al mismo. Por tanto, es necesario haberle asignado una con la llamada bind antes de aplicarle la llamada listen.

El valor retornado por esta llamada es 0 si no se produce error, o –1 en caso contrario. Los posibles valores de la variable `errno` son los que mencionamos a continuación:

- `EBADF`: el primer parámetro no es un descriptor válido.
- `ENOTSOCK`: el primer parámetro es un descriptor, pero no corresponde a un *socket*.
- `EOPNOTSUPP`: el *socket* especificado no soporta la operación (no es un *socket* orientado a conexión).
- `EINVAL`: el primer parámetro es incorrecto porque corresponde a un *socket* al que no se le puede aplicar la operación (por ejemplo, porque ya está conectado a otro *socket*).

2.2.2. Aceptar una petición de conexión

Un *socket* servidor al que se le haya aplicado la llamada `listen` debe utilizar la llamada **`accept`** para establecer las conexiones con los clientes que las soliciten. El prototipo de dicha llamada está declarado en el fichero cabecera `<sys/socket.h>` de la manera siguiente:

```
int accept(int descr, struct sockaddr *adr,
                  size_t *long_adr);
```

La llamada `accept` extrae la primera petición de conexión de la cola asociada al *socket* indicado por el primer parámetro, **`descr`**, y crea un nuevo *socket*, que tendrá la misma dirección que el original y estará conectado al originador de la petición. El *socket* original continuará preparado para recibir nuevas peticiones de conexión.

El valor retornado es el descriptor del nuevo *socket* creado, que deberá utilizarse en el intercambio de datos con el cliente. Asimismo, la llamada llena la estructura apuntada por el segundo parámetro, **`adr`**, con la dirección del *socket* cliente con que se ha establecido la conexión y el tercer parámetro, **`long_adr`** (que en un inicio debe contener la longitud máxima admisible), con la longitud total de esta dirección. Si se produce algún error, el valor retornado es –1.

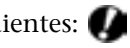
Si en el momento de invocar la llamada `accept` no hay ninguna petición de conexión a la cola, dicha llamada actúa de la misma manera que `read`: se bloquea y se queda esperando hasta que llegue una, salvo que el *socket* trabaje en modo no bloqueante.

Asimismo, es posible esperar o comprobar si hay peticiones por medio de la llamada `select`. Para ello, simplemente es preciso incluir el descriptor del *socket*

Consultad las llamadas `read` y `select` en los subapartados 2.4.1 y 2.1.5 de este módulo didáctico.

en la lista de descriptores del segundo parámetro, `d_lect`, puesto que, en el caso de un *socket* servidor al que se le ha aplicado la llamada `listen`, la condición de disponibilidad de datos de lectura se interpreta como una "presencia de peticiones de conexión en la cola".

Cuando la llamada `accept` retorna con error, la variable `errno` puede tener uno de los valores siguientes:



- `EBADF`: el primer parámetro no es un descriptor válido.
- `ENOTSOCK`: el primer parámetro es un descriptor, pero no corresponde a un *socket*.
- `EOPNOTSUPP`: el *socket* especificado no soporta la operación (no es un *socket* orientado a conexión).
- `EINVAL`: el primer parámetro es incorrecto porque corresponde a un *socket* al que no se le puede aplicar la operación (no se ha invocado la llamada `listen` y, por consiguiente, no está preparado para recibir peticiones de conexión).
- `EWOULDBLOCK`: el *socket* es no bloqueante y no hay conexiones pendientes de aceptar en la cola.
- `EINTR`: no se ha completado la operación porque ha llegado una señal no ignorada.

2.2.3. Ejemplos de programación de un servidor

Los ejemplos que vienen a continuación ilustran el uso de las llamadas de *sockets* para programar un servidor basado en el protocolo TCP. Utilizando dos técnicas de programación diferentes, los fragmentos de código siguientes muestran cómo se puede implementar un servidor que acepte conexiones de un número cualquiera de clientes. Para cada conexión, el servidor lee los datos que le envía el cliente, genera una respuesta procesando los datos recibidos y se la envía al cliente.

Ejemplo de gestión de conexiones con un solo proceso

En este ejemplo, el programa deja el *socket* servidor preparado para recibir peticiones de conexión y, a continuación, entra en un bucle en el que comprueba, por medio de la llamada `select`, si llegan nuevas peticiones y/o datos de las conexiones ya establecidas.

Cuando hay algún *socket* preparado, lo atiende: si es una petición de conexión, la acepta y la añade a la lista de *sockets* activos para las próximas llamadas `select` y, si son datos que han llegado por un *socket* ya conectado, los procesa y les envía la respuesta.

Cuando detecta el final de una conexión, cierra el *socket* correspondiente y lo borra de la lista.

El parámetro de la función servidor de este ejemplo es el número de puerto en que deben recibirse las peticiones de conexión. Para asignar la dirección al *socket* servidor, utiliza la función `asignar_dir_internet` y pone en la parte de dirección del servidor la constante `INADDR_ANY`. Asimismo, utiliza la función `crear_socket_TCP`.

Consultad el ejemplo de creación de un *socket* TCP en el subapartado 2.1.1 y la asignación de un nombre a un *socket* del espacio Internet en el subapartado 2.1.2 de este módulo.

Para generar las respuestas, supondremos que hay una función `procesa_datos` que recibe dos parámetros que indican dónde se encuentran los datos de entrada y cuántos bytes ocupan. Esta función retorna un puntero a los datos procesados (salvo que todavía no haya suficientes datos de entrada para generar la respuesta) y, en un parámetro por referencia (el tercero), retorna su longitud.

```
char *procesa_datos(char *, int, int *);

int servidor(unsigned short int port)
{
   char buffer[DIM_BUFFER], *buf_proc;
   struct sockaddr_in adr;
   fd_set activos, preparados;
   int sock, conn, i, n, n_proc;
   unsigned int long_adr;
   if ((sock = crear_socket_TCP())< 0) return -1;
   if (asignar_dir_internet(sock, INADDR_ANY,
       port) < 0) return -1;
   if (listen(sock, 1) < 0) return -1;
   FD_ZERO(&activos);
   FD_SET(sock, &activos);
   while (1)
   {
      preparados = activos;
      if (select(FD_SETSIZE, &preparados, NULL, NULL,
          NULL) < 0) return -1;
      for (i = 0; i < FD_SETSIZE; i++)
           if (FD_ISSET(i, &preparados))
      {
           if (i == sock)
           {
               long_adr = sizeof adr;
               if ((conn = accept(sock, (struct sockaddr *)
                    &adr, &long_adr))< 0) return -1;
               FD_SET(conn, &activos);
               continue;
```

```
          }
          if ((n = read(i, buffer, sizeof buffer)) < 0)
               return -1;
          if (n == 0)
          {
               close(i);
               FD_CLR(i, &activos);
               continue;
          }
          if ((buf_proc = procesa_datos(buffer, n,
               &n_proc)) ! = NULL)
               if (write(i, buf_proc, n_proc) < 0)
                   return -1;
     }
   }
}
```

La función de este ejemplo retorna cuando se produce cualquier error en las llamadas al sistema (quien haya llamado esta función siempre puede consultar el valor de la variable `errno`).

Ejemplo de bucle de creación de procesos

El ejemplo anterior gestiona todas las conexiones con un solo proceso; sin embargo, en UNIX, en numerosas ocasiones es más conveniente crear un nuevo proceso cada vez que llega una petición de servicio y dejar que cada uno de dichos procesos se encargue de atender a su cliente.

