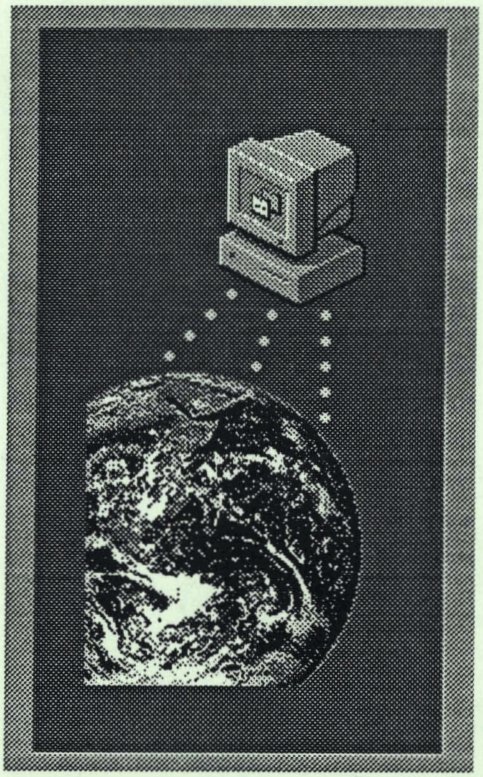
ESCUELA UNIVERSITARIA
DE INGENIERÍA TÉCNICA DE TELECOMUNICACIÓN
Departamento de Electrónica, Telemática y Automática

Introducción a la programación en red: Sockets y Windows Sockects



Miguel Angel Quintana Suárez
Miguel Fecanin Araujo

Estos apuntes han sido editados e impresos por el Servicio
de Reprografía de la Universidad de Las Palmas de Gran Canaria.
Campus Universitario de Tafira. Edificio Departamental de Ingenierías
Las Palmas de Gran Canaria (35017). Islas Canarias. España.
Abril de 1996

Depósito legal G.C. 342-1996
ISBN 84-87526-39-X
Modelo de máquina Docutec Ranx Xerox.
Nº Serie 1104516609

Indice

Capítulo 5 *Algoritmos en el diseño de aplicaciones cliente*

Capítulo 6 *Servidores*

Capítulo 7 *El mundo WINDOWS*

Capítulo 8 *Especificación Windows Sockets*

Capítulo 9 *WINDOWS SOCKETS ver. 2.0*

Bibliografía

Introducción.

La especificación Windows Sockets (W.S.) define un interfaz de programación en red para Windows, que está basada en la programación con sockets popularizada por la Universidad de Berkeley en California.

Esta especificación en su versión 1.1, contiene tanto rutinas referidas a los sockets de Berkeley como otras específicas creadas con la intención de aprovechar y hacer uso de la programación en Windows; programación ésta que se basa en los mensajes.

La base para las aplicaciones de red en la versión 4.3 de UNIX BSD es una abstracción referida a los sockets. Un socket, o más bien, el concepto de un socket, se puede comparar con el de un TSAP (Transport Service Access Point) o punto de acceso al servicio de transporte dentro del modelo de referencia OSI[#] de ISO[##] . Esta es la definición más acertada de un socket, pero no la más intuitiva. Un socket debe entenderse como un punto final de comunicación. Por ejemplo, en una comunicación telefónica, el socket podría entenderse como el teléfono. De todas formas, no es necesario extenderse mucho con la definición de un socket, ya que tampoco hay mucho que decir al respecto.

Si bien la interfase a los programas de aplicación basada en el paradigma de los sockets fue creada en principio bajo un entorno multiproceso UNIX, ahora se ha extendido su concepción a un entorno multitarea como es Windows. Además, debido a su gran difusión y aceptación, W.S. se ha convertido en el estándar "de facto" para la interconexión de redes bajo TCP/IP.

El propósito de la interfase de los sockets no es más que proporcionar a las aplicaciones un modo fácil y sencillo de acceso a los servicios de la capa de transporte. Además, los sockets pretenden esconder los diferentes detalles de los diferentes protocolos o capas de transporte que puedan estar disponibles en la máquina. Es decir, en nuestra capa

[#] Open System Interconection

[##] International Standards Organization

de transporte podremos tener los protocolos TCP y UDP correspondientes al conjunto TCP/IP y además tener también acceso al protocolo de transporte de X.25, así como otros muchos. Gracias a los sockets podremos acceder a cualquiera de ellos y diseñar aplicaciones de red sin necesidad de conocer las particularidades de cada uno.

La especificación W.S. en su versión 1.1 contempla hasta 19 familias diferentes de protocolos, entre las que cabe destacar, Xerox NS, OSI de ISO, IBM SNA, DECnet, AppleTalk, y otras más. De todas formas, los vendedores de paquetes de protocolos tan sólo han implementado la especificación W.S. para el conjunto TCP/IP. Debido a esta situación se ha dedicado un capítulo entero a la comprensión de este conjunto de protocolos sobre el cual se apoya la cada vez más famosa interconexión de redes llamada Internet, además de por el actual crecimiento y aceptación del mundo Internet.

Windows Sockets entonces debe ser interpretado de cara a los programadores como un conjunto de funciones que les van ha facilitar el diseño de aplicaciones que hagan uso de recursos a través de una red. Actualmente, la documentación existente respecto a esta especificación no es más que un manual de referencia sobre las funciones que posee W.S. Para poder obtener una idea de la filosofía y de la metodología en la programación con sockets es necesario recurrir a bibliografía sobre el S.O. UNIX ya que es en este entorno donde se han estudiado ampliamente.

La programación en un entorno UNIX responde a fórmulas clásicas tales como la programación estructurada, por procedimientos y secuencial mientras que en Windows se introducen conceptos nuevos, como la programación no secuencial y gobernada por mensajes, así como programación orientada a objetos. Es necesario por tanto que la filosofía de programación bajo Windows también sea tema de análisis en los siguientes capítulos debido a su gran importancia para la comprensión de las funciones y metodología de programación con sockets bajo Windows.

El diseño de programas que interactuan a través de una red es bastante diferente a los programas que estamos acostumbrados a ver y diseñar. Es necesario establecer unas reglas para la comunicación entre las aplicaciones, es decir, definir un protocolo de aplicación, además de sincronizar las comunicaciones para que no existan problemas. Un

modelo muy aceptado hoy en día y que se analizará también será el paradigma Cliente/Servidor. Este modelo será la base de diseño de las aplicaciones de red. No confundamos este modelo con el protocolo de aplicación. El modelo será el encargado de sincronizar la comunicación y el protocolo dictará las reglas de lo que se dicen ambas aplicaciones a través de la red.

Por tanto, empezaremos en nuestro primer capítulo con una breve introducción al conjunto de protocolos TCP/IP e Internet, siguiendo con una introducción al modelo Cliente/Servidor. Además se analizarán con un poco más de detalle las diferentes fórmulas existentes a la hora de diseñar aplicaciones de red utilizando el paradigma Cliente/Servidor. Una vez entendido el diseño de aplicaciones Cliente/Servidor se da una introducción a la programación bajo Windows con el fin de adquirir los conocimientos necesarios para poder explicar la funcionalidad de la especificación W.S. Y ya por último analizar de forma más detallada las funciones de la especificación W.S. y su posible utilización en el diseño de programas.

A la hora de realizar la secuencia de capítulos ha existido un problema sobre cual situar primero; si analizar las funciones de la especificación W.S. primero y después introducir los algoritmos de las aplicaciones Cliente/Servidor o a la inversa. Se ha preferido introducir primero las ideas básicas del diseño de aplicaciones Cliente/Servidor aún no teniendo idea de las funciones de W.S. dado que de lo contrario se perdería el rumbo de lo que realmente se intenta explicar. Es decir, comentar una por una las diferentes funciones de la especificación no sería muy didáctico. Por tanto se adoptó la solución de ir comentando algunas de las funciones de la especificación a medida que se hagan necesarias.

Capítulo 1

Introducción a Internet y TCP/IP

1.1. Introducción

Dado que el conjunto de protocolos comúnmente conocidos como **TCP/IP**[#] es el soporte de Windows Sockets es necesario dar una breve introducción sobre este tema.

Los protocolos Internet **TCP/IP** están organizados en cuatro capas conceptuales. En la figura se muestra una comparación con las capas del Modelo de Referencia para la Interconexión de Sistemas Abiertos (**OSI/RM**).

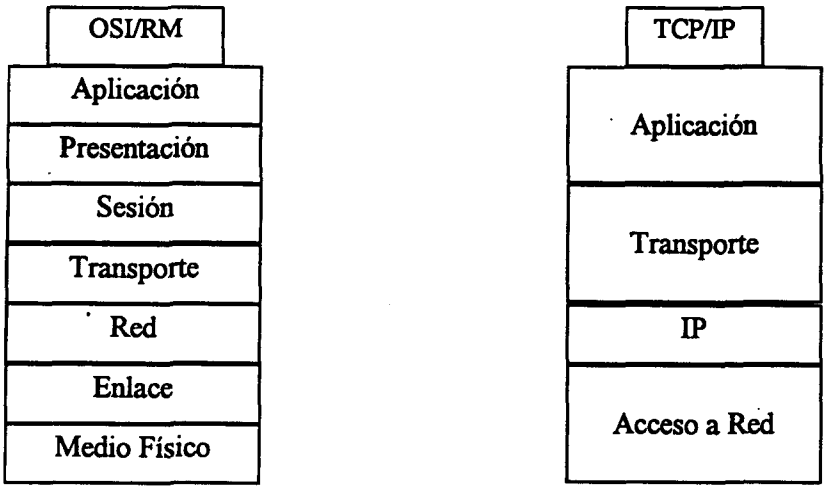


Figura 1.1.

En la capa de aplicación es donde están ubicados nuestros programas y todos los procesos de usuario. Estos procesos hacen uso de la capa inmediata inferior para solicitar los servicios de la capa de transporte. La capa de transporte en el modelo TCP/IP se divide

[#] Transmission Control Potocol/ Internet Protocol

en dos protocolos bien diferenciados, el TCP y el UDP[##] . El primero se encarga de suministrar un canal de comunicación orientado a conexión y el segundo se encarga de establecer una comunicación en modo datagrama. Y debajo de la capa de transporte aparece la capa de red IP.

1.2. Una visión global de las capas

Empecemos desde abajo en esta estructura de capas. En la capa de red (capa IP en el modelo TCP/IP) se deberá elegir entre un diseño orientado a conexión o no. En el caso de comunicaciones orientadas a conexión, la capa de red establece lo que normalmente se denomina un *circuito virtual* entre ambos extremos que desean conectarse. La idea general de circuito virtual es que se establece un camino entre ambos extremos que será el mismo durante toda la comunicación. Es decir, en el momento que se establece el circuito virtual, todos los datos serán encaminados por la misma ruta. Esto no implica que cada vez que se establezca el circuito virtual entre los mismos extremos la ruta sea la misma. En algunos casos como en *X.25* la ruta siempre será la misma debido a que las tablas de enrutamiento de los nodos de la red son fijas o estáticas, pero en otras redes dependiendo de la congestión de los nodos y de los algoritmos de encaminamiento, unas veces se establecerá el circuito virtual por una ruta y otra vez por otra.

Una vez establecido el circuito virtual, todos los paquetes serán encaminados por la misma ruta. Por el contrario, en el modo datagrama, cada datagrama o paquete se lanza a la red y se encaminan individualmente. Es decir, el primer datagrama puede cursar una ruta y el segundo datagrama dirigirse por otra ruta diferente. Este sistema puede provocar que los paquetes lleguen desordenados, por lo tanto, nuestra aplicación debe tener en cuenta este fenómeno y ordenar los paquetes recibidos para un correcto funcionamiento. Además, el modo datagrama no es fiable ni seguro en cuanto a transmisión se refiere, es decir, no proporciona mecanismos para detectar la pérdida de paquetes en la red.

[##] User Datagram Protocol

La capa IP del modelo TCP/IP es una capa de red no orientada a conexión. Un ejemplo de una capa de red orientada a conexión sería la capa de red del protocolo *X.25*. Por tanto, y haciendo un símil con el servicio de correos, la capa IP proporciona un servicio bastante inseguro. No asegura que los paquetes lleguen a su destino ni que lleguen en orden de emisión. Por lo tanto, la capa de red IP es bastante simple en cuanto al servicio que proporciona a la capa de transporte (TCP/UDP). Son estos dos protocolos y en concreto el TCP el encargado de proporcionar a la capa de aplicación un servicio de comunicación orientado a conexión a partir de una capa de red no orientada a conexión. Es decir, a partir de un servicio de red en modo datagrama hacer de intermediario entre la aplicación y éste para ofrecer una comunicación segura y fiable.

Como se puede advertir ya, la capa de red IP es relativamente sencilla comparada con una capa de red orientada a conexión y toda la complejidad se sitúa en la capa de transporte, más concretamente en el protocolo de transporte TCP.

