# Programación en red sobre TCP/IP
# Interface sockets

**Teresa Monreal y Pablo Ibáñez**

**Area de Arquitectura y Tecnología de Computadores**

**Departamento de Informática e Ingeniería de Sistemas**

## Indice

**Introducción.**

**Diseño en capas.**

**El modelo OSI.**

**La familia de protocolos TCP/IP.**

**Añadiendo E/S de red a UNIX.**

**La abstracción socket.**

**El modelo cliente-servidor.**

**Resumen de llamadas al sistema.**

## Introducción.

Hasta hace no mucho tiempo un computador era concebido como una entidad aislada, cada computador era autosuficiente y tenía todos los periféricos y software necesarios para realizar una determinada tarea. Si se requería una nueva funcionalidad como poder imprimir texto sobre papel se añadía una impresora al sistema. Si se necesitaba más espacio para guardar información se añadía un nuevo disco al computador.

Por otra parte, la transferencia de información entre usuarios se realizaba físicamente pasando cintas, discos o listados.

La posibilidad de interconectar sistemas formando redes de computadores ha cambiado totalmente la compartición tanto de recursos (impresoras, discos...) como de información.

Una red puede ser tan simple como dos ordenadores personales conectados mediante un modem a 1200 baudios o tan compleja como TCP/IP Internet que conecta varios cientos de miles de computadores de todo el mundo.

Utilidades típicas que un usuario busca al conectarse a una red son:

1. Correo electrónico con usuarios de otros computadores.
2. Intercambio de ficheros entre distintos sistemas.
3. Compartición de ciertos periféricos (impresoras laser...).
4. Ejecución de programas en otros sistemas.
5. Login remoto.

Con esta documentación solo pretendemos introducir los conceptos más elementales en el diseño de aplicaciones en red, la familia de protocolos TCP/IP y el interface que 4BSD UNIX ofrece para trabajar con esa familia. Podeis conseguir más información en la siguiente bibliografía.

Sobre conceptos básicos de redes:
"Redes De Ordenadores", A. S. Tanenbaum. Prentice Hall Hispanoamericana, 1991.
Sobre la familia de protocolos TCP/IP y el interface con 4BSD UNIX:
"UNIX Network Programming", W. R. Stevens. Prentice Hall, 1990.
"Internetworking With TCP/IP, Volume I", D. E. Comer. Prentice Hall, 1991.

Por otro lado no olvideis que teneis una descripción más detallada de todas las funciones que proporciona UNIX en el man del sistema.

## Diseño en capas.

Dado un particular problema que deseamos resolver, como puede ser ofrecer alguna forma de intercambiar ficheros entre dos computadores, un buen método de diseño consiste en dividir el problema en subtareas y resolver cada una de ellas, uniéndolas finalmente para resolver el problema completo. Además de las ventajas relativas a la facilidad de diseño y corrección de los algoritmos resultantes, si además buscamos productividad existen otras ventajas no tan evidentes pero igualmente importantes. Es posible que una parte de la solución desarrollada para nuestro transmisor de ficheros pueda ser usada cuando deseemos implementar un impresor remoto. También es posible que una vez diseñado nuestro transmisor de ficheros para operar entre dos ordenadores unidos mediante Ethernet, podamos usar parte del código para intercambiar ficheros entre computadores unidos por una línea telefónica.

En el contexto de las redes de computadores a esto se le llama diseño por niveles (layering) y es algo tan asumido que existe una norma ISO(International Standards Organization) que describe un modelo standard para el diseño de aplicaciones en red, el OSI(Open Systems Interconnection).

## El modelo OSI.

El modelo OSI descompone el diseño de una aplicación en red en siete niveles que se muestran en la figura 1.