En el ejemplo siguiente, el bucle del servidor espera conexiones y, cuando llega una, crea un proceso hijo con la llamada `fork`. El proceso padre vuelve a esperar nuevas conexiones, mientras que el hijo se encarga de intercambiar los datos con el cliente por medio de la función auxiliar `bucle_conexion` (sus parámetros son el descriptor de entrada y el de salida).

```
char *procesa_datos(char *, int, int *);

int bucle_conexion(int entrada, int salida)
{
   char buffer[DIM_BUFFER], *buf_proc;
   int n, n_proc;

   while ((n = read(entrada, buffer,
           sizeof buffer)) ! = 0)
   {
     if (n < 0) return -1;
```

```
      if ((buf_proc = procesa_datos(buffer, n,
           &n_proc)) ! = NULL)
           if (write(salida, buf_proc, n_proc) < 0)
               return -1;

   }
   return 0;
}

int servidor(unsigned short int port)
{

   struct sockaddr_in adr;
   int sock, conn, proceso;
   unsigned int long_adr;

   if ((sock = crear_socket_TCP()) < 0) return -1;

   if (asignar_dir_internet(sock, INADDR_ANY,
       port) < 0) return -1;
   if (listen(sock, 1) < 0) return -1;
   while (1)
   {
      long_adr = sizeof adr;
      if ((conn = accept(sock, (struct sockaddr *)&adr,
           &long_adr)) < 0) return -1;
      if ((proceso = fork()) < 0) return -1;
      if (proceso == 0)
      {
         if (bucle_conexion(conn, conn) < 0) exit(1);
         close(conn);
         exit(0);
      }
   }
}
```

En este ejemplo, para simplificar, el proceso padre no invoca la llamada `wait` para consultar el estatus de salida de los hijos cuando acaban, puesto que dicha llamada, como su nombre indica, es bloqueante. Por tanto, aunque ya no se ejecutan, los hijos se quedan ocupando una entrada en la tabla de procesos; es decir, pasan a ser lo que se conoce como **procesos zombis**. Lo que se puede evitar si se captura la señal `SIGCHLD` (que se ignora por defecto) y se lleva a cabo la llamada `wait` en el gestor de esta señal. Si se opta por esta solución, conviene considerar que la llamada `accept` puede retornar prematuramente con `EINTR` si un hijo acaba mientras se esperan conexiones y, por tanto, es preciso realizar una comprobación como la que debía hacerse cuando llegaban señales no ignoradas mientras se escribían datos, o bien utilizar la función `siginterrupt`.

Consultad el ejemplo "Escritura de datos en presencia de señales no ignoradas" en el subapartado 2.1.3.

2.2.4. El servidor `inetd`

Con frecuencia, los sistemas UNIX ofrecen una serie de servicios Internet, como por ejemplo, Telnet, FTP, correo electrónico, posiblemente WWW, etc. Una ma-

nera de poder atender a las peticiones correspondientes es tener corriendo tantos procesos como servicios y que cada uno se encargue de esperar las peticiones que lleguen al puerto asociado a su servicio. Sin embargo, para aprovechar mejor los recursos, lo que es más habitual es tener un único proceso, llamado **`inetd`** o *Internet daemon*, que se encarga de escuchar simultáneamente todos los puertos por los que pueden llegar peticiones y que, cada vez que llega una, arranca un proceso hijo que proporciona el servicio correspondiente.

No todos los servicios son atendidos por `inetd`

El proceso `inetd` no necesariamente debe atender a todas las posibles peticiones de servicio que lleguen a un sistema UNIX. Aparte de los servicios no estándar que puedan ser proporcionados por procesos de usuario, `inetd` por norma general no atiende al servicio de correo electrónico. El motivo es que el proceso encargado de este servicio (`sendmail`) no sólo atiende a peticiones de clientes, sino que también lleva a cabo otras acciones (procesar periódicamente la cola de mensajes, etc.). Asimismo, su fichero de configuración (`/etc/sendmail.cf`) puede tener cierta complejidad, y es más conveniente leerlo una vez al inicio que tenerlo que leer y analizar a cada petición de servicio. La complejidad del fichero de configuración también es el motivo por el cual algunas implementaciones de servidores WWW tampoco utilizan el proceso `inetd`.

Al proceso `inetd` se le indican los puertos que debe escuchar por medio de un fichero de configuración denominado `inetd.conf`, que suele estar en el directorio /etc. Cada línea del fichero consta de una serie de campos separados por espacios, de acuerdo con el esquema siguiente:

```
servicio estilo protocolo espera usuario programa argumentos
```

Ejemplo

Dos líneas que podríamos encontrar en el fichero `/etc/inetd.conf`:

- ftp stream tcp nowait root /usr/etc/ftpd ftpd -l
- talk dgram udp wait root /usr/etc/talkd talkd

El significado de los campos que forman cada línea de fichero es el siguiente:

- El primer campo de la línea es el **nombre del servicio**.
- El segundo campo indica el **estilo de comunicación**. Se expresa con los símbolos definidos en el fichero `<sys/socket.h>`, pero sin el prefijo `SOCK_` y en minúsculas.
- El tercer campo es el **protocolo** que debe utilizarse. Por norma general, será `tcp` si el estilo es `stream` y `udp` si el estilo es `dgram`.
- El cuarto campo indica si el servidor `inetd` debe **esperar** (`wait`) que el proceso hijo acabe, o no (`nowait`), antes de continuar atendiendo a nuevas peticiones del servicio correspondiente.

`wait` y `nowait`

`wait` sólo es aplicable a los servicios que utilizan el estilo datagrama; en todos los otros debe utilizarse `nowait`.

- En el quinto campo encontramos el usuario en cuyo nombre se ejecutará el proceso hijo.

- El sexto campo es el nombre del **programa** que debe ejecutar el proceso hijo. Alternativamente, este campo puede ser la palabra `internal` para indicar que el servicio es atendido directamente por el mismo servidor `inetd`, sin necesidad de llamar a ningún otro proceso. Es el caso de algunos servicios llamados **triviales**.

  Ejemplos de servicios internos del proceso `inetd`

  - `echo`: retorna los datos que recibe.
  - `discard`: no retorna nada.
  - `chargen`: genera una secuencia de caracteres.
  - `daytime`: envía una cadena de caracteres con el día y la hora.
  - `time`: envía 4 bytes que representan la hora actual en binario.

- El séptimo campo y los siguientes forman la lista de **argumentos** que se pasarán al proceso hijo. Por norma general, los procesos interpretan el primer argumento de la lista como el nombre con que se han invocado.

Privilegios de `inetd`

`inetd` debe tener privilegios de superusuario para escuchar en puertos reservados y cambiar el propietario de los procesos hijos.

En primer lugar, el servidor `inetd` crea un *socket* para cada servicio que encuentra en el fichero de configuración por medio del estilo indicado (segundo campo). A cada *socket* le asigna, con la llamada `bind`, un número de puerto obtenido a partir del nombre del servicio (primer campo) y del protocolo asociado (tercer campo) utilizando, por ejemplo, la función `getservbyname`, como veremos más adelante. A los *sockets* que sean orientados a conexión, se les aplica la llamada `listen`. Después se queda en un bucle esperando, por medio de la llamada `select`, a que lleguen peticiones de conexión a los *sockets* con estilo `stream`, o datagramas a los *sockets* con estilo `dgram`.

Modificación del fichero `inetd.conf`

El servidor `inetd` sólo consulta el fichero de configuración cuando se empieza a ejecutar (por norma general, cuando arranca el sistema operativo) y cuando recibe una señal `SIGHUP`. Por tanto, si se modifica el fichero `inetd.conf`, es preciso enviar esta señal al proceso `inetd` para que los cambios tengan efecto.