Por tanto el funcionamiento sería así. En la capa de transporte el protocolo TCP aceptaría mensajes de la capa de aplicación; mensajes de longitud arbitraria que deberá separarlos en paquetes que no excedan de 64K octetos. Estos paquetes se pasan a la capa de red IP la cual los transmite como datagramas a lo largo de la red hasta su destino. Como la capa de red no asegura que los datagramas lleguen a su destino, es tarea del protocolo TCP el utilizar temporizadores y retransmitir todos aquellos paquetes que sean necesarios. Por tanto, a cada datagrama se le da un tiempo de vida tras el cual si no se recibe un acuse de recibo por parte del otro extremo, se retransmite otra vez. De esta forma, el protocolo TCP proporciona una transmisión segura a la capa de aplicación. Además también el protocolo TCP debe tener en cuenta que los paquetes pueden ser entregados en desorden y por tanto además de reesamblar los paquetes recibidos en mensajes, debe antes ordenar los paquetes recibidos según la secuencia correcta. También, debido al mecanismo de retransmisión de paquetes haciendo uso de mecanismos de ***timeout*** puede ocurrir que se reciban en el destino paquetes repetidos. Es por tanto tarea del protocolo TCP el detectar los paquetes repetidos y descartarlos.

1.3. Por qué una capa IP no orientada a conexión

Veamos ahora por qué se decidió diseñar una capa de red no orientada a conexión y dejar todo el "*trabajo*" a la capa de transporte. En un principio la capa de red de ARPANET (IP) era orientada a conexión, se suponía que los servicios que proporcionaba la subred eran completamente seguros. Se diseñó por tanto un protocolo de transporte NCP[#] con la idea de una subred perfecta. Únicamente se limitaba a pasar las TPDU[##] o los mensajes (cuyo tamaño máximo eran de 64k) a la capa de red y la capa de red entregaría estas TPDU en el destino de forma ordenada. Pero con el tiempo ARPANET evolucionó hasta convertirse en la interconexión de redes ARPA, en las que se incluían varias redes LAN, una subred de transmisión de paquetes por radio y varios canales de satélite. Cada una de estas redes ofrecían un servicio de red no muy seguro lo cual hizo que la fiabilidad extremo a extremo disminuyese. Por este motivo se decidió diseñar una capa de red no orientada a conexión y pasar todos los mecanismos de fiabilidad y seguridad a la capa de transporte, de esta forma aunque los datos se encaminasen por redes cuya capa de red fuese insegura no sucedería nada ya que la seguridad la proporciona la capa de transporte.

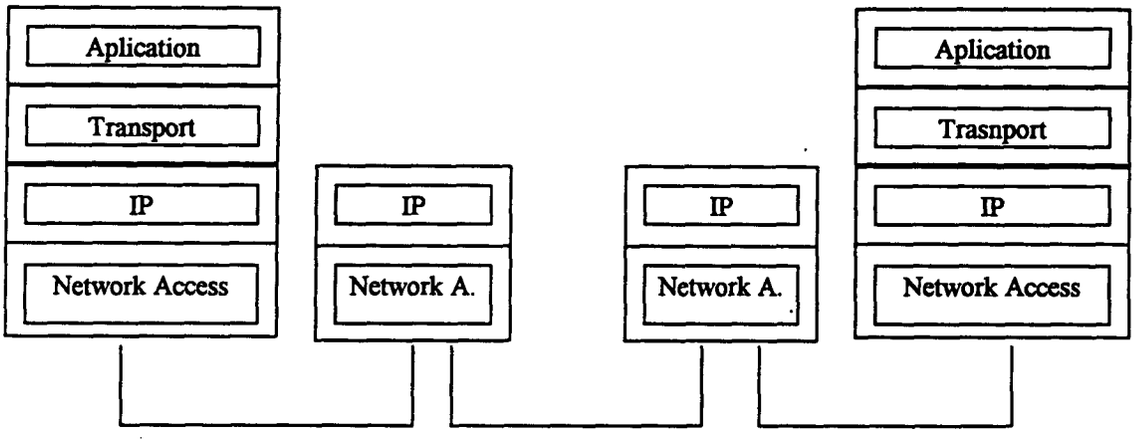


Figura 1.2.

Por ejemplo, el mecanismo que utiliza el protocolo TCP (capa de trasnporte) para detectar que la carta se recibió, utilizando el símil de correos, es el siguiente: sabremos que la carta llegó a su destino si exigimos recibir contestación o a lo que antes llamábamos acuse de recibo. Si esta contestación

[#] Net Control Protocol

[##] Transport Protocol Data Unit

no la recibíamos dentro de un plazo determinado (*timeout*) daríamos la carta por perdida y la enviaríamos otra vez.

1.4. El protocolo *UDP*

Hasta el momento nos hemos referido bastante al protocolo TCP situado en la capa de transporte. Pero en esta capa existe otro protocolo denominado UDP (User Datagram Protocol). Es necesario para cierto tipo de aplicaciones utilizar un modo de conexión por datagramas. La comunicación por datagramas es muy parecido al sistema de correos. Cada datagrama se puede asociar con la idea de una carta. Podemos enviar una carta y a menos que nos la rechacen en la oficina (porque el sello no es correcto o por otra razón) ya no sabremos si la carta a alcanzado su destino.

Además, siguiendo el ejemplo de correos, podemos enviar varias cartas y que éstas lleguen desordenadas al destino. Es por ello que normalmente se les ponga la fecha para que en la recepción se puedan ordenar. Algo muy parecido ocurre en una red por datagramas. Normalmente enviaremos un datagrama y no se nos informará de ningún error a no ser que el error se haya producido en el host local. Si el datagrama por cualquier motivo no llega a su destino no tendremos forma alguna de advertirlo. Además se debe dotar a los datagramas de un número de secuencia para poder reordenarlos en el destino.

1.5. Qué es Internet

Otro concepto que aparece mucho es el de Internet. Pues bien, Internet es una colección de redes, incluyendo a la Arpanet, NFSnet, redes regionales como NYsernet, redes locales ubicadas en un gran número de universidades e instituciones de investigación, así como otro gran número de redes militares. El término Internet por tanto se aplica a todo este conjunto de redes.

Figura 1.3.

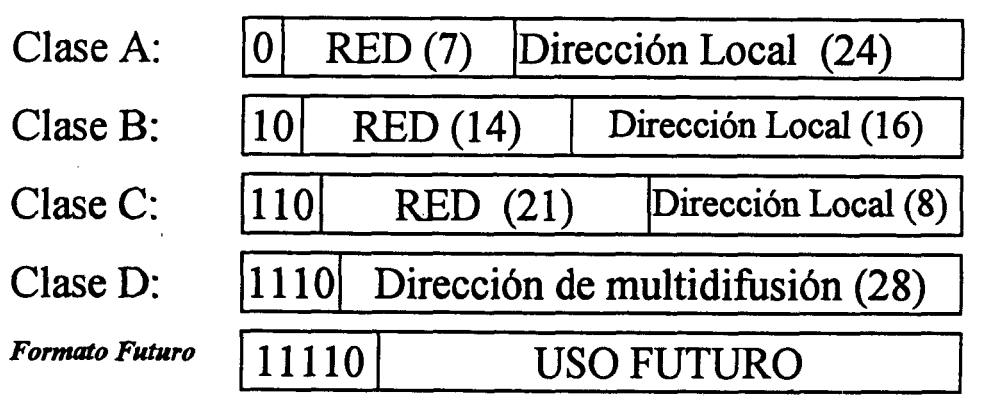
1.6. Direccionamiento a través de múltiples redes

En este lío de redes interconectadas entre sí, para poder transmitir "algo" a otro ordenador es normalmente necesario atravesar múltiples redes. Estas redes por lo general están conectadas mediante *gateways*. Un ordenador tiene que conocer por tanto una dirección, la dirección Internet del destino. Una dirección Internet se parece a 193.145.141.34. Esta es la notación con puntos de direcciones Internet o como se llama en inglés *dotted address*. Es un conjunto de 32-bits en el cual va definida la dirección de un ordenador. Cada número decimal representa un octeto u ocho bits. La palabra byte no se usa debido a que existen ordenadores cuyo tamaño de un byte es distinto de 8 bits. Centrándonos en la dirección antes dada, **193.145.141** por ejemplo, es un número de red que es asignado por una autoridad central a la Facultad de Telecomunicaciones en la Universidad de Las Palmas. Después se utiliza el siguiente byte para especificar un ordenador en concreto dentro de esa red. Así, el último octeto permite la identificación de hasta 254 ordenadores conectados a esa red y la dirección 193.145.141.**34** se referirá al

servidor Neumann en la Escuela de Telecomunicaciones en la Universidad de Las Palmas de Gran Canaria.

Esto ha sido un ejemplo de como se direcciona a un ordenador en Internet. Lo que está claro es que una dirección Internet se compone del par (red,ordenador). Es decir, primero se identifica una red y después un ordenador en concreto dentro de esa red. El problema viene a la hora de decidir cuantos bits codifican la red y cuantos el ordenador dentro de esa red. Puede suceder que sea necesario más bits para codificar un ordenador dentro de una red debido a que la red soporta muchos ordenadores o bien que no sean necesarios tantos bits para codificar un ordenador dentro de una red. Se han definido por tanto tres clases de direccionamientos bien diferenciados. La clase A se identifica por tener el primer número de la dirección Internet dentro del rango de 1 a 126. La clase B se identifica por tener los dos primeros números de su dirección Internet desde 128.1 hasta 191.254. Por último la clase C se identifica por ir sus tres primeros números de su dirección Internet desde 192.1.1 hasta 223.254.254. Como se puede advertir, la clase A utiliza el primer octeto para identificar a la red y los 24 bits restantes para codificar el ordenador dentro de esa red. La clase B utiliza los dos primeros octetos para identificar la red y los restantes 2 octetos siguientes para identificar al ordenador. Y por último la clase C utiliza 3 octetos para identificar la red y uno para referenciar a un ordenador dentro de esa red. Como se puede advertir se han omitido las direcciones que empiezan por 127 debido a que estas direcciones son usadas para propósitos especiales por algunos sistemas

En la siguiente figura se puede observar un ejemplo de tres redes diferentes conectadas mediante *gateways*. Notar que las redes tienen diferente clase de direccionamiento (clase A,...)



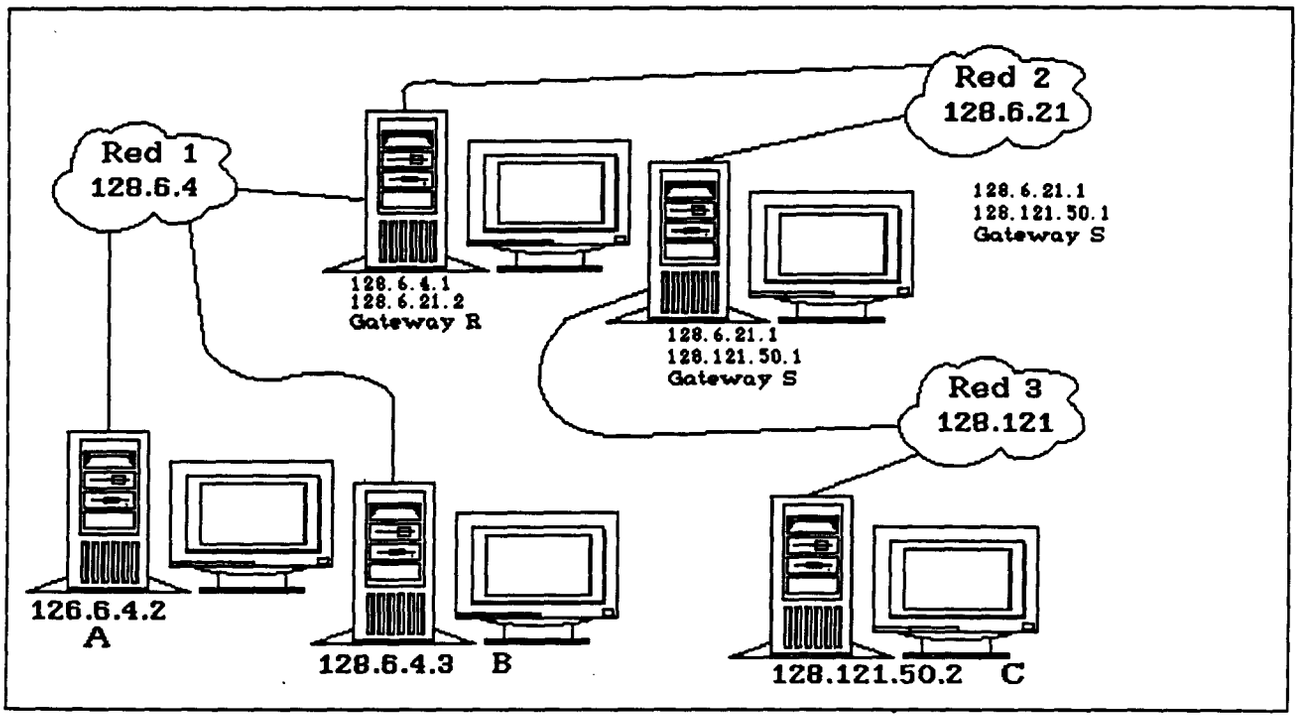


Figura 1.4.

1.7. El Sistema de dominios

El sistema de domino de nombres, el DNS[#] , es el servicio más importante de Internet. Es un componente esencial dentro de cualquier implementación de la capa IP.