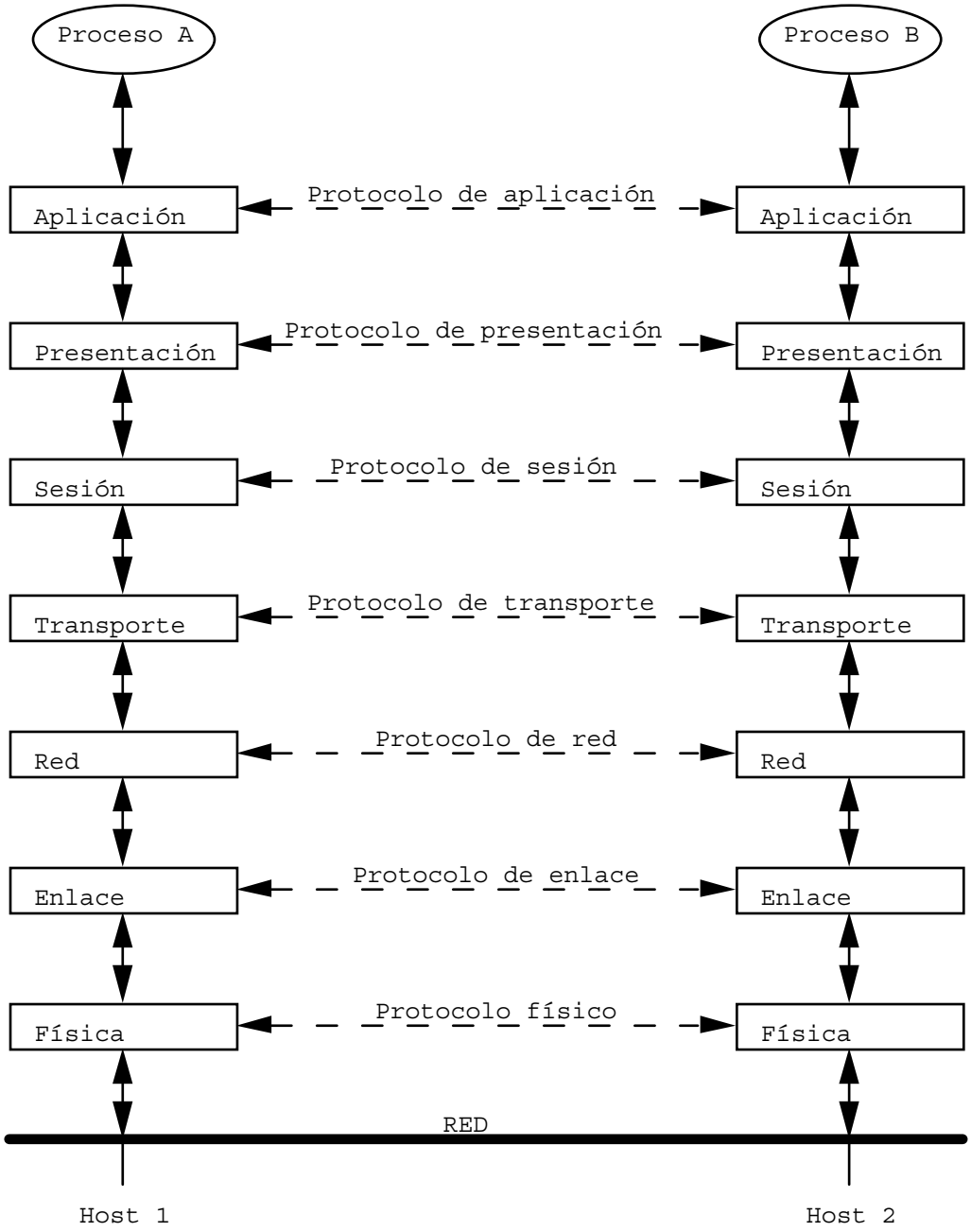



Figura 1. Descomposición del diseño de una aplicación en red según el modelo OSI.

En cada host, cada capa se construye sobre su predecesora y su propósito es siempre ofrecer servicios a las capas superiores liberándolas del conocimiento detallado de como se realizan dichos servicios.

La capa n de una máquina conversa con la capa n de otra máquina. Esta conversación consiste en el envio y recepción de datos y señales de control. Las reglas y convenciones utilizadas en esta conversación se conocen conjuntamente como protocolo de la capa n. En realidad no existe una transferencia directa de datos entre la capa n de una máquina y la de otra, sino que cada capa pasa la información de datos y control a la capa inmediatamente inferior y así sucesivamente hasta la capa 1 que controla directamente el medio físico a través del cual se realiza la transferencia efectiva de datos de una máquina a otra.

Entre cada par de capas adyacentes hay un interfase que define el conjunto de servicios y operaciones primitivas que la capa inferior ofrece a la superior. A continuación se describen muy brevemente los servicios ofrecidos por cada una de las 7 capas del modelo OSI:

**Capa física (N1).**

Se ocupa de la transmisión de bits a lo largo de un canal de comunicación. Su diseño debe asegurar que cuando un extremo envía un bit con valor 1 éste se reciba exactamente con el mismo valor en el otro extremo. El protocolo de nivel 1 asegura que los detalles eléctricos y mecánicos y las señales de control entre los dos extremos son los correctos.

**Capa de enlace (N2).**

Es capaz de transmitir datos entre dos máquinas conectadas entre sí de forma directa mediante un canal de comunicación, asegurando además que en dicha transmisión no hay errores.

**Capa de red (N3).**

No siempre tendremos las máquinas origen y destino de la comunicación unidas por un canal directo. Es más común que formen parte de una red más o menos jerarquizada en la que la información debe pasar por varias máquinas antes de llegar a su destino. De los mecanismos de encaminamiento de la información se encarga la capa de red.

**Capa de transporte (N4).**

Se encarga de aislar al resto de los niveles de todos los detalles de la red. El nivel de transporte ofrece a los niveles superiores conexiones virtuales fiables entre dos máquinas cualesquiera de la red. Puede ofrecer varias formas de transporte: con o sin

conexión (connection-oriented or connectionless, teléfono/correo) con más o menos control de calidad.

Hacemos un alto en este punto ya que entre el nivel de transporte y los superiores existe una barrera importante.

Como hemos visto, los cuatro niveles inferiores están orientados claramente a la organización de las comunicaciones, los tres superiores ya gozan de un mecanismo de comunicación seguro y de alto nivel y estarán orientados más bien a la organización de las aplicaciones.

Como consecuencia de esto, los cuatro niveles básicos estarán normalmente integrados en el hardware de la máquina y en el software de su sistema operativo. Los servicios que ofrece el nivel 4 aparecerán por tanto como funciones de librería utilizables directamente por el usuario.

**Capa de sesión (N5).**

Esta capa es esencialmente una versión ampliada del nivel 4, añadiendo control de diálogo y posibilidades de sincronización entre las dos aplicaciones que intentan comunicarse.

**Capa de presentación (N6).**

Se encarga de uniformizar la sintaxis de todas las máquinas, solucionando el problema que plantea el hecho de que un mismo dato (supongamos un entero) sea representado de varias formas en distintos computadores.

**Capa de aplicación (N7).**

Contiene un abanico de protocolos usados frecuentemente en entornos distribuidos como correo electrónico, transmisión de ficheros, conexión de terminales remotas, ejecución remota...

## La familia de protocolos TCP/IP.

Un gran número de redes se encuentran funcionando actualmente en todo el mundo. No todas ellas han sido diseñadas respetando el modelo OSI, este es el caso de la más importante de ellas por el número de máquinas que conecta: Internet.

La familia de protocolos TCP/IP fue creada a finales de los años 60 cuando aún no se había definido el modelo OSI. Consta de una serie de protocolos que cubren todos los niveles OSI uniendo varios de ellos. A continuación presentamos la descomposición en niveles de la familia TCP/IP junto con alguno de sus protocolos:

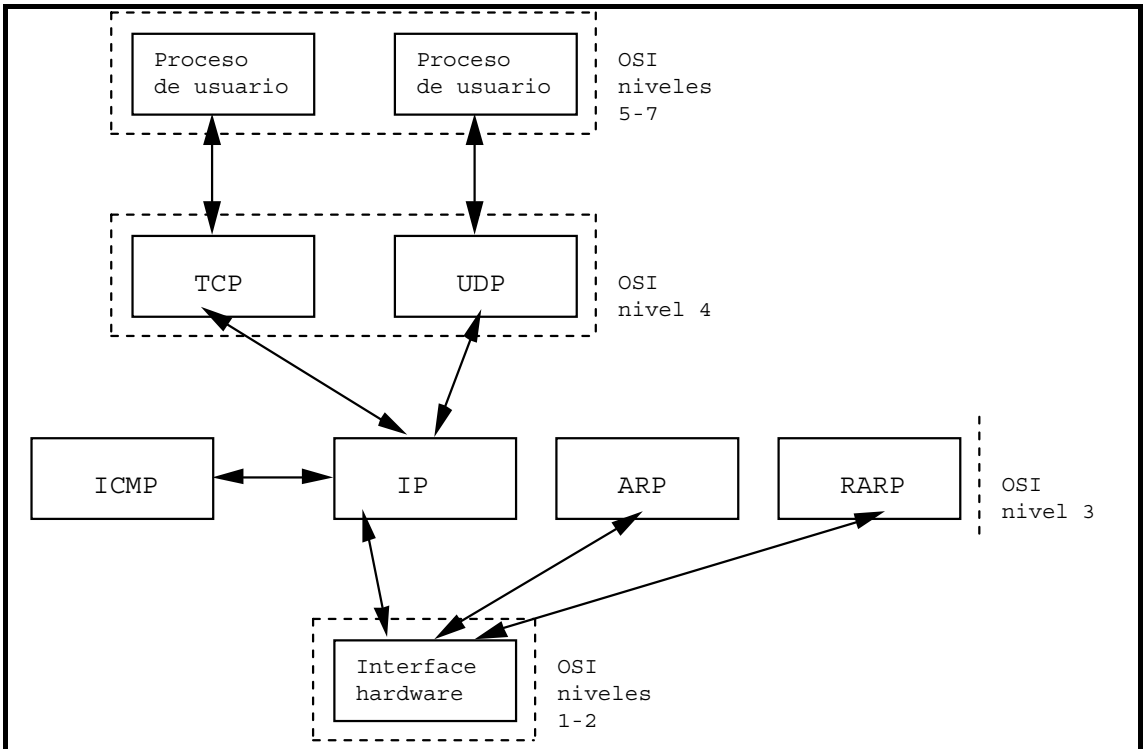
Data Link Layer: cubre los niveles 1 y 2 de OSI. Existen multitud de protocolos usados por TCP/IP ya que también existen muchos canales físicos de comunicación que permiten la conexión entre dos máquinas: líneas telefónicas, Ethernet, token ring, línea serie...