Cuando la llamada `select` notifique que se ha recibido una petición de un cliente, el servidor `inetd` aplicará la llamada `accept` al *socket* si es orientado a conexión. A continuación, creará un proceso hijo, hará que su propietario sea el usuario indicado (quinto campo), ejecutará el programa correspondiente (sexto campo) y le pasará los argumentos especificados (séptimo campo). Entonces ocurrirá lo siguiente:

- Si el cuarto campo es `nowait`, el servidor `inetd` continuará atendiendo a peticiones de servicio y poniendo en marcha nuevos procesos para cada una que llegue, con independencia de si los otros han acabado o no. Es decir, a cada cliente que solicite el servicio, le corresponderá un proceso hijo.

- Si el cuarto campo es `wait` (lo que implica que el servicio es no orientado a conexión), el servidor `inetd` se quedará esperando hasta que el proceso hijo acabe. De los nuevos datagramas que lleguen mientras tanto, tanto si son del mismo cliente, como si son de otro, deberá encargarse el proceso hijo.

Así pues, no es necesario que los procesos que deban ser ejecutados por el servidor `inetd` se preocupen por establecer la comunicación con el cliente (crear el *socket*, asignarle dirección, esperar a que llegue la petición y, en el caso de los servicios `nowait`, distinguir los diferentes clientes que solicitan la conexión), puesto que se encarga de los mismos el proceso `inetd`. Y no sólo eso, sino que, además, el proceso hijo recibe los descriptores correspondientes en la entrada y la salida estándar redireccionados al *socket* con que el cliente ha establecido la comunicación.

Ello significa que, cuando el proceso hijo escriba datos por la salida estándar, se los enviará al cliente por medio del *socket* y, cuando quiera leer datos de la entrada estándar, encontrará los que haya enviado el cliente. Por tanto, los programas servidores arrancados por el proceso `inetd` ni siquiera tienen por qué saber que se comunicarán mediante *sockets* y, por lo general, pueden utilizar directamente las llamadas `read`, `write`, etc. e, incluso, funciones de nivel más alto como `fgets`, `scanf`, `printf`, etc. para leer y escribir los datos.

Programación de un servidor invocado por el proceso `inetd`

Si el servidor encargado de crear un proceso para cada conexión debiera ser arrancado por el servidor `inetd`, como el descriptor 0 corresponde a la entrada estándar y el descriptor 1 a la salida estándar, la función servidor se podría escribir sencillamente de la manera siguiente:

```
int servidor(void)
{
   return bucle_conexion(0, 1);
}
```

En este caso, la función no necesitaría como parámetro el número de puerto en que debería escuchar las peticiones de conexión, puesto que esta información ya se habría proporcionado al servidor `inetd`.

Consultad el ejemplo de bucle de creación de procesos en el subapartado 2.2.3 de este módulo didáctico.

2.3. Operaciones propias de los clientes

2.3.1. Conectar un *socket*

La llamada **`connect`** sirve para que un cliente inicie de manera activa una conexión y su prototipo está declarado en el fichero cabecera `<sys/socket.h>` de la manera siguiente:

```
int connect(int descr, const struct sockaddr *adr,
                  size_t long_adr);
```

Si el *socket* indicado por el primer parámetro, **`descr`**, es orientado a conexión, la llamada sigue los pasos necesarios, según el protocolo correspondiente, para establecer una conexión con el *socket* servidor que tiene por dirección la que se especifica en el segundo parámetro, **`adr`**. El tercer parámetro, **`long_adr`**, indica el número de *bytes* que ocupa dicha dirección. La llamada no retorna hasta que el servidor no responde a la petición de conexión, excepto en los *sockets* no bloqueadores.

La llamada `connect` también puede aplicarse a *sockets* no orientados a conexión: simplemente les asocia una dirección de destino por defecto de manera que se les pueda enviar datos con la llamadas `sendto`, así como con `send` y `write`. Otro efecto de la llamada `connect` en este caso es que por el *socket* sólo se recibirán los datagramas que procedan de la dirección especificada.

La dirección remota asociada a un *socket* no orientado a conexión se puede cambiar en cualquier momento* volviendo a invocar la llamada `connect`. Asimismo, se puede cancelar esta asociación utilizando como parámetro una dirección que tenga en el campo `sa_family` el valor `AF_UNSPEC`.

*** En los *sockets* conectados no se puede llevar a cabo el cambio de dirección remota.**

No es necesario que el *socket* que se pasa a la llamada `connect` tenga asignada una dirección. En el espacio de nombres Internet, si el *socket* no tiene dirección, es decir, número de puerto, el sistema elige uno que no esté ocupado en el rango comprendido entre `IPPORT_RESERVED` e `IPPORT_USERRESERVED - 1` antes de intentar establecer la conexión.

Como es habitual, el valor retornado por la llamada será 0 o –1, dependiendo de si la operación se ha completado con éxito o no. Cuando hay error, los valores posibles de la variable `errno` son los siguientes:

- `EBADF`: el primer parámetro no es un descriptor válido.
- `ENOTSOCK`: el primer parámetro es un descriptor, pero no corresponde a un *socket*.
- `EINVAL`: el tercer parámetro es incorrecto porque no corresponde a una longitud de dirección válida en el espacio de nombres del *socket*.
- `EAFNOSUPPORT`: la dirección especificada no pertenece al espacio de nombres del *socket*.
- `EISCONN`: el *socket* es orientado a conexión y ya está conectado.
- `ENETUNREACH`: el *socket* es orientado a conexión y la dirección especificada es del espacio de nombres Internet; sin embargo, el sistema local no puede encontrar un camino para llegar a la red correspondiente al sistema remoto.
- `ETIMEDOUT`: el *socket* es orientado a conexión y ha transcurrido cierto tiempo máximo sin que se haya podido completar la operación de conexión (por ejemplo, porque el servidor no responda).
- `ECONNREFUSED`: el servidor ha rechazado la petición de conexión.

  Esta situación se da cuando en el servidor no hay ningún *socket* con la dirección especificada que esté esperando peticiones de conexión (es decir, al que se haya aplicado la llama-

da `listen`). Según el protocolo utilizado, también se puede dar cuando la cola de peticiones del *socket* servidor está llena, como sucede en el espacio de nombres de ficheros.

En el espacio Internet, algunos servidores rechazan las peticiones que llegan cuando la cola está llena y otros las ignoran. Con los primeros, la llamada `connect` en el cliente retorna con `ECONNREFUSED` y, con los segundos, continuará retransmitiendo las peticiones hasta que consiga conectarse o hasta que retorne con `ETIMEDOUT`.

- `EINPROGRESS`: el *socket* es no bloqueante y la conexión no se ha podido efectuar de manera inmediata.

  En este caso, la llamada retorna; sin embargo, el sistema continúa intentando la conexión. Para saber cuándo ha conseguido establecerla, se puede comprobar con la llamada `select` si el *socket* está preparado para escribir datos en la misma.

- `EALREADY`: el *socket* es no bloqueante e intenta establecer una conexión.

- `EINTR`: no se ha completado la operación porque ha llegado una señal no ignorada.

Además de estos posibles errores, en el espacio de nombres de ficheros también se pueden producir los errores propios del acceso a ficheros, como en la llamada `bind`: `ENOENT`, `ENOTDIR`, `EACCESS`, etc. Asimismo, hay un error específico de la llamada `connect`, `EPROTOTYPE`, que indica que el *socket* servidor existe en el sistema de ficheros; sin embargo, su tipo de protocolo (secuencial o datagramas) no coincide con el del *socket* cliente.

2.3.2. Ejemplo de programación de un cliente

El ejemplo siguiente ilustra cómo se pueden utilizar las llamadas de *sockets* para escribir un programa cliente.