El software de red generalmente se ve necesitado de una dirección Internet de 32-bits para poder abrir una conexión o enviar un paquete. Sin embargo los usuarios prefieren dar el nombre del ordenador antes que su número de Internet. De esta forma, existe una base de datos que permite al software buscar dado un nombre su dirección Internet. Cuando Internet era relativamente pequeña, este método era fácil y poco costoso de mantener en cada ordenador. Cada sistema tendría un fichero en el que se almacenarían todos los sistemas restantes dando su nombre y su dirección Internet. Este fichero normalmente estaba localizado en el subdirectorio *etc* y tenía el nombre host (*/etc/host*). Ahora debido al gran desarrollo y expansión de Internet, existen muchos sistemas como para que esta solución sea aceptable y práctica. Sería necesario dotar a cada sistema de un fichero

[#] Domain Name Service

/etc/host de varios megas para poder almacenar toda la información. Además, la búsqueda lineal dentro de este fichero para localizar un determinado host sería muy lenta y poco efectiva. Por tanto, se requería buscar una solución. Así, se decidió sustituir estos ficheros por un conjunto de servidores de nombres, DNS, que tenían la pista de los nombres junto con su correspondientes direcciones Internet. En este empeño se utilizaron varios servidores y no un único servidor central; se procedió por tanto así, a una descentralización y a una distribución de la información con todas las ventajas que esto significaba.

Actualmente existen muchas instituciones conectadas a Internet y sería impracticable para ellas notificar a una autoridad central la instalación o supresión de nuevos sistemas en la red. Es por ello, que los servidores de nombres forman un árbol (estructura jerárquica), correspondiendo a una estructura institucional. Veamos ahora un ejemplo típico en el siguiente nombre **neumann.teleco.ulpgc.es**. En este caso se trata del ordenador servidor en el laboratorio neumann en la Escuela de Telecomunicaciones. Para poder encontrar su dirección Internet, se debería en un principio consultar 4 servidores diferentes. Primero, se preguntaría en un servidor central (llamado *root*) dónde se encuentra el servidor ***es***. ***es*** por su parte, es un servidor que tiene la pista de todas las instituciones de España. Normalmente el *root* tiene la posibilidad de suministrar los nombres y direcciones Internet de varios servidores ***es***. La razón de la existencia de varios servidores a un mismo nivel en el árbol es lógica, ya que de esta manera se hace frente a la posibilidad de que uno de ellos pueda fallar. Bien, una vez conectado a **es** se le preguntaría donde está el servidor ***ulpgc***. Otra vez, el servidor ***es*** reportaría nombres y direcciones Internet de varios servidores ***ulpgc***. Generalmente no todos los servidores dados por ***es*** serán ***ulpgc***, para permitir la posibilidad de un fallo general en los servidores ***ulpgc***. Después preguntaríamos a ***ulpgc*** donde encontrar el servidor ***teleco*** y finalmente se le preguntaría a ***teleco*** sobre ***neumann***. El resultado final reportado por el último servidor (teleco) sería la dirección Internet correspondiente a **neumann.teleco.ulpgc.es**. Cada uno de éstos niveles se refiere a un "subdominio". El nombre entero, *neumann.teleco.ulpgc.es*, es llamado un nombre de dominio "*domain name*".

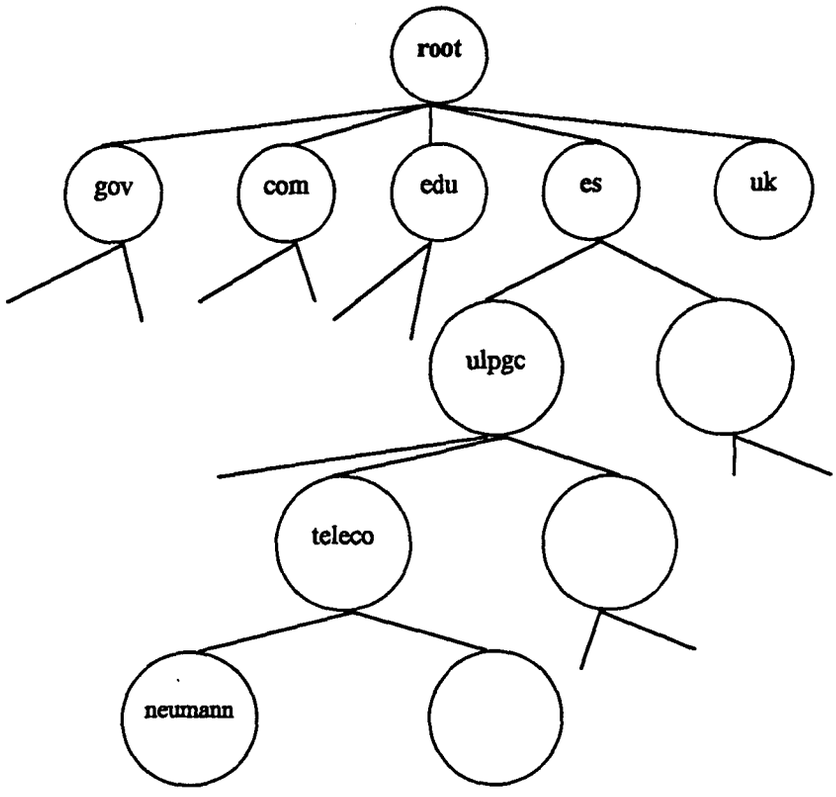


Figura 1.5.

Desde ahora nos referiremos a direcciones DNS y direcciones IP. La primera corresponde a un nombre de Host y la segunda corresponde a la dirección en formato numérico y punteada como 193.145.141.34.

Afortunadamente, no necesitamos ir a través de todo este tramado en la mayoría de los casos. Lo primero de todo es que el *root* tiene la información de los dominios altos y por tanto una petición al *root* como la anterior ya nos devolvería información sobre ***ulpgc*** sin necesidad de preguntar a ***es***, ya que el *root* suele tener esa información. En segundo lugar, el software generalmente suele recordar las respuestas que obtuvo anteriormente. Así, si anteriormente preguntamos por teleco.ulpgc.es, nuestro software recordará donde encontrar servidores para *teleco.ulpgc.es*, *ulpgc.es* y *es*.

Por tanto, gracias a este sistema llamado DNS, se logra una infraestructura distribuida. Es decir, la información no se encuentra en un super ordenador central al que

todo el mundo querría acceder provocando saturaciones, sino que se encuentra distribuida a lo largo de toda la red por diferentes servidores.

Finalmente, pasamos a dar una relación de documentos relacionados con el DNS para poder profundizar en el tema. Paul Mockapetris es el autor del DNS -- del diseño del protocolo, de la metodología, etc...

RFC-1035 P. Mockapetris, "*Domain names - implementation and specification*", 11/01/1987. (Pages=55) (Format=.txt) (Obsoletes RFC0973) (STD 13) (Updated by RFC1348)

RFC-1034 P. Mockapetris, "*Domain names - concepts and facilities*", 11/01/1987. (Pages=55) (Format=.txt) (Obsoletes RFC0973) (STD 13) (Updated by RFC1101)

RFC-1033 M. Lottor, "*Domain administrators operations guide*", 11/01/1987. (Pages=22) (Format=.txt)

RFC-1032 M. Stahl, "*Domain administrators guide*", 11/01/1987. (Pages=14) (Format=.txt)

Sin el DNS, Internet probablemente no hubiese sobrevivido dado las dimensiones que ha ido adquiriendo esta red.

Esto ha sido una visión global a lo que se conoce bajo el nombre de TCP/IP. Para profundizar en el tema se recomienda la siguiente bibliografía:

REDES DE ORDENADORES, Andrew S.Tanenbaum,ed. Prentice Hall.

INTERNETWORKING WITH TCP/IP,Volumen I,Englewood Clifs,ed. Prentice Halls.

Internet Tutorial via FTP from SunSite.UNC.EDU.

Capítulo 2

Modelo Cliente/Servidor

2.1. Introducción

Desde el punto de vista de una aplicación, TCP/IP, como muchos de los protocolos de comunicación de ordenadores, proporciona básicamente los mecanismos para la transferencia de datos. En particular, TCP/IP permite al programador establecer una comunicación entre dos programas de aplicación y pasar datos en ambas direcciones. Así, decimos que TCP/IP proporciona una comunicación par-a-par o en terminología inglesa *peer-to-peer*. Las aplicaciones pares además, pueden ser ejecutadas en una misma máquina o en diferentes máquinas.

Si bien TCP/IP especifica los detalles de cómo son transmitidos los datos entre un par de aplicaciones de comunicación, éste no dicta cuando y porqué interactúan las aplicaciones pares, no especifica cómo los programadores deberían organizar tales programas de aplicaciones en un entorno distribuido. En la práctica, un método organizacional domina el uso del TCP/IP hasta tal punto que casi todas las aplicaciones lo usan. El método ha dado en llamarse como el paradigma Cliente/Servidor. De hecho, el paradigma Cliente/Servidor ha llegado a convertirse en un esquema tan fundamental en los sistemas de red par-a-par que forma la base para la mayoría de las comunicaciones de ordenadores.

En la presente documentación se usará el paradigma Cliente/Servidor para describir todas las aplicaciones diseñadas. Se considerarán las motivaciones que hay detrás de este modelo, así como la descripción de las funciones de los componentes de un servidor y de un cliente. Además se explicará cómo construir una aplicación cliente y una aplicación servidor.

Pues bien, antes de entrar a ver cómo diseñar aplicaciones en base a este modelo es necesario definir antes algunos conceptos y la terminología que lleva consigo este modelo.

2.2. Motivación

La principal motivación que llevó a la creación de este modelo radica en el siguiente problema. Imaginemos a una persona intentando ejecutar dos programas en máquinas diferentes y hacer que se comuniquen. Es necesario recordar que el ordenador opera y computa bastante más rápido que nosotros. Bien, después de que la persona iniciase el primer programa, el programa comienza su ejecución y envía un mensaje a su par. En unos pocos milisegundos el programa se da cuenta que su par no existe todavía, por lo tanto se emite un mensaje de error y se aborta la ejecución. Mientras tanto, la persona ejecuta el segundo programa. Desafortunadamente, cuando el segundo programa comienza su ejecución, encuentra que su par acaba de abortar la ejecución. Por tanto, la probabilidad de que ejecutemos los dos programas en ambos extremos y de una manera sincronizada es tarea casi imposible más aun si es llevada a cabo por un humano que trabaja a una velocidad notablemente inferior a la de los ordenadores.

El modelo Cliente/Servidor resuelve este problema imponiendo que en cualquier pareja de aplicaciones de comunicación, una de ellas debe comenzar su ejecución y esperar (indefinidamente) a que la otra contacte con ella. La solución es importante ya que TCP/IP no responde a llamadas entrantes de petición de comunicación por sí mismo. Por tanto, es necesario implementar este mecanismo dentro de nuestra aplicación.

2.3. Terminología y Conceptos

El paradigma Cliente/Servidor divide a las aplicaciones de comunicación dentro de dos amplias categorías, dependiendo si la aplicación espera por llamadas de solicitud o si la inicia ella misma. En esta sección se intentará proporcionar una definición concisa de las ambas categorías.

2.3.1. Clientes y Servidores

El paradigma Cliente/Servidor utiliza la dirección de la iniciación de la comunicación para caracterizar si una aplicación es cliente o servidora. En general, una aplicación que inicia la comunicación par-a-par es denominada cliente. Los usuarios finales normalmente ejecutan aplicaciones del tipo cliente cuando necesitan hacer uso de un servicio de red. Mucho de los programas clientes consisten en programas de aplicación convencionales. Cada vez que se ejecuta una aplicación cliente, ésta contacta con una servidora, envía una solicitud o petición de un determinado servicio y espera por una respuesta de la aplicación servidora. Cuando llega la respuesta, la aplicación cliente continua su proceso. Las aplicaciones clientes son a menudo bastante más fáciles de diseñar que las servidoras, y además no necesitan de ciertos privilegios especiales dentro de los sistemas donde se ejecutan.

Por tanto, un servidor es cualquier programa que está en un estado de espera de alguna llamada por parte de una aplicación cliente que solicita alguno de los servicios que proporciona. Es decir, el servidor recibe la petición por parte del cliente, realiza o lleva a cabo el servicio oportuno y retorna el resultado al cliente.

Es necesario ahora aclarar el concepto de *programa servidor* dado que muy a menudo se confunde al **programa servidor** con la **máquina** en la que se está ejecutando. Es decir, se suele hablar de servidores refiriéndose a ordenadores cuando el término servidor se aplica a los programas que realizan tal servicio. Por poner un ejemplo, muchas personas dirían "éste ordenador es el servidor de ficheros" cuando deberían decir "en éste ordenador se está ejecutando el programa servidor de ficheros".

2.3.2. Privilegios y Complejidad

Debido a que las aplicaciones servidoras a menudo necesitan acceder a datos, realizar computaciones (acceder a la CPU), o a puertos de protocolo que protege el sistema operativo, la aplicación servidor requiere de una serie de privilegios especiales para poder acceder a todos estos recursos que el sistema operativo protege. Como la

aplicación servidora tiene acceso a estos recursos que a priori el sistema operativo tiene protegidos, hay que tener bastante cuidado para asegurar que de forma inadvertida se pasen de alguna forma estos privilegios a los clientes que usan dicho servidor. Por ejemplo, un servidor de ficheros que opera como un programa con privilegios debe tener códigos de chequeo sobre si un determinado fichero puede ser accedido por un cliente dado. El servidor no puede permitir que sea el sistema operativo el que controle esto ya que el servidor tiene privilegios adicionales sobre el S.O.