Internet Protocol (IP): se corresponde con el nivel de red de OSI.

Transmission Control Protocol (TCP) y User Datagram Protocol (UDP) que cubren el nivel de transporte dando servicios con caracteristicas distintas.

Los niveles 5-7 de OSI se incluyen dentro de los servicios al usuario ofrecidos por la familia TCP/IP:

Transferencia de ficheros (FTP)
Correo electrónico (SMPT)
Conexión remota (TELNET)




ICMP (Internet Control Message Protocol)
ARP (Address Resolution Protocol): mapeo de la @Internet en la @hardware.
RARP (Reverse Address Resolution Protocol): mapeo de la @hardware en la @Internet.
(ARP y RARP se usan para buscar la @Internet dada la @hardware y viceversa)

Figura 2. Niveles en la familia de protocolos TCP/IP.

**Nivel de red (IP).**

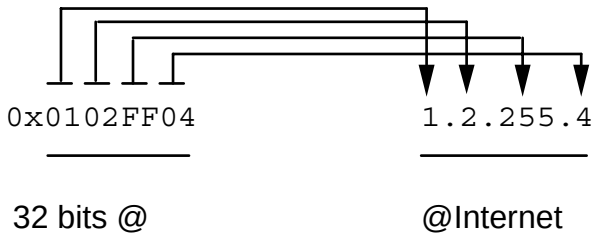
El protocolo IP de nivel de red nos proporciona un servicio sin conexión y no seguro (connectionless, datagram service).

Para comprender qué es un servicio sin conexión lo compararemos con el servicio postal. Cada mensaje lleva consigo las direcciones completas de origen y destino, por lo que cada uno de ellos se puede encaminar de forma independiente a través del sistema. Al igual que en correos, un mensaje puede sufrir ciertos retardos independientes del resto de mensajes por lo que dos mensajes con el mismo origen y destino pueden recibirse en distinto orden a como se enviaron.

No es un servicio seguro (unreliable) ya que no garantiza que todos los mensajes lleguen a su destino o que lleguen correctamente (igual que en correos). Esta será tarea de los niveles superiores.

Cada familia de protocolos define alguna forma de direccionamiento, algo que permita identificar a cada máquina dentro de la red. Una dirección Internet ocupa 32 bits y codifica un identificador de red y un identificador de host relativo a esa red

(netid/hostid). Cada host perteneciente a Internet debe tener una dirección de 32 bits única en la red, proporcionada por un organismo central (the Network Information Center (NIC)). Una dirección Internet se especifica a menudo mediante 4 números decimales separados por puntos. Cada uno de ellos codifica en decimal 2 dígitos hexadecimales de los 8 que tiene la dirección:




Por otra parte, IP acepta mensajes de cualquier tamaño menor a 64 Kbytes. IP se encargará de forma transparente de trocear estos mensajes si fuese necesario (si los niveles inferiores requieren paquetes de menor tamaño) y volver a reensamblarlos en el extremo de destino.

En realidad 64Kbytes es el máximo por definición. Cada implementación de TCP/IP impone un máximo por debajo de 64K para IP.

Podemos pensar ahora en como programar una cierta transferencia de información entre dos sistemas que sólo nos ofrecen el protocolo de nivel 3 IP.

Tendríamos dos funciones (enviar, recibir) a las que pasaríamos las direcciones origen y destino y el mensaje a transmitir. Estas funciones son los servicios que ofrece la capa 3 en nuestro caso.

Como los mensajes deben ser de un tamaño máximo, deberíamos conocer éste, trocear nuestro mensaje y enviar todos los pedazos uno a uno.

IP no asegura que el orden de emisión sea el mismo que el de recepción. Por tanto nuestra aplicación debería estar preparada para reordenar los pedazos a la llegada. Como IP no es un sistema seguro deberíamos añadir a nuestra aplicación algún método de detección/corrección de errores.

Por último, pensemos lo que puede ocurrir si dos procesos de una misma máquina A intentan comunicar con otros dos procesos de la máquina B. Cada uno de los procesos de A sólo conoce la dirección Internet de la máquina B. Cuando llegue un mensaje a B ¿a cuál de sus dos procesos se debe entregar el mensaje?. Necesitaremos implementar algún método para identificar varios destinos dentro de una misma máquina.

Se ven así claramente los objetivos del siguiente nivel.

**Nivel de transporte (TCP, UDP).**

La familia TCP/IP nos ofrece dos protocolos de nivel de transporte TCP y UDP. La diferencia fundamental es que mientras TCP es un servicio orientado a la conexión (connection-oriented, stream service), UDP no lo es. El servicio orientado a la conexión se modela basándose en el servicio telefónico. Un proceso que desea comunicarse con otro debe establecer una conexión (llamada telefónica) y una vez terminada la comunicación debe romper esa conexión (colgar el teléfono). La conexión actúa como un tubo: el que envía introduce objetos por un extremo y el receptor los recoge en el mismo orden por el otro extremo.

Por otro lado, el servicio sin conexión (connectionless) se modela con base en el servicio postal, tal como vimos para IP.