Consultad el ejemplo "Remisión de datos" en el subapartado 2.1.5 de este módulo didáctico.

La función cliente definida a continuación recibe dos parámetros: la dirección Internet y el número de puerto correspondientes al *socket* al que debe conectarse, en que se supone que hay un proceso servidor esperando. Si consigue establecer la conexión en el mismo, utiliza la función `recibir_enviar` para enviar al servidor todo lo que lea de la entrada estándar y para escribir por la salida estándar todo lo que reciba del servidor.

También aquí se utilizan las funciones auxiliares que se utilizan para crear un *socket* y llenar una dirección en Internet. En este caso, no es necesario asignar dirección al *socket*, puesto que, en el momento de intentar la conexión, el sistema le asignará una automáticamente.

Consultad el ejemplo de creación de un *socket* en el subapartado 2.1.1 y el ejemplo de relleno de una dirección en el subapartado 2.1.2 de este módulo didáctico.

```
int cliente(unsigned long int adr_host,
           unsigned short int port)
{
   struct sockaddr_in adr;
   int sock;
```

```
    if ((sock = crear_socket_TCP()) < 0) return -1;
    llenar_dir_internet(&adr, adr_host, port);
    if (connect(sock, (struct sockaddr *)&adr,
        sizeof adr) < 0) return -1;
    if (recibir_enviar(0, sock, 1) < 0) return -1;
    close(sock);
    return 0;
}
```

2.4. Operaciones auxiliares

Además de las llamadas al sistema que hemos visto hasta ahora, los programas que trabajan con *sockets* pueden utilizar otras llamadas o funciones auxiliares proporcionadas por la librería correspondiente.

2.4.1. Obtener direcciones de *sockets*

Hay dos llamadas al sistema que permiten saber la dirección de un *socket* local y la del *socket* remoto al que está conectado. Los prototipos de estas llamadas se declaran en el fichero cabecera `<sys/socket.h>` de la manera siguiente:

1) La llamada **`getsockname`** permite conocer la dirección del *socket* local:

```
int getsockname(int descr, struct sockaddr *adr,
                  size_t *long_adr);
```

- El parámetro **`adr`** contiene la dirección del *socket* correspondiente al parámetro **`descr`**.
- El parámetro **`long_adr`** en un inicio debe contener el número máximo de *bytes* de la estructura que deben llenarse y, cuando la función retorne, contendrá el número real de bytes que se han llenado (como en las llamadas `recvfrom` y `accept`).

El valor retornado es 0 o –1 para indicar si se ha podido llevar a cabo la operación o no. En este último caso, la causa de error se notifica con la variable `errno`, que puede tener valores como `EBADF` o `ENOTSOCK`.

Algunos sistemas no soportan la obtención de direcciones en el espacio de nombres de ficheros. En este caso, la llamada retorna un nombre de longitud 0.

2) La llamada **`getpeername`** permite obtener la dirección del *socket* remoto:

```
int getpeername(int descr, struct sockaddr *adr,
                  size_t *long_adr);
```

Esta llamada opera como la anterior; sin embargo, en lugar de retornar la dirección del *socket* indicado por el parámetro `descr`, retorna la del *socket* al que está conectado. En esta llamada hay una causa de error adicional: `ENOTCONN`.

2.4.2. Convertir direcciones Internet

Estas funciones permiten trabajar con la representación de las direcciones IP en la notación textual de los *bytes* en decimal separados por puntos (por ejemplo, 193.146.196.5). Existen dos prototipos encargados de realizar este cambio y se declaran en el fichero cabecera `<arpa/inet.h>`:

El fichero `<arpa/inet.h>`

En algunos sistemas, es necesario haber incluido el fichero `<netinet/in.h>` antes de incluir el fichero `<arpa/inet.h>`.

1) La llamada **`inet_aton`** está definida de la manera siguiente:

```
int inet_aton(const char *text, struct in_addr *adr);
```

Esta llamada convierte la dirección textual contenida en el primer parámetro en una `struct in_addr`, con los *bytes* en el orden de la red, y se la asigna al segundo parámetro. Si el primer parámetro no contiene una dirección válida, retorna 0; de otro modo, retorna un valor diferente de 0.

La función `inet_addr`

Esta función (obsoleta) recibe como único parámetro la representación textual de una dirección y retorna directamente un **`unsigned long int`** correspondiente a esta dirección, o bien la constante **`INADDR_NONE`** (definida en **`<netinet/in.h>`** con el valor –1) si no es válida.

2) La llamada **`inet_ntoa`** está definida de la manera siguiente:

```
char *inet_ntoa(struct in_addr adr);
```

Esta llamada genera una cadena de caracteres, que representa en notación textual la dirección correspondiente al parámetro, y la retorna (las llamadas posteriores utilizarán el mismo espacio de memoria para dejar el resultado). Se supone que los bytes del parámetro están ordenados en el orden de la red.

2.4.3. Consultar bases de datos de nombres

La familia de protocolos Internet trabaja con diferentes tipos de nombres que, a la hora de intercambiar paquetes por medio de la red, se representan mediante nombres. Los dos casos más habituales son los nombres de los servidores, que se traducen en direcciones IP*, y los nombres de los servicios, que se traducen en números de puerto**.

*** Por ejemplo, de `www.uoc.edu` se pasa a 213.73.40.217.**

**** Por ejemplo, de `www` se pasa a 80.**

La traducción de una representación a la otra se lleva a cabo consultando una base de datos que contiene una lista de los nombres y los números asociados.

El DNS se explica en el módulo "Aplicaciones Internet".

Según el sistema, los datos se pueden obtener de un fichero de texto en el que cada línea corresponde a una entrada (como, por ejemplo, los ficheros `/etc/hosts` o `/etc/services` en UNIX) y/o por medio de un servicio de información distribuida local (como NIS) o global (como DNS, el servicio de nombres Internet).

El servicio NIS

El servicio NIS (*Network Information Service*, antes conocido como *Yellow Pages*) permite al administrador de un conjunto de ordenadores mantener ciertas bases de datos, como la de usuarios o la de direcciones IP, de una manera centralizada. Cada vez que deba modificarse algún registro, en lugar de llevar a cabo las actualizaciones en cada uno de los ordenadores, sólo es preciso realizarlas en el servidor NIS y, automáticamente, se propagarán a los otros ordenadores.

El grupo de funciones que veremos a continuación permite consultar las bases de datos de nombres con una interfaz homogénea que esconda los detalles de acceso. Sus prototipos se declaran en el fichero cabecera `<netdb.h>` de la manera siguiente:

1) La función **`gethostbyname`**

```
struct hostent *gethostbyname(const char *nombre);
```

Esta función, que recibe como parámetro un **nombre** de servidor, lo busca en la base de datos correspondiente y, si lo encuentra, llena una variable de tipo `struct hostent` con información sobre el servidor. El valor retornado identifica esta variable (las llamadas posteriores utilizarán la misma variable para dejar el resultado en la misma). Si no se puede encontrar el nombre, el valor retornado es NULL.

El tipo `struct hostent`, que sirve para representar las entradas de la base de datos de nombres de servidores, se define en el fichero `<netdb.h>` de la manera siguiente:

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

– El campo **`h_name`** es el nombre "oficial" del servidor.

– El campo **`h_aliases`** es una lista de punteros que apuntan a diferentes alias o nombres alternativos de que puede disponer el servidor. Para saber dónde acaba la lista, después del puntero en el último alias, hay un puntero igual a NULL (si el primer puntero es NULL, significa que la lista está vacía).

– El campo **`h_addrtype`** indica el formato de la dirección o direcciones del servidor* y, por norma general, será igual a `AF_INET`.