Por tanto, las aplicaciones servidoras deben tener código que controle lo siguiente:

- **Autentificación** - verificar la identidad del cliente.
- **Autorización** - determinar si un cliente dado se le permite acceder al servicio que proporciona el servidor.
- **Seguridad de los datos** - garantizar que los datos no son inintencionadamente revelados o comprometidos.
- **Privacidad** - Mantener la información sobre un individuo protegida de cualquier acceso no autorizado.
- **Protección** - garantizar que las aplicaciones de red no abusen de los recursos del sistema.

Como se verá más adelante, las aplicaciones servidoras que realizan una intensa computación o manejan grandes cantidades de datos, operan de forma más eficiente si manejan las peticiones por parte de las aplicaciones clientes de una forma *concurrente*. La combinación de los privilegios especiales junto con la operación concurrente hacen que las aplicaciones servidoras sean bastante más difíciles de diseñar que las aplicaciones clientes.

2.3.3. Parametrización de los clientes

Algunos programas clientes proporcionan más generalidades que otros. En particular, algunos permiten al usuario especificar la máquina remota en la que el programa servidor está operando así como el número de puerto del protocolo en el cual

está escuchando el servidor. Por ejemplo, de todos es conocido el uso del protocolo TELNET para acceder al servicio de terminal remoto de una máquina especifica. Pero también permite que, mediante la especificación de un puerto de protocolo, accedemos no al servicio de terminal remoto convencional sino a otro, que esta escuchando por ese puerto.

Conceptualmente, la aplicación que permite a un usuario especificar un número de puerto de protocolo tiene más parámetros de entrada que otra aplicación, por lo que usaremos el término de cliente totalmente parametrizado para describirlo. Muchas implementaciones del cliente TELNET interpretan a un segundo parámetro opcional como el número del puerto. La especificación de una máquina remota se proporciona bajo el primer parámetro como sigue:

telnet nombre-máquina

Si sólo se le proporciona el nombre de la máquina, TELNET utilizará el número de puerto bien conocido para este servicio. Para especificar ambas cosas, el nombre de la máquina y el número de puerto de protocolo, el usuario deberá especificar ambos como sigue:

telnet nombre-máquina número-puerto

Es necesario comentar que no todos los vendedores proporcionan en sus programas de aplicación clientes una total parametrización. Por esta razón, en algunos sistemas, sería dificil o imposible usar cualquier otro puerto diferente del puerto oficial del protocolo TELNET. De hecho, sería necesario modificar la aplicación cliente TELNET o escribir otro nuevo que aceptase un argumento como el número del puerto que debería usar. Por tanto, cuando construyamos aplicaciones clientes se recomienda una parametrización total.

La parametrización total es especialmente importante cuando estamos en la fase de comprobación de las aplicaciones diseñadas ya que de esta forma podemos proceder a

una comprobación de forma independiente y sin afectar a las aplicaciones realmente enganchadas a los puertos oficiales y que están ya en uso. Por ejemplo, un programador puede construir la pareja Cliente/Servidor del protocolo TELNET. Para probar su correcto funcionamiento los ejecuta usando para ello números de puertos no estándar, y proceder a su prueba sin perturbar a los servicios estándar enganchados a los puertos estándar.

2.3.4. Servidores TCP yUDP

Cuando los programadores diseñan las aplicaciones Cliente/Servidor deben elegir entre dos tipos de interacción: un estilo no orientado a conexión o uno sí orientado a conexión. Ambos estilos de interacción corresponden directamente a dos de los principales protocolos de transporte que TCP/IP proporciona. Si el cliente y el servidor se comunican usando UDP, la interacción es sin conexión o modo datagrama; si usan el TCP, la interacción es orientada a conexión, o en terminología inglesa en modo *stream*.

Desde el punto de vista del programador, la distinción entre ambos tipos de interacción es crítica ya que ella determina el nivel de seguridad que proporcionan los sistemas. Los detalles de estos dos protocolos se analiza con más detalle en la bibliografía recomnedada sobre Internet y TCP/IP.

Aquí veremos cuando usar uno y cuando usar otro. Es habitual que los programadores algunas veces cometan el error de, eligiendo un modo datagrama o UDP, construyan una aplicación y la comprueben sólo en una red de área local. Debido a que una red de área local rara vez o nunca retrasa paquetes, o los pierde, o se desordenan, la aplicación parece que trabaja de forma correcta. Sin embargo, si el mismo programa es usado a través de Internet y pasando por varias redes diferentes, probablemente fallaría o produciría resultados incorrectos.

Los principiantes, como los más expertos profesionales, prefieren el uso de un estilo orientado a conexión. Un protocolo orientado a conexión hace la programación más simple, y descarga al programador de la responsabilidad de detectar y corregir los errores. De hecho, proporcionar seguridad a un protocolo sin conexión en modo

datagrama como UDP es una tarea no muy trivial que normalmente requiere una considerable experiencia en el diseño de protocolos. Normalmente, un programa de aplicación solo usa UDP si:

1°≻ el protocolo de la aplicación especifica que se deba usar UDP (presumiblemente, el protocolo de aplicación habrá sido diseñado para tratar el tema de errores)

2°≻ el protocolo de aplicación necesita poder hacer broadcast o multicast.

3°≻ la aplicación no puede tolerar el *overhead* de un circuito virtual.

2.3.5. Servidores como Clientes

Los programas no siempre encajan exactamente dentro de la definición de cliente o servidor. Un servidor podría necesitar acceder a un servicio de red que hiciese que éste actuase como un cliente. Por ejemplo, supongamos que nuestro programa servidor de ficheros necesita obtener la hora del día para de ésta manera poder completar la información sobre la hora a la que se accedió a un fichero. Supongamos también que el sistema en el cual opera nuestro servidor no tiene reloj. Por tanto, para obtener la hora, el servidor debe actuar por un momento como un cliente enviando una petición de la hora que es a un programa servidor que suministre la hora. Esto se puede ver en la siguiente figura:

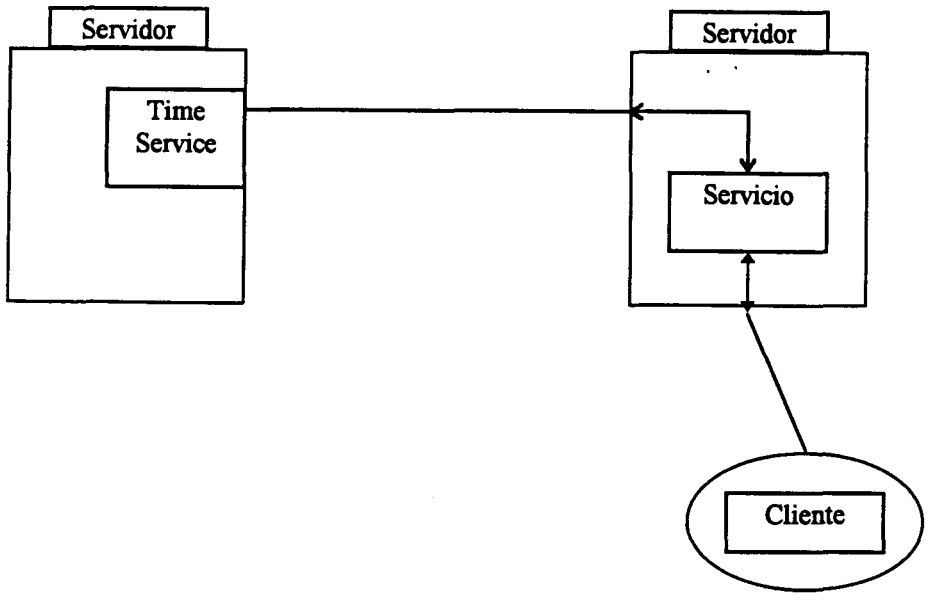


Figura 2.1.

Capítulo 3

Procesos Concurrentes en aplicaciones Cliente/Servidor

3.1. Introducción

En la sección anterior hemos definido el paradigma Cliente/Servidor. Ahora extendemos la noción de la interacción Cliente/Servidor introduciendo el proceso concurrente, un concepto que proporciona mucha más potencia a las interacciones entre clientes y servidores pero que a la vez hace de la programación una tarea más difícil, tanto en el diseño como en su construcción.

En esta sección a parte de la discusión del concepto de concurrencia también se analizará las facilidades que un sistema operativo multiproceso proporciona a la hora de la ejecución de procesos de forma concurrente. Es importante entender estas funciones ya que aparecerán muy a menudo en el código de ejemplos.

3.2. Concurrencia en redes

El término concurrencia se refiere a una utilización de un recurso, aparentemente o realmente, de forma simultánea. Por ejemplo, un sistema multiusuario puede lograr la concurrencia mediante una técnica denominada tiempo compartido, un diseño en el que a cada proceso se le asigna una franja de tiempo en la cual puede hacer uso del procesador. Si la franja de tiempo es suficientemente pequeña dará la impresión que se están atendiendo todos los procesos de forma simultánea. Pero esto es sólo aparentemente. El multiproceso es más potente ya que éste si permite gracias a que posee varios procesadores que se lleven a cabo múltiples computaciones de forma simultánea.

Los procesos concurrentes son fundamentales para distribuir la computación y ocurre de muchas formas. Entre una multitud de ordenadores en una única red, muchas

parejas de programas de aplicaciones se pueden comunicar de forma concurrente, compartiendo la red que las interconecta. Por ejemplo, una aplicación en una máquina A puede comunicarse con otra aplicación en otra máquina B, mientras otra aplicación en una tercera máquina C se comunica con la aplicación en una cuarta máquina D. Si bien todas ellas comparten una única red, las aplicaciones parecen proceder como si operasen de forma independiente. El hardware de red hace cumplir unas reglas de acceso que permiten a cada par de máquinas comunicadas el intercambio de mensajes. Las reglas de acceso impiden que un único par de aplicaciones que se están comunicando puedan emplear todo el ancho de banda del canal de comunicación excluyendo a otras de poder comunicarse.

Pero la concurrencia puede darse también dentro de una misma máquina. Por ejemplo, múltiples usuarios en un sistema de tiempo compartido pueden cada uno invocar una aplicación cliente que se comunique con una aplicación en otra máquina. Un usuario puede transferir un fichero mientras que otro usuario realiza un *login* remoto. Desde el punto de vista de un usuario la sensación es que todos los programas clientes ejecutados están operando de forma simultánea.

El software cliente normalmente no requiere una atención especial o esfuerzo por parte del programador para hacer que éste se pueda usar de forma concurrente. El programador diseña y construye una aplicación sin tener en cuenta una posible ejecución concurrente de su aplicación cliente; la concurrencia entre varios programas clientes ocurre de forma automática debido a que el sistema operativo permite múltiples usuarios que ejecuten de forma concurrente la aplicación cliente. De esta forma, los procesos clientes lanzados por diferentes usuarios operan como si de un programa convencional se tratasen.

3.3. Concurrencia en los programas servidores

En contraste a la concurrencia en el software cliente, la concurrencia dentro de un servidor requiere un esfuerzo considerable. Un único programa servidor debe atender varias peticiones por parte de los clientes de una forma concurrente. Para entender por qué la concurrencia es tan importante, consideremos operaciones del servidor que

requieren de una computación substancial o de un uso de los canales de comunicación bastante alto. Por ejemplo, pensemos en un servidor de un *login* remoto. Si éste no operase de forma concurrente sólo podría atender a un único *login* remoto a un tiempo. Desde que un cliente contacta con el proceso servidor, el servidor debe rechazar o ignorar subsiguientes peticiones por parte de otros clientes hasta que el primero terminase su sesión. Claramente, tales diseños limitan la utilidad del servidor e impiden a múltiples usuarios remotos su acceso a una misma máquina al mismo tiempo.

En lo que queda de esta sección se analizará la terminología y conceptos básicos usados.

3.4. Terminología y Conceptos

Debido a que pocos programadores de aplicaciones tienen experiencia en el diseño de programas concurrentes, el entender la concurrencia en los servidores puede ser un reto. Esta sección explica los conceptos básicos del proceso concurrente y muestra como un sistema operativo proporciona ésta concurrencia. Se definirá además la terminología usada más adelante y se verán algunos ejemplos.

3.4.1. El concepto de proceso

En sistemas de proceso concurrente, la abstracción "proceso" define la unidad fundamental de computación. La información más esencial asociada con un proceso es un puntero de instrucción que especifica la dirección por la que se va ejecutando el proceso dentro del código del programa. Otra información asociada con un proceso incluye la identidad del usuario al que pertenece, el programa compilado que se está ejecutando, y las posiciones de memoria donde se encuentra el código del proceso, así como de las áreas de datos.

Un proceso difiere de un programa porque el concepto de proceso incluye sólo la ejecución y no el código. Después de que el código haya sido cargado, el sistema operativo permite a uno o más procesos ejecutarlo. En particular, un sistema de proceso concurrente permite a múltiples procesos el ejecutar la misma porción de código "al

mismo tiempo". Esto significa que múltiples procesos deben estar ejecutándose en algún punto en el código. Cada proceso procede a su ritmo, y cada uno puede comenzar y terminar en un tiempo arbitrario. Debido a que tienen punteros de instrucción separados para cada proceso que indican la siguiente instrucción que será ejecutada, no existirá nunca ningún tipo de confusión.