Un servicio TCP nos deberá por lo tanto ofrecer primitivas para establecer la conexión y la desconexión además de las de envío y recepción.

Por otra parte, TCP acepta mensajes de longitud arbitrariamente grandes, que deberá separar en pedazos (segmentación) que no excedan de los 64Kbytes (máximo aceptado por IP), enviándolos como mensajes separados. Como IP no garantiza el orden de los mensajes, es trabajo de TCP reensamblar de forma correcta los submensajes antes de entregar el mensaje completo y libre de errores al destinatario.

TCP utiliza además números de 16 bits para identificar varios destinos dentro de una misma máquina. Son los llamados "port numbers" y solucionan el problema planteado en el apartado anterior entre las máquinas A y B. Un proceso que quiera contactar con otro de una máquina remota debe conocer pues, además de la dirección Internet de la otra máquina, el número de puerto en el que el otro proceso está escuchando.

Un servicio UDP por otro lado, ofrece en esencia lo mismo que el protocolo IP de nivel 3, con dos diferencias fundamentales:

1. Ofrece la posibilidad de detección de errores (pero se puede perder un mensaje entero y UDP no nos avisará).
2. Al igual que TCP, gestiona números de puerto permitiendo varias comunicaciones simultáneas en una misma máquina.

Para definir una comunicación sobre el nivel 4 necesitaremos por lo tanto 5 elementos:

El protocolo de nivel 4 que queremos utilizar (TCP o UDP).
La dirección Internet de la máquina local.

El número de port en la máquina local.
La dirección Internet de la máquina remota.
El número de port en la máquina remota.

Llamaremos asociación a la 5-tupla que contiene estos elementos:

{protocolo, @host-local, port-local, @host-remoto, port-remoto}

Hasta este punto, nos hemos dedicado a estudiar los conceptos básicos del diseño de aplicaciones en red, viendo un ejemplo práctico de los niveles bajos en la familia de protocolos TCP/IP. En ningún momento hemos hablado del interface entre los programas de aplicación y el software de protocolos de nivel 4.

No hemos hablado del interface hasta este momento debido a que no es único, no está definido dentro de la especificación de la familia de protocolos. Además es lógico que asi sea, ya que TCP/IP puede estar incluido en diferentes sistemas operativos, los detalles del interface dependerán de la arquitectura de dichos sistemas operativos.

## Añadiendo E/S de red a UNIX.

Como ya conocemos, UNIX fue en un principio diseñado como un sistema de tiempo compartido para correr sobre un computador aislado con un solo procesador. La entrada/salida en UNIX sigue un paradigma del tipo open-read-write-close. Antes de que un proceso pueda realizar una operación de entrada/salida, debe realizar la llamada "*open*" para especificar el fichero o dispositivo al que se dirigirá la entrada/salida y para conseguir permisos. La llamada a "*open*" devuelve un pequeño entero que se deberá usar para realizar las entradas "*read*" y salidas "*write*" deseadas.

Una vez ha terminado el proceso de transferencia de información se debe realizar la llamada "*close*" para avisar al sistema operativo que puede liberar los objetos empleados durante dicho proceso.

Las primeras implementaciones de UNIX que incluyeron TCP/IP intentaron usar "el paradigma open-read-write-close" sobre un nombre de fichero especial "*/dev/tcp*" como interface con los protocolos.

Debido a que la entrada/salida sobre red es más compleja que la entrada/salida convencional, el grupo encargado de añadir los protocolos de red a 4BSD UNIX decidió diseñar un nuevo interface entre programas de usuario y los protocolos de TCP/IP. En particular, el interface debería permitir a los programadores crear programas siguiendo tanto el modelo servidor (crea un extremo de la conexión y es capaz de esperar que alguien se conecte al otro lado) como el modelo cliente (crea el otro extremo de la conexión suponiendo que alguien ha creado el primero). Por otra parte debe proporcionarse a los programadores alguna llamada que les permita enviar datos especificando la dirección destino (modelo sin conexión, connectionless) además de los ya existentes para lectura y escritura sobre una conexión ya establecida. El interface de UNIX se complicó todavia más debido a que se diseñó de forma que sirviera como interface no sólo de la familia de protocolos TCP/IP, sino también de la XNS (Xerox Internet Protocols) y otras.

## La abstracción socket.

La entrada/salida sobre red en UNIX 4BSD se basa en una abstracción llamada socket. Se puede pensar en un socket como en una generalización del mecanismo de acceso a ficheros en UNIX. Un socket representa uno de los extremos de una conexión. Como en el acceso a ficheros, los programas piden al sistema operativo la creación de un socket cuando les es necesario. El sistema devuelve un entero que el programa deberá usar para referirse al nuevo socket.

Se pueden solicitar varios tipos de sockets dependiendo de las características de la conexión que se necesita. En principio los dos extremos de una conexión deben tener el mismo tipo de socket, aunque no hay nada que impida lo contrario.

Se ofrecen cuatro tipos de socket:

1.Stream socket (SOCK_STREAM) ofrece una conexión bidireccional, fiable (reliable), que mantiene el orden de los datos. El flujo de datos es continuo, no se pueden especificar bloques o registros. Salvo por la bidireccionalidad, un stream socket ofrece el mismo servicio que una pipe.

2.Datagram socket (SOCK_DGRAM) ofrece un flujo de datos bidireccional sin asegurar orden de llegada, fiabilidad (unreliable) ni duplicidad. Es decir un proceso que usa datagram socket puede recibir mensajes duplicados y posiblemente en un orden distinto al que se enviaron. Una característica importante de datagram socket es que mantiene los límites de registro. Se transmiten bloques de información en lugar de bytes independientes.

3.Raw socket (SOCK_RAW) ofrece al usuario acceso directo a los protocolos que soportan la abstracción socket. Normalmente sólo son usados por usuarios que desarrollan nuevos protocolos de comunicación.

4.Sequenced packet socket (SOCK_SEQPACKET) ofrece el mismo servicio que stream socket conservando además los límites entre registros. Lo que se transmite es una secuencia ordenada de bloques de información. Este servicio sólo se ofrece para la familia de protocolos XNS.

## El modelo cliente-servidor.

El modelo más utilizado para el desarrollo de aplicaciones en red es el de cliente-servidor. Un servidor es un proceso que espera contactar con algún proceso cliente para darle algún tipo de servicio. Un proceso de comunicación típico siguiendo este modelo es el siguiente: (como ejemplo podemos imaginar un servidor guía telefónica, al que enviamos nombres de personas y responde con su teléfono)

1.El proceso servidor se pone en ejecución en algún computador. Inicializa sus estructuras de datos y se queda esperando a que algún cliente requiera sus servicios.