*** Recordad que un *host* puede tener más de una dirección IP.**

– **El** campo **`h_length`** señala la longitud en bytes de cada una de las direcciones del servidor. Si el tipo de dirección es `AF_INET`, este campo será igual a 4.

– El campo `h_addr_list` es una lista de punteros que apuntan a las direcciones del servidor. La lista acaba con un puntero igual a NULL. Si las direcciones son del tipo

`AF_INET`, cada una se representa con 4 *bytes* que ya estarán ordenados de acuerdo con el orden de la red.

Para hacerlo compatible con versiones anteriores de la librería, así como para facilitar el acceso a la primera dirección de la lista, el fichero `<netdb.h>` suele incluir esta definición:

```
#define h_addr h_addr_list[0]
```

2) La función **`gethostbyaddr`**

```
struct hostent *gethostbyaddr(const char *adr,
                    int long_adr, int formato);
```

Esta función realiza una búsqueda inversa en la base de datos: dada la dirección especificada por los parámetros, retorna la información del servidor que tiene esta dirección o retorna NULL si no la puede encontrar. Si el tercer parámetro es `AF_INET` (el caso más habitual), el primero debe ser una secuencia de 4 bytes en el orden de la red y el segundo debe ser igual a 4.

Errores en la búsqueda

En muchos sistemas, cuando el valor retornado por las funciones `gethostbyname` y `gethostbyaddr` es NULL, se indica la causa del error en la búsqueda en una variable global denominada `h_errno`, la cual se declara en el fichero `<netdb.h>`. Este mismo fichero contiene las definiciones de constantes que sirven para representar las causas de error siguientes:

- `HOST_NOT_FOUND`: no hay ningún servidor con este nombre en la base de datos.
- `TRY_AGAIN`: el servidor DNS no responde, o bien ha enviado una respuesta sin autoridad diciendo que el servidor no existe (en este caso, puede ser que el nombre sí que figure en la base de datos, pero que se haya añadido recientemente).
- `NO_RECOVERY`: error en la consulta DNS (formato incorrecto, petición rechazada, etc.).
- `NO_ADDRESS`: el nombre existe, sin embargo no corresponde a un servidor (puede ser, por ejemplo, un nombre de dominio).

Respuesta con autoridad

Las respuestas de un servidor DNS pueden ser con autoridad o sin la misma, dependiendo de si la información que proporcionan proviene del mismo servidor o de una respuesta enviada con anterioridad por otro servidor.

3) La función **`getservbyname`**

```
struct servent *getservbyname(const char *nombre,
                      const char *protocol);
```

Esta función busca en la base de datos de servicios uno que tenga el **nombre** conferido por el primero parámetro y que utilice el **protocolo** indicado por el segundo parámetro, que por lo general será TCP o UDP. Si lo encuentra, retorna la dirección de una variable de tipo `struct servent` que contiene información sobre el servicio (las llamadas posteriores utilizarán la misma variable para dejar el resultado en la misma) y, si no lo encuentra, retorna NULL.

El tipo que representa las entradas de la base de datos de servicios se define en el fichero <netdb.h> de la manera siguiente:

Protocolos utilizados en los servicios

Existen servicios que sólo se utilizan con TCP, los hay que sólo con UDP, así como otros que se pueden utilizar tanto con TCP como con UDP. En este último caso, sin embargo, los números de puerto utilizados con los dos protocolos por lo general coinciden.

```
struct servent
{
   char *s_name;
   char **s_aliases;
   int s_port;
   char *s_proto;
};
```

- El campo **s_name** contiene el nombre "oficial" del servicio.
- El campo **s_aliases** contiene una lista de alias del servicio acabada en NULL.
- Los dos bytes de menos peso del campo **s_port** representan el número de puerto asociado al servicio (en el orden de la red).
- El campo **s_proto** indica el protocolo que es preciso utilizar con dicho servicio.

4) La función **getservbyport**

```
struct servent *getservbyport(int puerto,
                const char *protocolo);
```

Es la función inversa a la anterior: encuentra un servicio a partir de su número de **puerto** (representado de la misma manera que en el campo s_port del tipo struct servent) y el **protocolo** que utiliza.

Relleno de una dirección Internet con los nombres deseados

Podríamos reescribir la función llenar_dir_internet para que recibiera como parámetros el nombre del servidor, el nombre del servicio y el protocolo:

Consultad el ejemplo de relleno de una dirección Internet en el subapartado 2.1.2 de este módulo didáctico.

```
int llenar_dir_internet(struct sockaddr_in *adr,
      char *host, char *servicio, char *protocolo)
{
    struct hostent *h;
    struct servent *s;
    if ((h = gethostbyname(host)) == NULL) return 1;
    if ((s = getservbyname(servicio, protocolo)) == NULL)
        return 2;
    adr->sin_family = AF_INET;
    adr->sin_addr = *(struct in_addr *)h->h_addr;
    adr->sin_port = s->s_port;
    return 0;
}
```

No es preciso utilizar las funciones hton...

... puesto que ya obtenemos los nombres directamente en el orden de la red.

Ejemplo de uso de la función anterior (el nombre http es un alias típico de www):

```
error = llenar_dir_internet(&adr, "www.uoc.es", "http", "tcp");
```

2.4.4. Acceso secuencial a las bases de datos

Además del acceso aleatorio a las bases de datos de nombres por medio de las funciones que acabamos de ver, también es posible realizar un acceso secuencial a las mismas. Ello permite, por ejemplo, recorrer toda la base de datos de servidores o servicios conocidos, o listar los nombres que satisfacen una determinada condición.

Las funciones siguientes, declaradas en el fichero `<netdb.h>`, proporcionan el acceso secuencial a las bases de datos, y lo hacen de manera homogénea y con independencia de si los nombres están almacenados en un fichero o si deben obtenerse por medio del servicio NIS:

- `void sethostent(int no_cerrar)`
- `void setservent(int no_cerrar)`
- `struct hostent *gethostent(void)`
- `struct servent *getservent(void)`
- `void endhostent(void)`
- `void endservent(void)`

Las dos primeras dejan las bases de datos respectivas preparadas para empezar el acceso secuencial a partir del primer elemento. Si el parámetro es diferente de 0, la base de datos quedará abierta para las operaciones siguientes y, si es 0, se abrirá y se cerrará a cada acceso que se haga a la misma. Las funciones `gethostent` y `getservent` retornan el elemento siguiente de la base de datos, o NULL si ya no quedán más. El orden en que se obtienen los diferentes elementos no es significativo; lo único que es conveniente saber es que estas funciones las leen todos antes de retornar NULL. Las dos últimas cierran la base de datos correspondiente.

La función `gethostent`...

... no utiliza el servicio DNS para acceder secuencialmente a la base de datos de nombres de servidores, sino sólo la base de datos local o el servicio NIS.

2.4.5. Opciones de los *sockets*

Es posible variar algunos aspectos del modo de funcionamiento de los *sockets* modificando una serie de opciones. La interfaz de programación proporciona una llamada para cambiar el valor de una opción en un *socket* y otra para consultar su valor actual. Los prototipos de estas llamadas se declaran en el fichero cabecera `<sys/socket.h>` de la manera siguiente:

1) La llamada **`setsockopt`**: cambia el valor de una opción en un *socket*.

```
int setsockopt(int descr, int nivel, int nombre_opc,
           const void *val_opc, size_t long_opt);
```

- El parámetro **`descr`** es el descriptor del *socket*.
- El parámetro **`nivel`** y el **`nombre_opc`** identifican la opción, como veremos a continuación.

- El parámetro **`val_opc`** apunta a una variable que contiene el valor que debe asignarse a la opción.
- El parámetro **`long_opt`** indica cuántos *bytes* ocupa este valor.