En una arquitectura monoprocesador, la única CPU sólo puede ejecutar un proceso en un mismo instante de tiempo. En estos casos es el sistema operativo el que hace que parezca que se están realizando más de una computación al mismo tiempo, conmutando el tiempo de CPU entre todos los procesos de forma muy rápida. Desde el punto de vista de un usuario, parece que muchos procesos operan de forma simultánea. De hecho, un proceso procede por un corto lapso de tiempo, después otro y así sucesivamente. Se usa el término ejecución concurrente para describir esta idea. Significa *'aparentemente ejecución simultánea'*. En un monoprocesador, el sistema operativo maneja la concurrencia, mientras que en un multiprocesador, todas las CPU pueden ejecutar los procesos de forma simultánea.

El concepto que debe quedar claro es que los programadores pueden construir sus aplicaciones para un entorno concurrente, con independencia de si las capas subyacentes del hardware consisten en una arquitectura monoprocesador o multiprocesador.

3.4.2. Programas frente a Procesos

En un sistema de proceso concurrente, un programa de aplicación convencional es meramente un caso especial: consiste en una pieza de código que es ejecutada por exactamente un único proceso a un tiempo. La noción de proceso difiere de la noción convencional de programa en otros aspectos. Por ejemplo, muchos programadores de aplicaciones piensan que el conjunto de variables definidas en el programa están siendo asociadas con el código. Sin embargo, si más de un proceso ejecuta el código de forma simultánea, es esencial que cada proceso tenga su propia copia de las variables. Para entender por qué, consideremos el siguiente fragmento de código en C que imprime los enteros de 1 a 10:

```
for (i=0; i<10 ; i++)
    printf("%d\n", i);
```

En un programa convencional, el programador piensa que el almacenamiento de la variable *i* está siendo situada junto con el código. Sin embargo, si dos o más procesos ejecutan el segmento de código concurrentemente, uno de ellos podrá estar en la iteración sexta cuando el otro comienza la primera iteración. Cada proceso deberá por tanto tener su propia copia de la variable *i* o aparecerá una gran confusión.

Resumiendo, cuando múltiples procesos ejecutan un mismo segmento de código de forma concurrente, cada proceso debe tener su propia e independiente copia de las variables asociadas con el código.

3.4.3. Llamadas a procedimientos

En un lenguaje orientado a llamadas a procedimientos como Pascal o C, el código ejecutado puede contener llamadas a subprogramas (procedimientos o funciones). Los subprogramas aceptan argumentos, calculan un resultado, y retornan el control justo después del punto donde se les hizo la llamada. Si múltiples procesos ejecutan código concurrentemente, puede suceder que éstos se encuentren en diferentes puntos en la secuencia de ejecución de las llamadas a los procedimientos. Un proceso A, puede comenzar la ejecución, llamar a un procedimiento, y entonces llamar a un procedimiento de un segundo nivel antes de que otro proceso B comience. El proceso B podrá retornar del procedimiento del primer nivel justo cuando el proceso A retorna del procedimiento del segundo nivel.

Los sistemas en tiempo de ejecución para lenguajes de programación orientados a procedimientos usan un mecanismo de pila para manejar las llamadas a procedimientos. El sistema en tiempo de ejecución pone (*push*) un registro de activación de procedimiento en la pila siempre que se hace una llamada a un procedimiento. Entre otras cosas, el registro de activación mantiene información sobre la localización en el código en la cual se produce la llamada al procedimiento. Cuando el procedimiento

termina su ejecución, el sistema "*run-time*" saca (*pops*) el registro de activación de la cabeza de la pila., y retorna al procedimiento desde el cual la llamada ocurrió. Análogamente a la regla para las variables, los sistemas de programación concurrente proporcionan una separación entre las llamadas a los procedimientos para diferentes procesos.

Resumiendo, cuando ejecutan múltiples procesos un segmento de código concurrentemente, cada uno tiene su propia pila en tiempo de ejecución de los registros de activación de procedimientos.

3.5. Ejemplo de un Proceso Concurrente

Veamos ahora dos ejemplos muy sencillos en C, para clarificar la idea de concurrencia y que efectos puede provocar.

3.5.1. Un ejemplo en C secuencial

El siguiente ejemplo ilustra el proceso concurrente en un sistema operativo UNIX. Como muchos de los conceptos de ordenadores, la sintaxis del lenguaje de programación es trivial; sólo ocupa unas pocas líneas de código. Por ejemplo, el siguiente código es un programa convencional en C que imprime los enteros del 1 al 5 junto con su suma total:

```
#include <stdio.h>
int sum; // Variable sum como variable global.
main() {
    int i; // Variable i como local.
      sum = 0;
      for (i=1; i<=5 ; i++) { //Bucle que iteractúa de 1 a 5 sobre i.
            printf("El valor de i es %d\n", i);
            fflush(stdout); // Limpia el buffer.
            sum += i;
      } printf("La suma es %d\n", sum); exit(0); // Finaliza el programa.}
```

Cuando se ejecuta el programa produce la siguiente salida:

El valor de i es 1
El valor de i es 2
El valor de i es 3
El valor de i es 4
El valor de i es 5
La suma es 15

3.5.2. La versión Concurrente

Para crear un nuevo proceso en UNIX, un programa llama a la función del sistema *fork()*. En esencia, *fork()* divide el programa que se está ejecutando en procesos idénticos, ambos ejecutándose en el mismo lugar en el mismo código. Los dos procesos continúan igual como si dos usuarios los hubiesen llamado simultáneamente. Por ejemplo, la siguiente versión modificada del ejemplo anterior llama a la función *fork()* para crear un nuevo proceso. (Notar que si bien la introducción de la concurrencia cambia el significado del programa completamente, la llamada a *fork()* sólo ocupa una línea dentro del código).

```
#include <stdio.h>
int sum;
main() {
        int i;
        sum=0;
        fork(); // Creamos un nuevo proceso que se ejecutará paralelamente.
        for (i=0; i<=5; i++) {
                printf("El valor de i es %d\n", i);
                fflush(stdout);
                sum +=i;
        }
        printf("La suma es %d\n", sum);
        exit(0);}
```

Cuando un usuario ejecuta la versión concurrente del programa, el sistema comienza con un único proceso ejecutando el código. Sin embargo, cuando el proceso alcanza la llamada *fork()* en el código, el sistema duplica el proceso y permite a ambos, el original y el nuevo, que sean ejecutados. De acuerdo, cada proceso tiene su propia copia de las variables que usa el programa. De hecho, la manera más fácil de ver qué pasa es imaginarse que el sistema hace una segunda copia entera del programa que se está ejecutando. Entonces se imagina uno que ambas copias se ejecutan (cómo si dos usuarios hubiesen ejecutado de forma simultánea el programa).

En un sistema particular monoprocesador, la ejecución de nuestro ejemplo concurrente produciría 12 líneas de salida:

El valor de i es 1
El valor de i es 2
El valor de i es 3
El valor de i es 4
El valor de i es 5
La suma es 15
El valor de i es 1
El valor de i es 2
El valor de i es 3
El valor de i es 4
El valor de i es 5
La suma es 15

En el hardware que estamos usando, el primer proceso es ejecutado tan rápido que entra dentro de la franja de tiempo en la que el procesador está atendiendo a ese proceso y por tanto termina antes de pasar al siguiente proceso. Para entender un poco mejor que ocurre recurramos al siguiente gráfico:

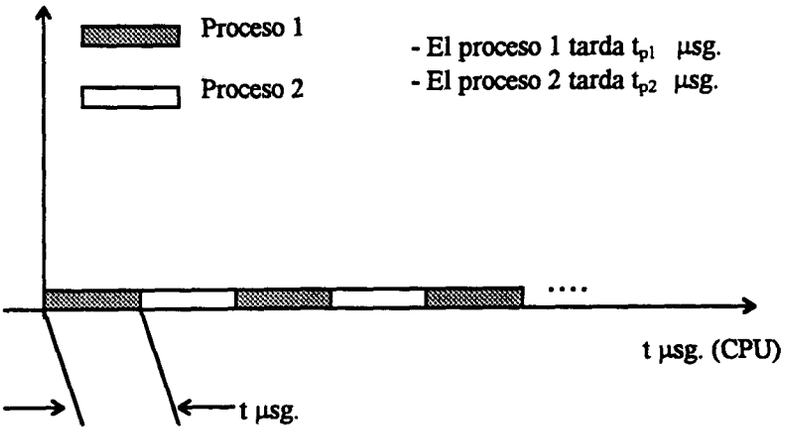


Figura III. 1

t: duración de una franja de CPU.

⇨ Si $t_{p1} \leq t$ y $t_{p2} \leq t$ entonces le da tiempo a ambos procesos de terminar dentro de una ranura y el resultado final es que se han ejecutado secuencialmente, primero P1 y después P2.

⇨ Sin embargo, si $t_{p1} > t$ y/o $t_{p2} > t$ entonces para que uno de los procesos finalice es necesario de más de una ranura de tiempo tal y como se indica en el gráfico. Es entonces cuando ambos procesos se llevan a cabo de forma intercalada durante el tiempo necesario hasta que uno de ellos termine. En este caso, si la salida de los resultados es por la pantalla, durante t µsg. saldrían los resultados del proceso 1 y durante los siguientes t µsg. saldrían los resultados del proceso 2, así hasta que finalizase alguno de los dos procesos. El resultado en pantalla sale por tanto desordenado.

3.5.3 Reparto de la CPU por Franjas de Tiempo

En el programa de ejemplo, cada proceso lleva a cabo una cantidad pequeña de computación ya que sólo iteractúa durante 5 veces en un bucle bastante simple. Por tanto, desde que un proceso gana el control de la CPU éste completa su ejecución rápidamente debido a su poco volumen de cálculo computacional que requiere. Si examinamos procesos concurrentes que llevan a cabo substancialmente más cantidad de computaciones que nuestro ejemplo, tiene lugar entonces un fenómeno interesante que

ya se comentó en el gráfico anterior: el sistema operativo asigna la potencia disponible de la CPU a cada uno durante un corto periodo de tiempo antes de pasar el control sobre la CPU al siguiente proceso. Usaremos el término de CPU compartida por franjas de tiempo o en inglés *timeslicing* para describir o referirnos a sistemas que comparten la disponibilidad de la CPU entre varios procesos concurrentes, de forma que a cada proceso se le asigna una "*rodaja*" de tiempo durante la cual puede hacer uso de la CPU. Por ejemplo, si un sistema de éstos tiene sólo una CPU para ser asignada y un programa se divide en dos procesos, uno de los procesos se ejecutará por un periodo de tiempo, entonces el segundo se ejecutará por otro periodo igual, después volverá a ejecutarse el primero y así seguirá hasta que alguno de los procesos finalice.

Un mecanismo para repartir el uso de la CPU en franjas de tiempo intenta asignar de forma igualitaria el tiempo de proceso disponible entre todos los procesos existentes. Si sólo existen dos procesos y el ordenador tiene un único procesador, cada uno recibiría aproximadamente el 50% del tiempo disponible de la CPU. Si existen N procesos en un ordenador con un único procesador, cada recibiría aproximadamente 1/N del tiempo disponible de la CPU. Así, todos los procesos parecen proceder a una misma velocidad, sin importar cuantos procesos existan a la vez. Como es de esperar y bastante lógico, cuanto más procesos existan de forma concurrente, más baja será la velocidad de proceso en su conjunto.

Para ver el efecto de este sistema, necesitamos un programa de ejemplo en el cual cada programa tarde en ejecutarse más tiempo que el equivalente a una ranura de tiempo del procesador. Ahora extendamos el programa anterior a 1000 iteraciones del bucle en vez de a cinco. El efecto será que el primer proceso empezará a imprimir números hasta que se acabe su tiempo de CPU y entonces se pase al segundo proceso que empezará su impresión hasta que de nuevo se le acabe su tiempo de CPU. Así hasta que terminen. El efecto es que los resultados saldrán por pantalla mezclados los de un proceso y los de otro.