2.Un proceso es puesto en ejecución en el mismo computador o en otro de la red. En algún momento este proceso necesita acceder a la guía telefónica, se convierte en este

momento en proceso cliente, enviando una petición de servicio a través de la red hacia el servidor de guía telefónica y se queda esperando respuesta.

3.El servidor despierta, atiende la petición, responde a su cliente actual y se queda de nuevo esperando otro cliente.

Se puede pensar en dos modelos de servidor:

1.Servidor iterativo: Es el propuesto en el ejemplo anterior. Sólo se utiliza si la respuesta puede ser elaborada en un tiempo conocido y no muy largo. Si elaborar la respuesta puede durar demasiado tiempo, durante este tiempo puede llegar una petición de otro cliente, que no será atendida por el servidor hasta que no finalice la actual. En algunos problemas esta solución no se puede dar por buena.

2.Servidor concurrente: En lugar de servir las peticiones de los clientes directamente, el servidor crea un proceso cada vez que llega una nueva petición de servicio. Los nuevos procesos hijos se encargarán de servir directamente la petición y acabarán, mientras que el proceso padre siempre permanecerá escuchando.

Otra opción de diseño para nuestra aplicación será el tipo de comunicación que vamos a usar, con conexión o sin ella (stream socket (connection-oriented)/datagram socket (connectionless)). En las figuras 4 y 5 ofrecemos los esquemas del proceso que deben seguir un cliente y un servidor para realizar una comunicación sin conexión y con ella respectivamente.

Empezaremos por repasar las llamadas ofrecidas para la comunicación sin conexión por ser estas más sencillas:

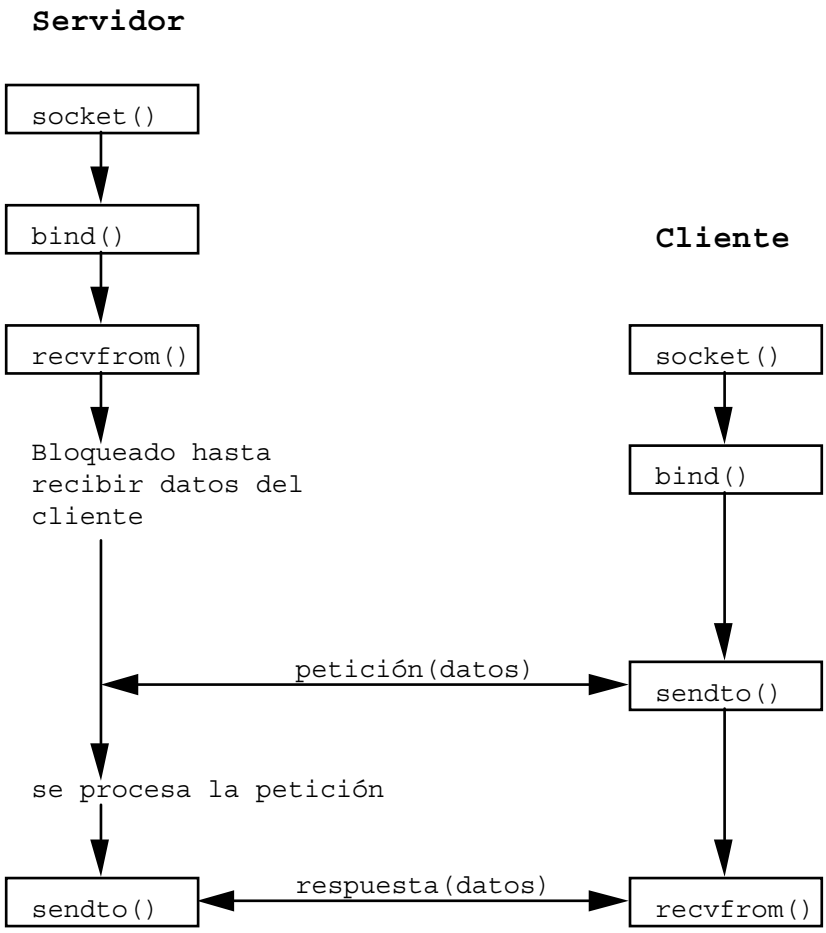



Figura 4. Llamadas al sistema para sockets en un protocolo no orientado a la conexión.

En primer lugar el servidor crea un socket (llamada *socket()*). En la creación, al socket no se le asigna nombre (dirección local) por lo que el resto de procesos no pueden referenciarlo, y por lo tanto no se pueden recibir mensajes en él.

Mediante la llamada *bind()* se asigna una dirección local al socket (para TCP/IP una dirección local es la dirección IP de la máquina más un número de port). Esta dirección deberá ser conocida por los clientes. Una vez configurado el socket, el servidor puede quedar esperando la recepción de un mensaje utilizando la llamada *recvfrom()*.

Por otro lado el cliente deberá igualmente crear un socket y asignarle una dirección de su máquina. Una vez hecho esto podrá enviar mensajes (peticiones de servicio) al servidor usando la llamada *sendto()* en la que especificará como dirección destino la del socket creado por el servidor.

La llamada *recvfrom()* proporcionará al servidor la dirección del cliente, que podrá ser usada por el primero para enviar la respuesta.

Si se desea una comunicación con conexión el proceso es algo más complejo debido a que antes de comenzar el envío y recepción de datos se debe establecer la conexión.