La mayoría de los valores de las opciones se representan con enteros (`int` o `size_t`); sin embargo, los hay que se representan con otros tipos. Por este motivo, el valor se pasa a la llamada por medio de un puntero genérico (`void *`) y de su longitud en bytes.

Si la llamada tiene éxito, retorna 0 y, si no lo tiene, retorna -1 y asigna a la variable global `errno` un código correspondiente a la causa del error.

2) La llamada **`getsockopt`**: retorna el valor actual de una opción en un *socket*.

```
int getsockopt(int descr, int nivel, int nombre_opc,
                void *val_opc, size_t *long_opt);
```

Los tres primeros parámetros son como los de la llamada `setsockopt`.

El cuarto parámetro, **`val_opc`**, es un puntero que indica dónde debe dejarse el valor leído.

El quinto parámetro, **`long_opt`**, debe contener al principio el número máximo de bytes que debe ocupar este valor; la llamada le asignará el número de bytes que ocupa en realidad.

El valor retornado y los códigos de error son los mismos que en la llamada `setsockopt`.

Cuando el valor retornado por `setsockopt` o `getsockopt` indica que ha habido error, la variable `errno` puede tener uno de los valores siguientes:

- `EBADF`: el primer parámetro no es un descriptor válido.
- `ENOTSOCK`: el primer parámetro es un descriptor; sin embargo, no corresponde a un *socket*.
- `ENOPROTOOPT`: el valor del tercer parámetro no corresponde a ninguna opción del nivel indicado por el segundo. Las opciones que se pueden especificar para un *socket* se agrupan en diferentes niveles identificados por el segundo parámetro de estas llamadas y se representan con alguna de las constantes siguientes, que están definidas en el fichero `<sys/socket.h>`:
  - `SOL_SOCKET`: para las opciones en el ámbito de la interfaz de *sockets*.
  - `SOL_TCP`, `SOL_UDP`, `SOL_IP`, etc.: para las opciones relativas al protocolo de comunicaciones correspondiente.

Según el valor del segundo parámetro de las llamadas anteriores, tendremos diferentes tipos de opciones:

a) Opciones de *socket*: algunas de las opciones de *socket* (segundo parámetro igual a `SOL_SOCKET`), en que se indica el nombre definido en el fichero `<sys/socket.h>` que es preciso utilizar como tercer parámetro de las llamadas y el tipo del valor correspondiente al cuarto parámetro, son las siguientes:

- `SO_REUSEADDR` (tipo `int`): si el valor de la opción es diferente de 0, significa que la llamada `bind` permitirá asignar al *socket* un número de puerto que ya sea utilizado por otro.

Un ejemplo...

... de aplicación que puede requerir el uso de diferentes *sockets* con el mismo puerto es la transferencia de ficheros por medio del FTP.

- `SO_KEEPALIVE` (tipo `int`): si el valor es diferente de 0 y el protocolo asociado al *socket* es orientado a conexión, se enviarán de manera automática mensajes periódicos al *socket* remoto para comprobar si la comunicación continúa establecida y, si no se recibe ninguna respuesta, se dará la conexión por cerrada. Esta opción permite detectar cortes en la comunicación aunque no se intercambien datos.

- `SO_DONTROUTE` (tipo `int`): si el valor es diferente de 0, se aplica automáticamente el parámetro indicador `MSG_DONTROUTE` a todos los datos enviados.

Consultad el significado del parámetro indicador `MSG_DONTROUTE` en el subapartado 2.1.3 de este módulo didáctico.

- `SO_LINGER` (tipo `struct linger`): esta opción indica cómo debe actuar el *socket* si utiliza un protocolo de transmisión fiable y se le aplica la llamada `close` cuando todavía tiene datos pendientes de enviar. El tipo `struct linger` se define en el fichero `<sys/socket.h>` de la manera siguiente:

```
struct linger
{
   int l_onoff;
   int l_linger;
};
```

  - Si el campo **`l_onoff`** es diferente de 0, significa que la llamada `close` retornará cuando se hayan enviado los datos pendientes o haya transcurrido el número de segundos indicado por el campo **`l_linger`**.
  - Si el campo **`l_onoff`** es 0, la llamada retornará inmediatamente.

- `SO_BROADCAST` (tipo `int`): si el valor es diferente de 0, se pueden enviar mensajes de difusión (*broadcast*) por el *socket*.

- `SO_OOBINLINE` (tipo `int`): si el valor es diferente de 0, los datos urgentes o fuera de banda que se reciban se tratarán de la misma manera que los datos normales.

- `SO_SNDBUF` y `SO_RCVBUF` (tipo `size_t`): el valor de estas opciones es la longitud en *bytes* del *buffer* de salida y la del de entrada, respectivamente, asociadas al *socket*.

- SO_STYLE (tipo int): esta opción es aplicable sólo a la llamada getsockopt. Su valor es el estilo de comunicación que utiliza el *socket* (SOCK_STREAM, SOCK_DGRAM, etc.)*.

*** En versiones anteriores de la interfaz de los *sockets*, la opción SO_STYLE se denomina SO_TYPE.**

- SO_ERROR (tipo int): esta opción también es aplicable sólo a la llamada getsockopt. Su valor es la última condición de error que se ha producido en el *socket* (este valor se reinicializa una vez se ha consultado con getsockopt).

Ejemplo de tiempo de cierre de conexión de un *socket*

La función tiempo_espera_cierre sirve para ajustar el tiempo que debe esperar un *socket* antes de dar por cerrada una conexión cuando no puede enviar los últimos datos que le han pasado:

```
int tiempo_espera_cierre(int sock, int tiempo)
{
   struct linger l;

   l.l_onoff = 1;
   l.l_linger = tiempo;
   return setsockopt(sock, SOL_SOCKET, SO_LINGER, &l, sizeof l);
}
```

b) Opciones de TCP. Algunos ejemplos de opciones de TCP (segundo parámetro igual a SOL_TCP) son los siguientes:

- TCP_MAXSEG: indica la longitud máxima de los segmentos TCP.

- TCP_NODELAY: indica si debe aplicarse el algoritmo de Nagle a la hora de decidir cuándo era preciso enviar un paquete o si no debe hacerse.

c) Opciones de IP. Algunos ejemplos de opciones de IP (segundo parámetro igual a SOL_IP) son los siguientes:

- IP_TOS: representa el tipo de servicio.
- IP_TTL: representa el tiempo de vida de los datagramas.

3. *Sockets* con lenguaje Java

En el apartado anterior, hemos visto cómo en C hay una función para cada una de las operaciones que deben llevarse a cabo en los dos extremos de la comunicación y cómo podemos ajustar su comportamiento por medio de un ingente número de parámetros. Ello es así por la filosofía de diseño del mismo lenguaje C: acercarse al máximo al funcionamiento de la máquina.

En cambio, el lenguaje Java, y los lenguajes orientados a objetos en general, persigue unos objetivos bien diferentes: esconder detalles de implementación y proporcionar al programador esencialmente lo que necesita.

3.1. Las clases *Socket* y *ServerSocket*

El mecanismo principal para conseguir este nivel de abstracción y encapsulamiento en un lenguaje orientado a objetos es la **clase**. La clase *Socket*, que puede encontrarse en el paquete *java.net*, es la que implementa toda la funcionalidad de los *sockets*, con el espacio de nombres Internet.

El constructor de la clase es el equivalente a la función `socket` de la librería de C, la que realiza la operación de creación del *socket*, mientras que el resto de las operaciones que hemos descrito con anterioridad (asignar dirección, escuchar peticiones, aceptar conexiones, etc.) constituyen métodos de la clase.