3.5.4. Divergencia de los procesos

Veamos un poco más sobre la función *fork()*. Hemos dicho que *fork()* puede ser usada para crear un nuevo proceso que ejecute exactamente el mismo código que el del proceso original. La creación de una copia idéntica de un programa en ejecución no es muy interesante y no es muy útil debido a que significa estar ejecutando ambas copias para llevar a cabo la misma tarea y computación. En la práctica, el proceso creado por *fork()* no es absolutamente idéntico al original: difiere en un pequeño detalle. *fork()* es una función que retorna un valor al que la invoca. Después de ejecutada la función el valor retornado al proceso original difiere del valor retornado al proceso recién creado. En el proceso recién creado, *fork()* retorna un valor igual a cero; en el proceso original, *fork()* retorna un pequeño entero que identifica al proceso recién creado. Técnicamente, el valor retornado es llamado identificador de proceso o en terminología inglesa *procces identifier*. En algunos libros se habla de los identificadores de procesos bajo el nombre de pid (*procces id.*)

Los programas concurrentes usan el valor retornado por la función *fork()* para decidir como proceder. En el caso más común, el código contiene una cláusula condicional que mira si el valor retornado es cero o no para distinguir si se trata del código del proceso recién creado o si se trata del proceso base:

```
int sum;
main() {
        int pid;
        sum=0;
        pid=fork();
        if (pid!=0) { // Si el pid es distinto de cero entonces se trata del Proceso original
                printf("El Proceso original escribe esto. \n");
        } else {  // Por el contrario, si el pid es 0, se trata del Proceso recién creado.
                printf("El nuevo proceso es el que escribe esto. \n");
        }
        exit(0);}
```

Este ejemplo que parece tan sencillo esconde un mundo. En la variable pid registramos el valor retornado por la función *fork()*. Recordemos que si se trata del proceso recién creado retornará cero y en caso de ser el original retornará el identificador del proceso recién creado. Si hacemos memoria y nos vamos unos apartados atrás, veremos que cuando existen varios procesos de un mismo código ejecutándose se hace una copia de sus variables para cada proceso. Aquí ocurre eso. Podemos hacer una primera aproximación suponiendo que se crea una copia idéntica del código mostrado y en la primera copia que se ejecuta(proceso original) la variable pid al llegar a la sentencia *fork()* retornará un valor que será el identificador del nuevo proceso creado, y que es diferente del que se retorna en la copia creada por *fork()* al llegar a ese punto, en la que retornará un valor igual a cero. Ahora hagamos la aproximación real a lo que pasa. No se copian los programas íntegros sino que del código se mantiene una única copia y son las variables las que se duplican al llegar a la sentencia *fork()*. El valor de la variable pid del proceso original será distinto de cero y el del proceso creado será igual a cero. Por eso, cuando el sistema operativo esté con el proceso original, irá a buscar la variable pid del mismo y ejecutará el código mostrado, mientras que cuando el sistema operativo esté tratando al otro proceso buscará la variable pid de éste y recurrirá al mismo código que antes pero ahora con la variable pid del segundo proceso.

Resumiendo, el valor retornado por *fork()* difiere del proceso original y del recién creado; los programas concurrentes usan esta diferencia para permitir al nuevo proceso ejecutar una parte diferente del código que el proceso original.

3.6. Concurrencia y Entradas/Salidas Asíncronas

Por último y para terminar ésta sección añadir que al igual que algunos sistemas operativos proporcionan soporte para el uso concurrente de la CPU, también algunos sistemas permiten que una única aplicación inicie y controle varias operaciones de entradas y salidas de forma concurrente. En BSD UNIX, la llamada al sistema *select()* proporciona una operación fundamental al respecto para que los programadores puedan construir programas que traten de forma concurrente varias operaciones de I/O. En un primer acercamiento, la misión de esta función es bastante fácil de entender: permite a un

programa preguntar al sistema operativo cuales de los dispositivos de I/O están disponibles para su uso.

Un ejemplo, imaginemos un programa de aplicación que lee caracteres de una conexión TCP y los escribe en la pantalla. El programa debería también permitir al usuario escribir comandos en el teclado para controlar cómo son mostrados los datos en la pantalla. Debido a que rara vez (o nunca) un usuario escribirá algún comando, el programa no puede estar esperando por una entrada desde el teclado - debe continuar y leer los mensajes de la conexión TCP y mostrarlos. Sin embargo, si el programa intenta leer de la conexión TCP y no hay datos disponibles, el programa se bloqueará hasta que lleguen más datos. El usuario no podrá escribir un comando mientras el programa está bloqueado esperando por datos que leer desde la conexión. El problema es que la aplicación no puede saber si los datos llegarán primero desde el teclado o desde la conexión TCP. Para resolver este dilema, un programa UNIX llama a *select()*. Haciendo esto, se pregunta al sistema operativo que le deje saber cual de las fuentes de entrada de datos esta disponible primero. La llamada retorna un valor tan pronto como una fuente de datos este lista, procediendo nuestro programa a leer de ella. Por ahora, tan sólo es importante entender la idea que hay detrás de la función *select()*. En secciones venideras se analizará con más detalle.

Capítulo 4

Interfase de los programas con los protocolos

4.1. Introducción

A principios de los 80, ARPA (Advanced Research Projets Agency) fundó un grupo en la Universidad de California en Berkeley para transportar el software TCP/IP al sistema operativo UNIX y hacer que el resultado de este software estuviese disponible en otros lugares. Como parte del proyecto, los diseñadores crearon una interfase que utilizaban las aplicaciones para comunicarse entre ellas. Decidieron usar las llamadas al sistema ya implementadas en UNIX siempre que fuese posible y añadir otras nuevas para soportar funciones del TCP/IP que no se adaptasen fácilmente dentro del conjunto de funciones ya existentes. El resultado dio lugar a lo que denominan ***The Sockets Interface***, y al sistema operativo Berkeley Software Distrubition UNIX o BSD UNIX.

4.2. Especificando un Protocolo de Interfase

Cuando los diseñadores consideran cómo añadir funciones a un sistema operativo para proporcionar a los programas de aplicación el acceso al software del protocolo TCP/IP, deben elegir nombres para las funciones y deben especificar los parámetros que cada función acepta. En esta labor, decidieron el alcance de los servicios proporcionados por las funciones y el estilo en el que las aplicaciones harían uso de ellas. Los diseñadores deben considerar también la posibilidad de implementar funciones para soportar protocolos TCP/IP ó ir más allá y diseñar funciones para ser utilizadas bajo otros protocolos en el futuro. Así, los diseñadores deben elegir entre dos de las siguientes aproximaciones:

a) Definir funciones específicas para soportar comunicaciones bajo TCP/IP.

b) Definir funciones que soporten comunicaciones de red en general, y usen parámetros para hacer del protocolo TCP/IP un caso especial.

Las diferencias entre ambas aproximaciones son fáciles de comprender por su impacto en los nombres de las funciones del sistema y los parámetros que las funciones requieren. Por ejemplo, en la primera aproximación, un diseñador debería elegir tener una función del sistema llamada make*tcp*connection() , mientras que en la segunda, un diseñador debería crear una función general llamada makeconnection() y usar un parámetro para especificar que se trata del protocolo TCP.

Debido a que los diseñadores de Berkeley han querido acomodar múltiples conjuntos de protocolos de comunicación, han usado la segunda aproximación. De echo, durante todo el diseño han proporcionado generalidad más allá del TCP/IP. Han permitido múltiples familias de protocolos, con todos los protocolos TCP/IP representados en una única familia (familia PF_INET). Han decidido también tener operaciones específicas de aplicaciones usando un "tipo de servicio" en lugar de especificar el nombre del protocolo. Así, en lugar de especificar que se quiere una conexión TCP, una aplicación pediría el tipo de servicio "*stream transfer*" usando la familia de protocolos Internet.

La interfase de Sockets de Berkeley proporciona por tanto funciones generalizadas que soportan comunicaciones de red usando muchos posibles protocolos. Una llamada Socket se refiere a todos los protocolos TCP/IP como una única familia de protocolos. La llamada permite al programador especificar el tipo de servicio requerido antes que el nombre de un protocolo específico.

4.3. La abstracción de los sockets

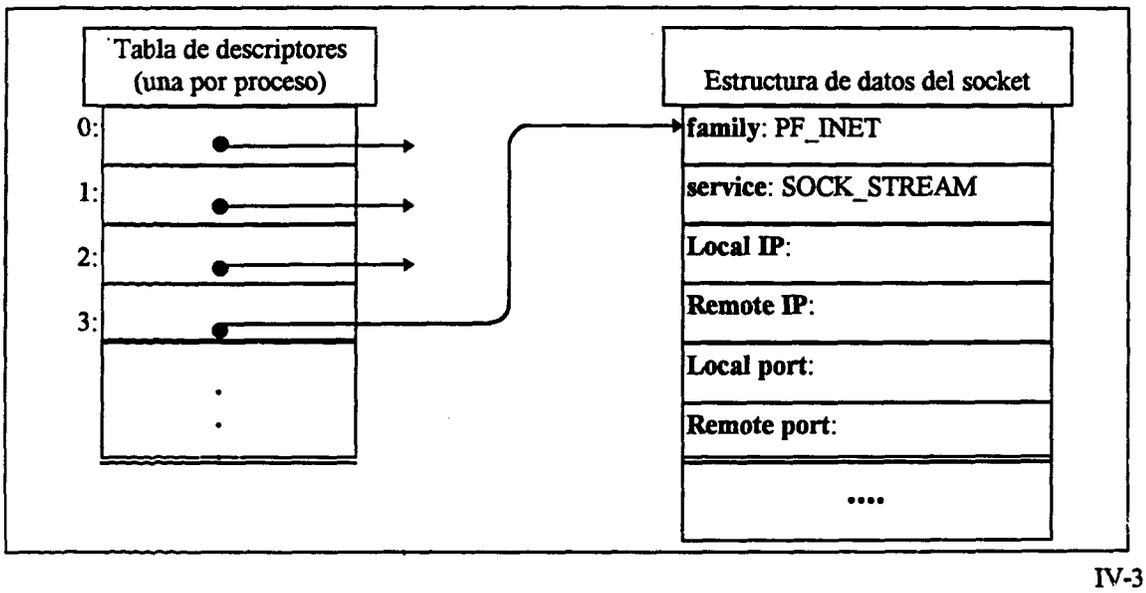
En UNIX, una aplicación que necesita llevar a cabo una I/O llama a la función *open()* para crear un "*file descriptor*" que es usado para acceder al fichero. El sistema operativo implementa el descriptor de fichero como un entero corto que referencia a un índice de una matriz de punteros. Cada puntero referencia a una estructura de datos interna del fichero en cuestión. El sistema mantiene una tabla de descriptores de ficheros para cada proceso. Cuando un proceso abre un fichero, el sistema sitúa un puntero, que direcciona la estructura de datos interna para ese fichero, en la tabla de descriptores de fichero para ese proceso y retorna el índice del puntero dentro de la matriz al que llama

la función *open()*. Los programas de aplicaciones tan sólo deben recordar el número del descriptor dentro de la tabla y usarlo en siguientes llamadas que requieran cualquier operación sobre el fichero.

La interfase socket añade una nueva abstracción para las comunicaciones de red, el socket. Como en los ficheros, un socket es identificado por un entero corto llamado *socket descriptor* ó descriptor del socket. UNIX sitúa a los descriptores de sockets en la misma tabla que los descriptores de ficheros. Por este motivo, un descriptor de un fichero y el de un socket no pueden tener el mismo índice en la tabla, es decir, no pueden tener el mismo descriptor un fichero y un socket.

La idea general que subyace en los sockets es que una única llamada al sistema es suficiente para crear cualquier socket. Desde que el socket ha sido creado, una aplicación deberá realizar llamadas al sistema adicionales para especificar los detalles de su uso exacto, es decir, completar si es necesario las estructuras de datos asociadas al socket y que están direccionadas por el puntero de la matriz de descriptores del proceso cuyo índice se indica por el descriptor del socket.

La manera más fácil de entender la abstracción de los sockets es echar una mirada en las estructuras de datos. Cuando una aplicación realiza la llamada socket, el Sistema Operativo sitúa una nueva estructura de datos para almacenar la información necesaria para la comunicación, y rellenar una nueva entrada a la tabla de descriptores que contendrá un puntero que hace referencia a la estructura de datos antes mencionada.



Cuando realizamos una llamada a la función *socket*() para crear un nuevo socket, la estructura de datos asociada a él tendrá muchos campos que tendremos que rellenar posteriormente. Como se verá, la aplicación que haya creado el socket deberá hacer llamadas adicionales al sistema para completar esa información en la estructura de datos del socket antes de que éste pueda ser usado.

Una vez que un socket ha sido creado, éste puede ser usado para esperar una solicitud de conexión o bien para iniciar nosotros una a través de él. Un socket usado por un servidor para esperar llamadas entrantes que soliciten conexiones es llamado un socket pasivo, mientras que un socket usado por un cliente para iniciar una conexión es llamado un socket activo. La única diferencia entre un socket activo y uno pasivo se basa en cómo son usados por las aplicaciones; los sockets son creados en un principio de igual forma.

La base para aplicaciones de red en la versión 4.3 de BSD es una abstracción referida a los sockets. Un socket es un punto final de una comunicación. Esta primera definición de un socket es bastante simple pero muy intuitiva. **La verdadera interpretación de un socket es como la de un punto de acceso al servicio de transporte**. Los programas de aplicación piden al Sistema Operativo la creación de un socket cuando necesitan de los servicios de la capa de transporte con el fin de llevar a cabo un comunicación a través de la red. El sistema en respuesta retorna un entero que la aplicación utiliza para referenciar a dicho socket. La versión 4.3 BSD por ejemplo, se refiere a ambos, sockets y descriptores de ficheros, como descriptores y tan sólo los diferencia entre ellos cuando son usados.

La principal diferencia entre los descriptores de sockets y los de ficheros es que en el Sistema Operativo un socket puede ser creado sin necesidad de especificarle una dirección mientras que un descriptor de ficheros sí es necesario enlazarlo con un fichero o device. La aplicación puede elegir entre suministrar la dirección destino cada vez que usa el socket o elegir enlazarlo a la dirección destino y así permitir especificar la dirección destino repetidamente.

Dependiendo del tipo de comunicación que se desea realizar a través de la red, existen diferentes tipos de sockets:

➔**Sock_Stream**: Es usado para comunicaciones orientadas a conexión y los servicios son proporcionados por el protocolo TCP en la capa de transporte.
➔**Sock_Dgram**: Es usado para comunicaciones no orientadas a conexión y los servicios son proporcionados por el protocolo UDP en la capa de transporte.
➔**Sock_Raw**: Es un socket utilizado para acceder directamente con el protocolo IP en la capa de red. Es una interfase directa desde la capa de aplicación hacia la capa de red.