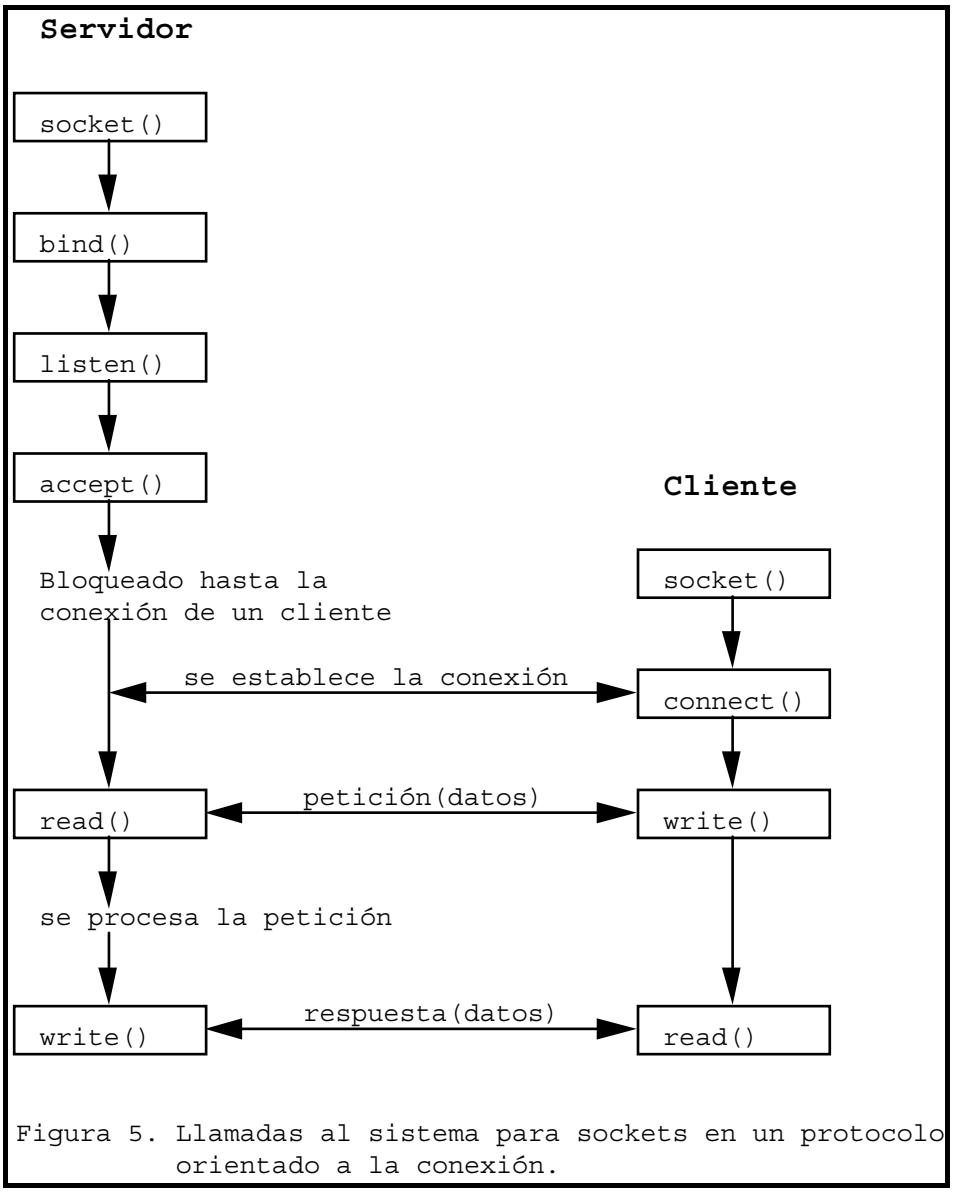



Figura 5. Llamadas al sistema para sockets en un protocolo orientado a la conexión.

El servidor creará inicialmente un extremo de la conexión pidiendo un socket (*socket()*) y dándole una dirección local (*bind()*). El servidor puede en algún momento recibir varias peticiones de conexión simultáneas por lo que se ofrece una llamada que permite especificar el número máximo de conexiones pendientes (*listen()*). Después queda esperando la petición de conexión por parte de algún cliente (*accept()*).

Por otro lado el cliente creará un socket, le dará una dirección local y lanzará una petición de conexión al servidor (*connect()*). Si el servidor está disponible (ha ejecutado *accept* y no hay peticiones anteriores encoladas) inmediatamente se desbloquean tanto cliente como servidor. En la parte del servidor se habrá creado un nuevo socket (cuyo identificador devuelve *accept()*) que es el que realmente está conectado con el cliente,

de forma que el original sigue sirviendo al servidor para atender nuevas peticiones de conexión.

Cliente y servidor se intercambiarán datos mediante *read()* y *write()* pudiendo finalmente cerrar la conexión mediante la llamada *close()*.

## Resumen de llamadas al sistema.

### Creación de socket.

Para crear un socket se usa la llamada al sistema *socket()* especificando la familia de protocolos que usará y el tipo de conexión.

```
int socket (int af, int type, int protocol);
sockfd = socket (af, type, protocol)
```

*af* especifica la familia de protocolos que se desea usar en la comunicación. En el fichero <*sys/socket.h*> se definen constantes para los más utilizados:

AF_UNIX para protocolos internos de UNIX.
AF_INET para protocolos Internet.
AF_NS para protocolos de Xerox NS.
AF_APPLETALK para protocolos con red Appletalk.

*type* indica el tipo de comunicación que se desea. Posibles tipos son: SOCK_STREAM, SOCK_DGRAM, SOCK_RAW, SOCK_SEQPACKET.

No todas las combinaciones son válidas, ya hemos comentado que la familia TCP/IP sólo soporta stream, datagram y raw sockets.

El tercer argumento sirve para especificar un protocolo específico. Si se pasa valor 0 el sistema elige el apropiado según los otros dos parámetros.

La función devuelve un entero que llamaremos socket descriptor y servirá para referenciar al socket en el resto de las llamadas (al igual que el file descriptor en el acceso a ficheros).

### Asignación de dirección local.

Como hemos visto la llamada *socket()* no asigna una dirección al socket creado. Automáticamente las llamadas *send()* y *connect()* asignan a nuestro socket la dirección IP de nuestra máquina y un número de port libre. Sin embargo hay casos en que

necesitaremos dar una dirección determinada a un socket. Todos los clientes deben conocer la dirección de su servidor para poder pedirle servicios, para ello el servidor creará un socket con una dirección IP y un puerto determinados, que hará públicos.

Para asignar una dirección determinada a un socket se usa la llamada *bind()*:

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

Donde *myaddr* corresponde a la dirección local que se asignará al socket *sockfd* y es una tira de bytes que será interpretada según el protocolo utilizado. En la familia TCP/IP una dirección está compuesta por una dirección IP de 32 bits y un número de port de 16 bits.