Para simplificar la gestión en ambos extremos de la comunicación, existe otra clase, la *ServerSocket*, pensada para ser utilizada en el extremo del servidor. Su constructor se encarga de crear el *socket*, vincularlo a una dirección y crear la cola en que se almacenarán las peticiones de conexión todavía no atendidas. De este modo, la clase *Socket* se utiliza en el extremo del cliente, en que el constructor realiza la creación del *socket* y la petición de conexión con el servidor.

Clase: *Socket*

Constructor:

```
Socket(InetAddress adr, int port)
Socket(InetAddress adr, int port, InetAddress adr_local,
   int port_local)
Socket(string maquina, int port)
```

WEB

Existen ocho versiones del constructor de la clase Socket que se pueden consultar en la web de Sun, los creadores del lenguaje: http://java.sun.com/j2se/1.4.1/docs/api/java/net/Socket.html.

La manera de especificar el servidor al que se quiere conectar condiciona cuál de los constructores se ejecuta:

- Una manera consiste en pasar por parámetro la dirección IP y el puerto en que se quiere realizar la conexión.
- Una segunda manera de hacerlo consiste en suministrarle, además de lo anterior, la dirección y el puerto locales que deben utilizarse.
- Una tercera manera consiste en especificar la máquina remota con el nombre, y no con la dirección IP.

Formato

Las direcciones IP se especifican con objetos de la clase `InetAddress`, mientras que los nombres de las máquinas se especifican con simples cadenas de texto.

Clase: *`ServerSocket`*

Constructor:

```
ServerSocket(int port)
ServerSocket(int port, int n_petic)
ServerSocket(int port, int n_petic, InetAddress adr)
```

De nuevo, la sobrecarga proporciona diferentes versiones del constructor, según los parámetros que interese suministrar:

- En el primer caso, sólo el puerto en que debe vincularse el *socket*.
- En el segundo, el puerto y el tamaño de la cola.
- En el tercero, además de estos dos, la dirección IP de la interfaz a la que se quiere vincular el *socket*.

Una vez creado el objeto de la clase *`ServerSocket`*, el método `accept` (análogo a la función `accept` de la librería de C) se encarga de esperar conexiones de clientes por medio del *socket*. Cuando se haya efectuado la conexión, ya podrá empezar el intercambio de información.

Clase: *`ServerSocket`*

Método:

```
Public Socket accept()
```

WEB

Hay más de treinta métodos definidos en la clase *Socket*, que sirven para realizar operaciones auxiliares o cambiar opciones de trabajo.
La diferencia con el lenguaje C es que ni siquiera es necesario conocerlos para un funcionamiento básico de los *sockets*.
Se pueden consultar todos en la web de Sun: http://java.sun.com/j2se/1.4.1/

Este método se bloquea esperando una petición de conexión desde un cliente y, cuando llega, continúa la ejecución del programa servidor. El resultado que

retorna es un nuevo *socket*, que es donde deben leerse/escribirse los datos que se quieren intercambiar con el *socket* remoto.

En el extremo del cliente, la creación de un objeto de la clase *Socket* incluye, asimismo, la solicitud de conexión. Por tanto, una vez aceptada esta conexión por parte del servidor, ya se puede proceder igualmente al intercambio de información.

Para leer desde el *socket*, y para escribir en el mismo, se utilizan objetos de las clases *inputStream* y *outputStream*, que se conectan al *socket* mediante los métodos `getOutputStream` y `getInputStream`.

3.2. Ejemplo de programación de un servidor y un cliente

Servidor:

```
import java.io.*;
import java.net.*;

public class ejemplo_servidor {

   public static void main(String args[]) {

      ServerSocket mi_servicio = null;
      String linea_recibida;
      DataInputStream entrada;
      PrintStream salida;
      Socket socket_conectado = null;

      try {
         mi_servicio = new ServerSocket(2000);
      }
      catch (IOException excepcion) {
         System.out.println(excepcion);
      }

      try {
         socket_conectado = mi_servicio.accept();
         entrada = new DataInputStream(socket_conectado.getInputStream());
         salida = new PrintStream(socket_conectado.getOutputStream());
         linea_recibida = entrada.readLine();
         salida.println(linea_recibida);
         salida.close();
         entrada.close();
         socket_conectado.close();
```

```
      }
      catch (IOException excepcion) {
         System.out.println(excepcion);
      }
   }
}
```

Cliente:

```
import java.io.*;
import java.net.*;

public class ejemplo_cliente {

   public static void main(String args[]) {

      Socket cliente = null;
      DataInputStream entrada = null;
      DataOutputStream salida = null;

      try {
         cliente = new Socket("servidor.uoc.edu", 2000);
         salida = new DataOutputStream(cliente.getOutputStream());
         entrada = new DataInputStream(cliente.getInputStream());
      }
      catch (UnknownHostException excepcion) {
         System.err.println("No encuentro ‘servidor.uoc.edu’");
      }
      catch (IOException excepcion) {
         System.err.println("Error de entrada/salida");
      }

      if (cliente != null && salida != null && entrada!= null) {
         try {
            String linea_recibida;
            salida.writeBytes("¡Hola, servidor!\n");
            linia_recibida = entrada.readLine();
            System.out.println("Servidor: " + linea_recibida);
            salida.close();
            entrada.close();
```

```
          cliente.close();
        }
        catch (UnknownHostException excepcion) {
        System.err.println("No encuentro 'servidor.uoc.edu'");
        }
        catch (IOException excepcion) {
          System.err.println("Error de entrada/salida");
        }
      }
   }
}
```

3.3. Comunicación con datagramas

Todo lo que hemos expuesto hasta ahora sirve para crear *sockets* con el estilo de secuencia de bytes, es decir, con el protocolo TCP.

Para crear *sockets* en el estilo datagrama, es decir, con el protocolo UDP, disponemos de las clases `DatagramSocket` y `DatagramPacket`: la primera para los objetos *sockets* UDP y la segunda para los paquetes que deben enviarse.

La mecánica es muy sencilla y la ilustramos con un ejemplo:

Servidor:

```
import java.io.*;
import java.net.*;

public class ejemplo_servidor_datagrama {

   public static void main(String args[]) throws IOException {

      byte[] buf = new byte [1024];
      DatagramSocket s = new DatagramSocket(2000);
      DatagramPacket p = new DatagramPacket(buf, 1024);
      s.receive(p);
      s.send(p);
      s.close();
   }
}
```

Cliente:

```
import java.io.*;
import java.net.*;

public class ejemplo_data_c {

   public static void main(String args[]) throws IOException {
      InetAddress adr = InetAddress.getByName("servidor.uoc.edu");
      DatagramSocket s = new DatagramSocket();
      DatagramPacket p = new DatagramPacket ("hola\n".getBytes(), 5, adr, 2000);
      s.send(p);
      s.receive(p);
      System.out.println (p.getAddress().getHostAddress());
      s.close();
   }
}
```

Resumen

A continuación, presentamos las principales funciones de C relacionadas con la interfaz de programación de *sockets* y sus formatos.