Más adelante entraremos con más detalle en qué tipos de sockets soporta la especificación de Windows Sockets, pero por el momento veamos todos los aspectos de forma general y tal y como se entienden en UNIX para después particularizar a Windows Sockets.

Queda por tanto claro que un socket no es más que un Punto de Acceso al Servicio de Transporte y que las aplicaciones pueden usarlo para llevar a cabo comunicaciones a través de la red.

La interfase Sockets de Berkeley proporciona una serie de llamadas al sistema con las que el programador podrá establecer comunicaciones a través de la red. Más adelante, cuando estudiemos Windows Sockets veremos cada una de estas rutinas.

4.4. Especificar una dirección de un destino

Cuando un socket es creado este todavía no contiene información detallada de como será usado. En particular, el socket no contiene información sobre el número del puerto de protocolo o la dirección IP. Antes de que una aplicación use un socket ésta debe primero especificar la dirección de la máquina remota ó de la máquina remota y la local.

Así, el conjunto de protocolos TCP/IP define a un punto final de comunicación como una dirección IP más un número del puerto de protocolo. Por tanto, cualquier dirección en TCP/IP se compondrá del par (IP,port).

Tal y como se vio en el capítulo primero, la dirección Internet puede ser suministrada según un nombre de host (cic.teleco.ulpgc.es) que se resolverá mediante el DNS en una dirección numérica, o mediante un número (198.145.144.1). Con esta dirección IP sólo identificamos al host, y ahora es necesario identificar el programa con el que nos queremos comunicar en ese host. Para ello deberemos especificar el número de puerto a través del cual ese programa esta "escuchando" la llegada de peticiones de servicio por parte de clientes.

4.5. Funciones Bloqueantes y no-bloqueantes

El concepto de bloqueo es un concepto bastante importante cuando utilizamos funciones de I/O. Un bloqueo en una aplicación es causado por una función o llamada al sistema que tarda un tiempo considerable en llevarse a cabo. El ejemplo más típico de bloqueo suele ser el que produce la función recv() cuando no hay datos que leer. Esta función se queda esperando hasta que lleguen datos para leer.

Un socket puede funcionar en modo bloqueante o no-bloqueante. Por defecto actuará en modo bloqueante, lo que significa que todas aquellas funciones marcadas en la especificación como bloqueantes podrán producir bloqueos en nuestro programa. Por el contrario y mediante la utilización de otras funciones podremos conmutar el estado del socket a modo no-bloqueante. En ese modo si la función que se llama produce bloqueo, entonces inmediatamente es anulada y se devuelve un error. De ese modo el programa no se ve bloqueado pero deberá tener en cuenta el tratamiento de ese error.

Veremos con más detalle este apartado de los bloqueos en el capítulo dedicado a la especificación Windows Sockets.

Capítulo 5

Algoritmos en el diseño de aplicaciones cliente

5.1. Introducción

En los capítulos anteriores se consideró la abstracción del socket que usan las aplicaciones para acceder a los servicios de TCP/IP. En este capítulo discutiremos los algoritmos básicos en el diseño de aplicaciones clientes. Se mostrará cómo las aplicaciones llegan a ser clientes al iniciar una comunicación, cómo usan los protocolos TCP y UDP para contactar con un servidor, y cómo usan las llamadas a las funciones de los sockets para interactuar con dichos protocolos.

5.2. Algoritmos en general

Debido a que TCP/IP proporciona una rica funcionalidad que permite a los programas comunicarse en una variedad de formas, una aplicación que hace uso de TCP/IP debe especificar bastantes detalles sobre la comunicación que desea. Por ejemplo, la aplicación debe especificar si quiere actuar como un cliente o como un servidor, las direcciones de los puntos finales que serán usados, si utilizará una comunicación orientada a conexión o no, y detalles como el tamaño de los buffers que usará entre otros parámetros.

Anteriormente ya hemos analizado un conjunto de operaciones que puede utilizar una aplicación. Desafortunadamente, conociendo el nivel más bajo de los detalles de todas las posibles llamadas a las funciones y sus parámetros de forma exacta, no proporciona a los programadores una comprensión de cómo diseñar programas distribuidos bien construidos. De hecho, mientras que una comprensión general sobre las funciones usadas para la comunicación es importante, pocos programadores recuerdan los detalles de cada una. En vez de eso, los programadores aprenden y comprenden las líneas generales de cada posible diseño. En esencia, los programadores conocen lo suficiente sobre los algoritmos de la computación distribuida para tomar decisiones en sus diseños y elegir entre los algoritmos disponibles de una forma rápida. Es entonces cuando se consulta un manual de

programación para encontrar los detalles necesarios para poder escribir un programa que implemente un algoritmo particular en un sistema dado.

5.3. Arquitectura Cliente

Las aplicaciones que interactúan como clientes son conceptualmente más sencillas que sus homólogas las aplicaciones servidoras por muchas razones. Una de las principales razones es que la mayoría de los programas clientes que existen o que se diseñan no manejan explícitamente interacciones de forma concurrente con múltiples servidores, por lo que el tema de concurrencia en programas clientes no suele aparecer.

5.4. Identificando la localización de un Servidor

El programa cliente puede usar varios métodos para encontrar la dirección IP de un servidor junto con su puerto de protocolo.

- ⇨ tener el nombre del servidor o su dirección IP especificada como una constante cuando se compila el programa.
- ⇨ requerir al usuario la identificación de un servidor cuando se invoca el programa.
- ⇨ obtener información sobre el servidor desde un fichero o una unidad de disco local.
- ⇨ ó usar un protocolo separado para encontrar un servidor. (por ejemplo un mensaje multicast o broadcast al cual debe responder el servidor buscado).

El especificar la dirección del servidor como una constante hace al programa cliente más rápido. Sin embargo, esto significa también que el programa cliente debe ser recompilado si se cambia de servidor. Y más importante, esto significa que el programa cliente no podrá ser utilizado con un servidor alternativo, incluso para cuestiones de pruebas y comprobaciones del programa. Como un compromiso algunos programas clientes fijan un nombre de un servidor en vez de una dirección IP. Fijando por tanto un nombre en vez de una dirección IP se retrasa el enlazamiento hasta el tiempo de ejecución. Esto permite a un lugar elegir un nombre genérico para un servidor y añadir un *alias* al sistema

de dominios de nombres para dicho nombre. El uso de los *alias* permite al manager del sistema cambiar la localización de un servidor sin necesidad de cambiar el programa cliente. Para mover el servidor, el manager necesita cambiar solo el *alias*. Por ejemplo, es posible añadir un *alias* para el mailhost en el dominio local y disponer que todos los clientes miren por la cadena "*mailhost*" en vez de mirar por una dirección específica de una máquina. Debido a que todos los programas clientes hacen referencia a dicho nombre genérico en vez de a una máquina en concreto, el manager del sistema puede cambiar la localización del servidor *mailhost* sin necesidad de que el programa cliente sea recompilado.

Existen otras posibilidades de cómo especificar las direcciones del servidor. Una de ellas es que estén almacenadas en un fichero y que nuestro programa acceda a ese fichero para ver dichas direcciones. Otra es utilizar un protocolo de *broadcast*, es decir, en este tipo de protocolos se lanza a la red un paquete que va dirigido a todos los terminales de la red. El terminal al que se quiere conectar uno, que es especificado en el paquete enviado, reconoce este paquete y envía una respuesta dando a conocer su dirección en la red. Este tipo de protocolos además de crear un gran tráfico en la red no son muy eficientes en redes de gran tamaño como Internet, por tanto esta solución queda descartada.

Por tanto, para evitar mucha complejidad y ninguna dependencia con el entorno local en el que trabajamos es preferible diseñar programas que pidan al usuario que identifiquen la dirección IP o el nombre del servidor al cual quieren conectarse.

5.5. Buscando en el dominio de nombres

Un cliente debe especificar la dirección de un servidor mediante el uso de la estructura *sockaddr_in*. Hacer esto significa convertir una dirección en notación decimal de punto (ej: 193.145.140.2) o un nombre que identifica al host (cic.teleco.ulpgc.es) en una dirección IP de 32 bits representada en binario. La conversión desde la notación decimal es bastante inmediata y trivial. La conversión desde un nombre del dominio de nombres sin embargo requiere más esfuerzo. La interfaz socket BSD incluye rutinas de librería que hacen este trabajo. Las funciones que llevan a cabo este trabajo son *inet_addr()* y *gethostbyname()* respectivamente. *Inet_addr()* tiene un parámetro de entrada que es una cadena ASCII la cual contiene la dirección en decimal, y retorna su equivalente en binario y

en orden de red. *gethostbyname()* tiene como parámetro de entrada una cadena ASCII que contiene el nombre del host y retorna un estructura hostent donde se almacena información relativa a este host. Dentro de esta estructura se encuentra la dirección IP del host ya en orden de red también.

```
struct hostent {
char    *h_name;  /*Nombre oficial del host*/
char    **h_aliases;  /*lista de otros alias*/
int     h_addrtype;   /* tipo de direcciones*/
int     h_length;    /*longitud de la dirección*/
char    **h_addr_list;  /*lista de direcciones*/
};
#define h_addr h_addr_list[0]
```

Los campos que contienen nombres y direcciones deben ser listas porque los host que tienen múltiples interfaces tienen también múltiples nombres y direcciones. Veamos ahora un pequeño código donde se hace uso de esta función y esta estructura.

```
struct hostent *hptr;
char *example="cic.teleco.ulpgc.es \0";
if (hptr=gethostbyname(example)) {
        /* La dirección IP esta ahora en hptr->h_addr */
} else {
        /* ERROR */
}
```

Si la llamada a la función se realiza correctamente entonces *gethostbyname()* retorna un puntero a una estructura hostent válida. Si el nombre del host no puede ser mapeado dentro de una dirección IP entonces gethostbyname retorna cero.

5.6. Localizar el número de un puerto bien conocido

Muchos programas clientes deben averiguar el número del puerto al que se deben conectarse en función del servicio que desean usar. Para ayudar al programador en esta situaciones la interfaz socket BSD ha proporcionado otra rutina de librería la cual se encarga de: dado un nombre de un servicio bien conocido, retornar su número de puerto en una estructura *servent*. Esta función se denomina *getservbyname()* y tiene dos parámetros de entrada. El primero es un puntero a una cadena de caracteres en el cual especificamos el nombre del servicio, y el segundo es otro puntero a una cadena de caracteres donde especificamos el protocolo a usar (TCP o UDP). Este último parámetro es opcional y si lo queremos omitir debemos poner *NULL*. Veamos ahora la estructura servent.

```
struct servent {
char    *s_name;  /*nombre oficial del servicio */
char    **s_aliases;  /*lista de alias */
int     s_port;  /* número de puerto para este servicio */
char    *s_proto;       /* protocolo a usar */
};
```

Si un programa cliente TCP necesita averiguar el número oficial del puerto para engancharse al servicio SMTP éste debería llamar a *getservbyname()*...

```
struct servent *sptr;

if (sptr=getservbyname("smpt", "tcp")) {
        /* Número de puerto en sptr->s_port  */
} else {
        /* ERROR  */ }
```

El número de puerto contenido en ***sptr->s_port*** está en orden de red.

5.7. El algoritmo del programa cliente TCP

El algoritmo usado por una aplicación que usa TCP a grandes rasgos es como sigue:

1°.-Encontrar la dirección IP y el número del puerto de protocolo del servidor con el que se desea comunicar.
2°.-Crear un socket.
3°.-Especificar que la conexión necesita de un puerto arbitrario y no usado en la máquina local y permitir a TCP que elija uno.
4°.-Conectar el socket al servidor.
5°.-Comunicarse con el servidor haciendo uso del protocolo de aplicación correspondiente.
6°.-Cerrar la conexión.

5.8. Crear un socket

Antes de poder realizar cualquier tipo de comunicación es necesario crear un socket a parte de realizar después otras tareas. Cuando se crea un socket se debe especificar primero una familia de protocolos, así como el tipo de servicio requerido, si se trata de un servicio orientado a conexión o no. Finalmente se debe especificar qué protocolo se utilizará para dar este servicio. En TCP/IP para servicios orientados a conexión se utiliza el TCP y para servicios no orientados a conexión se utiliza el UDP. Si en el parámetro protocolo no especificamos ninguno (ponemos un 0) entonces Windows Sockets elige o bien TCP o UDP según el servicio solicitado en el parámetro anterior. Por su parte, la función *socket(int* af, *int* type, *int* protocol*)* devolverá un descriptor del nuevo socket.

5.9. Elegir un número de puerto local

Una aplicación necesita especificar una dirección local para un socket antes de que pueda ser usado para la comunicación. Un servidor opera sobre un número de puerto bien conocido, el cual deben conocer todos los clientes. Sin embargo, un cliente TCP no tiene

porque operar en un puerto preasignado. En general, el cliente no se debe preocupar por el número de puerto que utilice siempre y cuando:

Primero: el puerto no coincida y por tanto no provoque conflictos con los puertos de otras aplicaciones que estén presentes en ese momento.

Y segundo: que el puerto no coincida con alguno de los preasignados como puertos bien conocidos.