Todas las llamadas al sistema para trabajar en red son capaces de hacerlo con cualquier protocolo y siempre esperan un formato fijo de dirección. En el fichero <*sys/socket.h*> se define el struct *sockaddr* que es el tipo struct esperado por todas las llamadas:

```
struct sockaddr{
u_short sa_family; /*familia de direcciones a usar*/
char sa_data[14]; /*14 bytes para la dirección del protocolo dado*/
};
```

En particular y si estamos usando protocolos TCP/IP tenemos en <*netinet/in.h*> las siguientes estructuras que nos servirán para definir una dirección Internet:

```
struct in_addr{
u_long s_addr; /* 32 bits para la dirección IP*/
};
struct sockaddr_in{
short sin_family; /*AF_INET*/
u_short sin_port; /*16 bits para el número de port*/
struct in_addr sin_addr; /*32 bits para la dirección IP*/
char sin_zero[8]; /*no usados*/
};
```

Una vez definida la dirección sobre una variable del tipo *sockaddr_in* podremos pasarla a las funciones que manejan sockets haciendo un 'cast' del tipo *sockaddr* sobre esta variable:

```
struct sockaddr_in myaddr_in;
...
bind(sockfd, (struct sockaddr *) &myaddr_in, sizeof(myaddr_in));
```

Si queremos que los mensajes enviados a nuestro socket lleguen a nuestra

máquina la dirección IP de la llamada *bind()* debe ser la del host en el que estamos ejecutando el proceso. Por lo tanto en el campo *sin_addr* del struct *sockaddr_in* deberemos poner la dirección IP de nuestra máquina. Para conseguirla disponemos de varias funciones entre las que nos interesan las 2 siguientes:

```
int gethostname(char *hostname, int length)
```

que nos proporciona el nombre de nuestro host.

```
struct hostent *gethostbyname(char *name)
```

que dado el nombre de un host nos devuelve un struct que contiene entre otras cosas la dirección IP de dicho host. Estas funciones se especifican más detenidamente en el apartado "obtener información de la máquina".

### La llamada al sistema listen()

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

En "connection-oriented" permite al servidor definir el número máximo de peticiones de conexión pendientes de ser atendidas.

*qlength* es la longitud de la cola de peticiones para el socket. Normalmente el máximo valor soportado por el sistema operativo es 5.

### La llamada al sistema accept()

```
int accept (int sockfd, struct sockaddr *peer, int *addrlen);
newsock = accept (sockfd, peer, addrlen)
```

Una vez realizada la llamada *listen()*, todas las peticiones de conexión que lleguen al socket serán guardadas en su cola de peticiones. A continuación el proceso propietario del socket podrá atender la primera de ellas mediante la llamada *accept()*. Si al realizar esta llamada la cola de peticiones estaba vacía el proceso quedará bloqueado hasta la llegada de una nueva petición. Una vez ha llegado una petición el sistema crea un nuevo socket cuyo descriptor devuelve la llamada *accept()*, y que será el utilizado en la comunicación. Además *accept()* rellena el struct *peer* con la dirección del socket del cliente que pidió la conexión y *addrlen* con el tamaño en bytes de esa dirección. Estos datos podrán ser usados por el servidor para conocer la identidad de su cliente, aceptar o rechazar servicio.

**Estableciendo conexión.**

```
int connect (int sockfd, struct sockaddr *servaddr, int addrlen);
connect (sockfd, servaddr, addrlen)
```

La llamada *connect()* sirve al cliente para realizar una petición de conexión al servidor. Esta petición será inmediatamente atendida si el servidor estaba bloqueado en *accept()*, o será encolada si la cola no estaba vacía o el servidor no acepta conexión en ese momento.

*servaddr* es la dirección destino a la que queremos conectar el socket.

*addrlen* es un entero que indica la longitud en bytes de la dirección destino.

**Transferencia de datos.**

Cuando dos procesos ya han quedado "enganchados" por una conexión, pueden enviar-recibir datos, para ello se usan las llamadas *read* y *write*, además de otras nuevas como *send* y *recv*. Estas dos últimas añaden un argumento: *flags* que puede tomar un valor 0 o el de algunas constantes definidas en <*sys/socket.h*> para indicar algún tipo de envio concreto (datos "urgentes"...). Estas cuatro llamadas sólo trabajan con sockets "conectados" ya que no permiten especificar dirección destino.

**La llamada al sistema read()**

```
int read(int sockfd, char *buffer, int length);
read(sockfd, buffer, length)
```

*buffer* indica la dirección de memoria donde se cargarán los datos.

*length* máximo número de bytes a leer.

**La llamada al sistema write()**

```
int write(int sockfd, char *buffer, int length);
write(sockfd, buffer, length)
```

*buffer* indica la dirección donde se encuentran los datos a enviar.

*length* número de bytes a enviar.

### La llamada al sistema send()

```
int send(int sockfd, char *message, int length, int flags);
send(sockfd, message, length, flags)
```

*message* indica la dirección donde se encuentran los datos a enviar..

*length* número de bytes a ser enviados.

*flags* controla la transmisión. Puede tener un valor cero o bien OR'es entre las constantes:

MSG_OOB para enviar o recibir datos out-of-banda (datos "urgentes").
MSG_DONTROUTE para enviar sin utilizar las tablas de routing (para administradores de red y procesos privilegiados).

### La llamada al sistema recv()

```
int recv(int sockfd, char *buffer, int length, int flags);
recv(sockfd, buffer, length, flags)
```

*buffer* da la dirección en memoria donde se colocará el mensaje.

*length* longitud del buffer.

*flags* controla la recepción (como en *send()*).

### La llamada al sistema close()

```
int close(int sockfd);
close(sockfd)
```

Es la misma llamada que utilizamos para cerrar ficheros en UNIX. Si el protocolo que estamos usando es "seguro" (TCP) cuando hagamos close, el sistema nos asegurará que antes de destruir el socket todos los datos que han sido enviados serán recibidos en el otro extremo. El usuario puede pedir que no se haga esta comprobación final mediante la llamada *shutdown()*:

**La llamada al sistema shutdown()**

```
int shutdown(int sockfd, int howto);
```

*howto* = 0 indica que los datos que se puedan recibir ya no interesan.

*howto* = 1 indica que los datos que aún no se han enviado no se envien.

*howto* = 2 indica que el socket deja de trabajar en el acto haya lo que haya pendiente.

**La llamada al sistema sendto()**

```
int sendto(int sockfd, char *message, int length, int flags,
          struct sockaddr *destaddr, int addrlen);
sendto(sockfd, message, length, flags, destaddr, addrlen)
```

*sockfd*, *message*, *length* y *flags* son como en la llamada *send*.

*destaddr* especifica la dirección destino.

*addrlen* la longitud en bytes de la dirección destino.

**La llamada al sistema recvfrom()**

```
int recvfrom(int sockfd, char *message, int length, int flags, struct sockaddr
*fromaddr, int *addrlen);
recvfrom(sockfd, message, length, flags, fromaddr, addrlen)
```

*sockfd*, *message*, *length* y *flags* son como en la llamada *recv*.

El struct *fromaddr* es rellenado por la función con la dirección del emisor del mensaje recibido y *addrlen* con la longitud en bytes de esa dirección. *addrlen* debe estar inicializado antes de la llamada con el sizeof(fromaddr).