1) Crear un *socket*:

```
int socket(int espacio, int estilo, int protocolo);
```

2) Asignar dirección a un *socket*:

```
int bind(int descr, const struct sockaddr *adr,
         size_t long_adr);
```

3) Dejar un *socket* preparado para recibir peticiones de conexión:

```
int listen(int descr, unsigned int n_petic);
```

4) Esperar que haya disponibilidad de datos en un *socket*:

```
int select(int n_descr, fd_set *d_lect, fd_set *d_escr,
         fd_set *d_excep, struct timeval *tiempo);
```

5) Enviar una petición de conexión:

```
int connect(int descr, const struct sockaddr *adr,
           size_t long_adr);
```

6) Aceptar una petición de conexión:

```
int accept(int descr, struct sockaddr *adr,
           size_t *long_adr);
```

7) Leer datos de un *socket*:

```
int recvfrom(int descr, void *datos, size_t longitud,
   int flags, struct sockaddr *adr, size_t *long_adr);
```

```
int recv(int descr, void *datos, size_t longitud,
         int flags);
```

```
ssize_t read(int descr, void *datos, size_t longitud);
```

8) Escribir datos en un *socket*:

```
int sendto(int descr, const void *datos, size_t longitud,
   int flags, const struct sockaddr *adr, size_t long_adr);
```

```
int send (int descr, const void *datos,
          size_t longi tud, int flags);
```

```
ssize_t write(int descr, const void *datos,
              size_t longitud);
```

9) Cambiar opciones de un *socket* y consultarlas:

```
int setsockopt(int descr, int nivel, int nombre_opc,
               const void *val_opc, size_t long_opt);
```

```
int getsockopt(int descr, int nivel, int nombre_opc,
               void *val_opc, size_t *long_opt);
```

10) Cerrar un *socket*:

```
int shutdown(int descr, int modo);
```

```
int close(int descr);
```

Por lo que respecta al lenguaje Java, las clases `ServerSocket` (para los servidores) y `Socket` (para los clientes) son las que incluyen toda la funcionalidad de los *sockets* con el espacio de nombres Internet y orientados a conexión.

Constructores:

```
ServerSocket(int port)
ServerSocket(int port, int n_petic)
ServerSocket(int port, int n_petic, InetAddress adr)
```

```
Socket(InetAddress adr, int port)
Socket(InetAddress adr, int port, InetAddress
       adr_local, int port_local)
Socket(string maquina, int port)
```

Además de los constructores, sólo es necesario ejecutar el método `accept` en el extremo del servidor para tener dos *sockets* conectados, a punto de intercambiarse información.

Las clases `DatagramSocket` y `DatagramPacket` permiten la comunicación de *sockets* en modo no orientado a conexión (con protocolo UDP).

Actividades

1. En un sistema UNIX, inspeccionad el contenido del fichero `/etc/inetd.conf`. ¿Qué servicios internos ofrece el proceso `inetd`? ¿Hay algún servidor que no se ejecute como `root`?

Si encontráis que hay una serie de servicios (no internos) para los cuales el valor del sexto campo es igual, probablemente en el sistema hay instalado un programa llamado *wrapper*, que actúa de intermediario entre el servidor `inetd` y los procesos hijos. Averiguad para qué sirve un *wrapper*.

2. Intentad establecer una conexión a alguno de los servicios internos que ofrece el servidor `inetd`. Desde el mismo sistema ello puede llevarse a cabo, por ejemplo, con lo siguiente:

```
telnet localhost daytime
```

El segundo argumento (opcional) del comando `telnet` es un nombre de servicio o un número de puerto en el que se puede establecer la conexión. El comando acaba cuando el servidor cierra la conexión, o bien cuando la cierra el usuario (tecleando el carácter de escapada, que por norma general és '^]' y entrando el comando `quit`).

3. En el sistema UNIX al que habéis accedido, ¿se encuentra disponible el comando `netstat`? Si lo está, intentad hacer `netstat -a`. ¿Podéis interpretar la información que proporciona? Probablemente, os será útil haber hecho antes `man netstat`.

4. Inspeccionad el contenido de los ficheros `/etc/hosts` y `/etc/services`, y observad su formato. Si el sistema utiliza el servicio NIS, podéis listar las bases de datos haciendo `ypcat hosts` e `ypcat services`.

Ejercicios de autoevaluación

1. ¿Por qué creéis que no se permite a los procesos de usuario (no privilegiados) crear *sockets* con el estilo `SOCK_RAW`?

2. Explicad qué diferencia existe entre trabajar con el estilo de comunicación secuencia de bytes y con el estilo datagrama en los *sockets* del espacio de nombres de ficheros.

3. De las condiciones de error con que puede retornar la llamada `connect`, indicad cuáles se pueden dar en *sockets* del espacio de nombres de ficheros o del espacio Internet, en *sockets* orientados a conexión o no orientados a conexión y en *sockets* bloqueadores o no bloqueadores.

Solucionario

Ejercicios de autoevaluación

1. Uno de los motivos es porque, de este modo, un usuario podría ejecutar un programa que generara paquetes TCP o UDP con puertos de origen reservados. Aparte de ello, un usuario inexperto, o malintencionado, podría generar paquetes con cabeceras incorrectas que provocaran errores en los equips de la red.

2. Suponiendo que, en el espacio de nombres de ficheros, los datagramas no se perdieran ni se duplicaran o se desordenaran, la diferencia es que puede suceder que las operaciones de lectura en el estilo datagrama no lean todos los datos. Éste sería el caso si el número de bytes que se quisiera leer fuera menor que la longitud del datagrama. En cambio, en el estilo secuencia de bytes, no se perderá ningún dato, sea cual sea la longitud de la memoria intermedia de lectura.

3. La relación entre los diferentes tipos de *sockets* y los posibles errores se expone en el cuadro siguiente:

Tipos de *socket*	Tipos de error posible
Todos los *sockets*	`EBADF, ENOTSOCK, EINVAL` y `EAFNOSUPPORT`
Del espacio de nombres de ficheros	`ENOENT,ENOTDIR, EACCESS` y `EPROTOTYPE`
Del espacio de nombres Internet	`ENETUNREACH` y `ETIMEDOUT`
Orientados a conexión	`EISCONN, ENETUNREACH, ETIMEDOUT, ECONNREFUSED` y `EINPROGRESS`
No orientados a conexión	No hay ninguno específico.
Bloqueadores	`ETIMEDOUT` y `EINTR`
No bloqueadores	`EINPROGRESS` y `EALREADY`

Glosario

algoritmo de Nagle *m* Algoritmo para minimizar el número de paquetes TCP que deben transmitirse cuando la red está cargada.

dato fuera de banda *m* Dato urgente que se transmite al destinatario con una prioridad más alta que el dato normal.

espacio de nombres *m* Dominio de comunicaciones al que pertenece un *socket* y que determina el formato de su nombre o dirección. Hay dos básicos: el de ficheros y el de Internet.

estilo de comunicación *m* Semántica asociada a la emisión y recepción de datos por medio de un *socket*. El estilo de comunicación determina, por ejemplo, cuál es la unidad de datos transmitida (*bytes* o datagramas), si se prevé la posibilidad de perder datos, o si la comunicación requiere el establecimiento de una conexión, o bien cada paquete especifica su dirección de destino.

familia de protocolos *f* Conjunto de protocolos de comunicación que se pueden utilizar con los *sockets* que pertenecen a un determinado espacio de nombres.

orden de *bytes* de la red *m* Orden en el que se transmiten los bytes que representan un número binario; el *byte* de mayor peso se transmite el primero y el de menor peso, el último.

tipos de *socket* *m pl* Conjunto de propiedades de los *sockets* que utilizan un mismo estilo de comunicación.

Bibliografía

Loosemore, S.; Stallman, R.M.; McGrath, R.; Oram, A.; Drepper, U. (1992). *The GNU C Library Reference Manual*. Boston: Free Software Foundation.

Márquez García, F.M. (1996). *UNIX. Programación avanzada*. Madrid: Ra-ma.

Orfali, R. (1998). *Client/server programming with Java and CORBA*. Nueva York: Wiley & sons.

Rifflet, J.M. (1992). *Comunicaciones en UNIX*. Madrid: McGraw-Hill.