**Llamadas de conversión.**

Cada arquitectura puede definir un orden de almacenamiento distinto para los bytes de una variable. Los 4 bytes de un entero por ejemplo pueden ser almacenados comenzando por el byte más significativo o al contrario. Si una máquina envía un entero a otra con distinto orden de almacenamiento el entero será interpretado de forma errónea. Para solucionar este problema el protocolo de red especifica un orden, y

todas las máquinas que lean o escriban enteros a través de esa red reescribirán sus enteros en el formato de red. En el fichero <*netinet/in.h*> aparecen las cabeceras de varias funciones proporcionadas para realizar estas conversiones.

```
u_long htonl(u_long hostlong); /*host to network long*/
```

Convierte enteros "long" (32 bits) a la ordenación de bytes de red estandar.

```
u_short htons(u_short hostshort); /*host to network short*/
```

Convierte enteros "short" (16 bits) a la ordenación de bytes de red estandar.

```
u_long ntohl(u_long netlong); /*network to host long*/
```

Inverso a *htonl*.

```
u_short ntohs(u_short netshort); /*network to host short*/
```

Inverso a *htons*.

**Rutinas de manipulación de direcciones IP.**

Como muchos programas traducen direcciones IP de 32 bits a su correspondiente notación decimal, 4BSD incluye funciones para realizar esta traducción.

```
#include <arpa/inet.h>
unsigned long inet_addr(char *ptr);
char *inet_ntoa(struct in_addr inaddr);
```

*inet_addr* convierte una cadena en notación decimal a una dirección Internet de 32 bits.

*inet_ntoa* realiza la conversión inversa.

**Operaciones especiales para manipular cadenas.**

Se disponen de funciones para tratar cadenas que no son estandar de C, es decir no están acabadas por un byte NULL. Serán cadenas que puedan tener carácteres NULL intercalados.

```
bcopy(char *src, char *dest, int nbytes);
```

Mueve *nbytes* de la cadena fuente a la destino.

```
bzero(char *dest, int nbytes);
```

Escribe *nbytes* NULL en la cadena destino.

```
int bcmp(char *ptrl, char *ptr2, int nbytes);
```

Compara dos cadenas de bytes arbitrarias. Devuelve 0 si las dos cadenas son idénticas y != 0 en caso contrario.

**Obtener información de la máquina.**

Existen funciones de librería que nos permiten obtener información sobre una máquina dada o su dirección IP, consultando ficheros o bases de datos internas.

```
int gethostname(char *hostname, int length);
```

Rellena *hostname* con el nombre de la máquina en la que se está ejecutando el proceso que la llama, *length* con su longitud, y devuelve 0 si todo va bien y -1 en caso de error.

Hay otras dos funciones para obtener información sobre una máquina dándole su nombre:

```
struct hostent *gethostbyname(char *name);
```

Devuelve un puntero a una estructura donde hay información de: nombre oficial del host, lista de alias del host, tipo de dirección del host (por ejemplo si es IP), longitud de la dirección y una lista de una o más direcciones para el host. El struct hostent está definido en <*netdb.h*>:

```
#define h_addr h_addr_list[0];
struct hostent{
char *h_name; /*nombre oficial del host*/
char **h_aliases; /*lista de alias*/
int h_addrtype; /*tipo de las direcciones de hosts (ej.: AF_INET)*/
int h_length; /*longitud de la dirección*/
char **h_addr_list; /*lista de direcciones, acabada en NULL*/
};
```

En las máquinas que trabajaremos normalmente el campo *h_addrtype* será AF_INET y sólo el primer elemento de la lista de direcciones estará definido con la dirección Internet de nuestra máquina.

```
struct hostent *gethostbyaddr(const char *addr, int len, int type);
```

Esta función espera en *addr* un puntero a una secuencia de bytes que contienen la dirección del host, en *len* el tamaño de la dirección y en *type* su tipo.

**Obtener información sobre los protocolos.**

Existen otras funciones de librería que nos permiten obtener información sobre los protocolos disponibles en la máquina, consultando bases de datos internas. Cada protocolo posee un nombre oficial, alias registrados y un número oficial. Están definidos en */etc/protocols*.

```
struct protoent *getprotobyname(const char *name);
```

Devuelve un puntero a una estructura que tiene los campos con el nombre oficial del protocolo, una lista de alias y un valor entero único asignado al protocolo:

```
struct protoent{
char *p_name; /*nombre oficial del protocolo*/
char **p_aliases; /*lista de alias*/
int p_proto; /*número del protocolo*/
};
```

*name* es un puntero a una cadena ASCII que contiene el nombre del protocolo para el cual se desea información.

```
struct protoent *getprotobynumber(int proto);
```

Permite acceder a la información del protocolo utilizando su número.

**Obtener información sobre los servicios de la red.**

Existen ciertos números de ports reservados para servicios como FTP o TELNET. Se dispone así de ciertas funciones de librería para obtener información sobre los servicios y los puertos que utilizan. Están definidos en */etc/services*.

```
struct servent *getservbyname(const char *name,const char *proto);
```

Devuelve un puntero a una estructura que contiene los campos para el nombre del servicio, lista de alias, identificación del protocolo con el cual el servicio es usado y un entero que da el número de port para el servicio:

```
struct servent{
char *s_name; /*nombre oficial del servicio*/
char **s_aliases; /*lista de alias*/
```

```
int s_port; /*número del puerto*/
char *s_proto; /*protocolo a usar*/
};
```

*name* da la dirección de una cadena que contiene el nombre del servicio deseado.

*proto* da el protocolo con el cuál el servicio es utilizado.

Como alternativa a esta función está:

```
struct servent *getservbyport(int port, const char *proto);
```

**Obtener información sobre la red.**

Existen otras funciones de librería que nos permiten obtener información sobre la red que estamos utilizando, consultando bases de datos internas (fichero */etc/networks*).

```
struct netent *getnetbyname(const char *name);
```

Devuelve un puntero a una estructura que contiene los campos para el nombre oficial de la red, una lista de alias registrados, un entero que indica cuál es el tipo de la dirección y una dirección de red de 32 bits (dirección IP con la parte del host a 0):

```
struct netent{
char *n_name; /*nombre oficial de la red*/
char **n_aliases; /*lista de alias*/
int n_addrtype; /*tipo de la dirección de la red*/
int n_net; /*número de la red*/
};
```

*name* es un puntero a una cadena que contiene el nombre de la red de la cual queremos información.

```
struct netent *getnetbyaddr(int net, int type);
```

Permite buscar información sobre la red mandándole la dirección.

*net* es la dirección de la red en 32 bits.

*type* un entero que especifica el tipo de la red.