Puertos bien conocidos preasignados son todos aquellos de los servicios oficiales de Internet como Telnet, Gopher, fpt, ...

Así, un cliente puede elegir un número de puerto arbitrario siempre y cuando cumpla las dos reglas anteriormente citadas. Puede parecer que esto es una tarea complicada de cara al programador pero la interfaz socket hace que la elección del puerto sea simple dado que proporciona una forma de detección y asignación de éste automática y el programador no tiene que implementar procedimientos para realizar esta tarea. La elección del puerto local que cumpla los requisitos listados arriba ocurre como un efecto lateral de la función *connect(SOCKET s, const struct sockaddr FAR *name, int namelen)*. Como podemos ver los parámetros de esta función constan de un descriptor del socket a conectar, una estructura referenciada por un puntero del tipo sockaddr en la cual se especifica la dirección (IP,puerto) del lugar a donde nos vamos a conectar y finalmente la longitud de esa estructura.

5.10. Conectándose un socket TCP a un servidor

Para poder conectarnos a un servidor es necesario hacer uso de la función *connect()*. Esta función dado un descriptor de un socket y una dirección de un punto final, realiza la conexión. Tras una llamada a *connect()* sin errores se establece una conexión y por tanto ya serán posibles las transferencias de datos mediante las funciones *send(SOCKET s, const char FAR *buf, int len, int flags)* y *recv(SOCKET s, char FAR *buf, int len, int flags)*. En los parámetros de las funciones *send*() y *recv*() se debe especificar el descriptor del socket a través del cual se van a enviar/recibir los datos, un puntero que referencia a un buffer

donde están almacenados los datos a enviar o donde se almacenarán los datos recibidos, la longitud de ese buffer y como último parámetro un flag. Este flag será usado para actuar sobre la conducta de la función más allá de lo especificado en las opciones del socket. Esto es, la semántica de la función vendrá determinada por las opciones del socket y por este flag. El flag sólo puede tomar dos valores, uno MSG_DONTROUTE y MSG_OOB. El significado de ambas opciones se detalla en la especificación.

5.11. Comunicándose con el Servidor usando TCP

En el siguiente ejemplo en C bajo UNIX se muestra una posible interacción de un cliente con un servidor. Hacer notar que los ejemplos mostrados aquí y durante estos capítulos están diseñados en un entorno UNIX debido a la simplicidad del código. En un entorno Windows el código aumenta considerablemente y se pierde por tanto la visión del problema. Además, es necesario advertir que bajo UNIX las funciones *send()* y *recv()* son implementadas por *write()* y *read()* respectivamente. Como se puede observar, la emisión de una petición es bastante sencilla y tan sólo es necesario especificar sobre qué socket se mandarán los datos (el socket previamente debe estar conectado), dónde se encuentran los datos y qué longitud en bytes tienen esos datos.

```
#define BLEN 120  /*definición del tamaño del buffer */
char *req = "petición de cualquier tipo";
char buf[BLEN]; /* buffer para almacenar la respuesta */
char *bptr;  /*puntero que referencia al buffer*/
int n; /*número de bytes leídos*/
int buflen; /*espacio libre en el buffer*/
bptr=buf;
buflen = BLEN;

/* Se supone que el socket ya ha sido conectado */
/* al extremo remoto previamente */

/* Enviamos la petición */
```

```
write (s,req,strlen(req));
/*leer la respuesta (puede venir troceada en varias piezas) */
while ((n=read(s,bptr,buflen) > 0) {
        bptr += n;  // Adelanta el puntero hasta el final de los escrito
        buflen -= n; // Decrementa el tamaño del buffer en función del resto
}
```

5.12. Leyendo una respuesta desde una conexión TCP

Por otro lado, la lectura de un respuesta puede ser una tarea algo más complicada. Puede suceder que los datos que lleguen deban ser leídos de varias veces debido a su longitud, o que por el contrario los datos sean tan pocos que no terminen de llenar el buffer. Para poder manejar estas situaciones de forma eficiente la función read() ó su análoga recv() en W.S., retorna el número de bytes leídos. En caso de que el número de bytes leídos de la conexión fuese -1 sería una indicación de error. Como se puede ver en el ejemplo, la función *read()* se encuentra dentro de un bucle, el cual se esta repitiendo mientras el número de bytes leídos sea mayor que cero. Los datos leídos son escritos en el buffer *bptr* que tiene una longitud de *buflen*.

5.13. Cerrando una conexión TCP

La operación de desconexión o cierre de una conexión TCP es un tema bastante importante. Existen dos funciones involucradas, la *closesocket()* y la *shutdown()*. La segunda se comenta en el siguiente apartado. De todas formas, en el capítulo dedicado a las funciones de Windows Sockets se dará una explicación más detalladas de ambas funciones.

5.13.1. La necesidad de una desconexión parcial

Debe existir una coordinación entre servidor y cliente ya que en una comunicación TCP existe un canal en el que coexisten ambos sentidos de la comunicación. Romper ese canal significaría dejar de escuchar al servidor por parte de un cliente y por parte de un servidor significará el no saber si desea más peticiones el cliente. El servidor no puede cortar de una forma abrupta porque no sabe si el cliente le va a enviar más peticiones y el

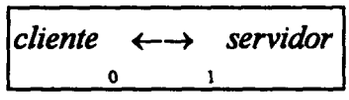
cliente tampoco puede cortar por lo sano porque no sabe cuando ha terminado de recibir todos los datos del servidor. Por lo tanto debe existir una coordinación entre ambos.

5.13.2. La operación de un cierre parcial

Para resolver este problema la interfaz socket BSD incluye una primitiva que permite a las aplicaciones finalizar una conexión TCP en una sola dirección. La función *shutdown()* acepta dos argumentos, uno que es un descriptor de un socket y otro que es la especificación de la dirección; la función entonces cierra la conexión del socket especificado en la dirección dada.

shutdown (s, dirección);

- dirección es un entero. Si es **0** significa que se ha cerrado la conexión de forma que no se permiten más entradas de datos por el socket. Si es **1** significa que no se permite más salidas de datos por el socket. Y si es **2** significa que se ha deshabilitado la transmisión en ambos sentidos.



5.14. Algoritmo de un cliente UDP

El algoritmo de un cliente UDP es el siguiente:

1°.- Encontrar la dirección IP y el número de puerto de protocolo del servidor con el que nos vamos a conectar.

2°.-Crear el socket.

3°.-Especificar que el socket no necesita de un número de puerto determinado y por tanto se puede elegir uno al azar.

4°.-Especificar el servidor al cual los mensajes serán enviados.

5°.-Comunicarse con el servidor según el protocolo de aplicación correspondiente.

6°.-Cerrar el socket.

Aunque el algoritmo parezca igual que el de TCP se debe aclarar que la forma utilizada a la hora de comunicarse con el servidor según el protocolo de aplicación correspondiente difiere mucho de un esquema a otro. Por ejemplo, en el modelo UDP en ese punto sería necesario implementar mecanismos de seguridad propios para asegurar que la información llegue a su destino correctamente, mientras que el modelo TCP esto no sería necesario.

5.15. Socket UDP conectados y no conectados

En principio, el modo datagrama es un modo no orientado a conexión. Ese modelo permite enviar un paquete a través de un mismo socket a lugares diferentes cada vez. Gracias a la función *sendto()* se puede llevar a cabo esta tarea. Por otro lado, en modo datagrama cada vez que recibimos un mensaje sería bastante interesante el obtener la dirección del que lo envió. Es decir, dado que no existe ninguna conexión con ninguna máquina, podemos recibir mensaje a través del socket de varios puntos finales. Por tanto, para poder enviarle una respuesta es necesario antes obtener la dirección del mensaje recibido. Esta tarea la lleva acabo a función *recvfrom()*.

5.16. Usando connect() con UDP

El utilizar la función *connect()* en un modelo no orientado a conexión puede resultar algo absurdo, sin embargo no lo es. Gracias a esta función, es posible asignar, por decirlo de alguna forma, una dirección final por defecto. Es decir, cuando se envíe un datagrama con un *send()* en vez de con un *sendto()*, el datagrama se dirigirá a la dirección a sobre la cual se hizo un *connect()* previamente.

La llamada *connect()* no produce por tanto ningún tipo de intercambio de paquetes y tampoco comprueba la validez o existencia del otro extremo de la comunicación. Tan sólo almacena en las estructuras relacionadas con el socket la dirección del punto final para posteriores transferencias de datos con *send()* y *recv()*.

5.17. Comunicarse con un servidor usando UDP

Cada vez que una aplicación llama a *send()* se envía un único mensaje. El mensaje contiene todos los datos pasados a *send()*. De forma similar la llamada *recv()* retorna un mensaje completo. Se asume que el cliente ha especificado un buffer lo suficientemente grande como para poder almacenar un mensaje. Así, un cliente UDP no necesita hacer repetidas llamadas a la función *recv()* para obtener un único mensaje; por cada llamada a *recv()* se extrae un mensaje completo.

5.18. Cerrar un socket que usa UDP

Un cliente UDP hace uso de la función *closesocket(SOCKET s)* para cerrar un socket y liberar todos los recursos asociados con él. Una vez que el socket ha sido cerrado, el programa cliente debe rechazar subsiguientes mensajes que lleguen a la dirección donde trabajaba el socket. Sin embargo, la máquina donde ocurre la llamada *closesocket()*, no informa al otro extremo de la comunicación que el socket ha sido eliminado. Así que una aplicación que utilice un protocolo de transporte no orientado a conexión debe estar diseñada para que el extremo remoto sepa cuando se cierra la conexión, ya que UDP no proporciona ninguno.

5.19. Un cierre parcial para UDP

La función *shutdown()* puede ser usada con un socket UDP si lo que se quiere es parar la transmisión de datos en uno de los sentidos de la comunicación. Desafortunadamente, al contrario que en un cierre parcial en TCP, cuando se aplica a un socket UDP, *shutdown()* no envía ningún tipo de mensaje al otro extremo comunicando el hecho. En vez de ello, marca al socket local como no disponible para la transferencia de datos en las direcciones especificadas. Así, si un cliente cierra con *shutdown()*, la transmisión hacia el servidor por ese socket, el servidor no recibirá nunca ninguna indicación de que la comunicación por parte del cliente ha cesado.

Capítulo 6

Servidores

6.1. Algoritmos en el diseño de servidores

6.1.1. Introducción

En esta primera parte del tema se considerará el diseño de los programas servidores. Se discutirán temas tales como cuándo usar un acceso al servidor orientado a conexión o no, cuándo realizar implementaciones concurrentes frente a implementaciones iterativas, etc... Se describen por tanto las ventajas de cada una de las aproximaciones comentadas, dándo ejemplos de situaciones en las cuales dichas aproximaciones son válidas.

6.1.2. El algoritmo conceptual del servidor

Conceptualmente, cada servidor sigue un algoritmo simple: se crea un socket y se enlaza a un puerto bien conocido sobre el que se quiere que se reciban todas las peticiones. Es entonces cuando entra en un bucle infinito en el cual acepta la petición llegada por parte del cliente, procesa la petición, formula una respuesta adecuada, y envía el resultado al cliente.

Desafortunadamente este algoritmo conceptual poco sofisticado sólo funciona para los servicios más triviales. Para entender por qué, considérese un servicio tal como un "*file transfer*" (aplicación de transferencia de ficheros), el cual requiera un tiempo considerable para tratar cada petición por parte de un cliente. Supongamos que el primer cliente que contacta con el servidor pide que le sea transferido un fichero de 6 megabytes, mientras que un segundo cliente tan sólo pide que le sea transferido un pequeño fichero de poco más de 20 bytes. Si el servidor espera hasta que la primera transferencia sea llevada a cabo para comenzar con la segunda , el segundo cliente tendrá que esperar un tiempo muy grande por un fichero muy pequeño.

6.1.3. Servidores concurrentes frente a iterativos

Usamos el término iterativo para describir una implementación servidora que procesa una única petición a un mismo tiempo, y el término concurrente cuando nos referimos a un servidor que es capaz de manejar más de una petición al mismo tiempo. Sin embargo muchos de los servidores concurrentes logran una aparente concurrencia sin ser necesaria una implementación concurrente. En particular, si un servidor lleva a cabo el proceso de los datos de un cliente de forma rápida en comparación con la cantidad de operaciones de entrada/salida (I/O) que realiza, el servidor puede ser implementado como un único proceso que utiliza operaciones de entrada/salida de forma asíncrona para permitir el uso simultáneo de múltiples canales de comunicación. Desde el punto de vista de un cliente, el servidor parecería que se comunica con múltiples clientes de forma concurrente.

Por tanto, el término de servidor concurrente se refiere a si el servidor maneja o controla múltiples peticiones concurrentemente, y no a si las implementaciones subyacentes usan múltiples procesos de forma concurrente.

En general, los servidores concurrentes son más difíciles de diseñar y construir, y el código resultante es más complejo y difícil de modificar. Sin embargo muchos programadores eligen implementaciones concurrentes en el diseño de sus servidores dado que los servidores iterativos causan retardos innecesarios en aplicaciones distribuidas y pueden formar un cuello de botella que afecta a muchas aplicaciones clientes. El efecto del cuello de botella no es más que la aparición de una cola de clientes ya que no pueden ser atendidos todos a la vez y deben esperar unos por otros, y los retardos producidos por una aplicación iterativa son los relacionados con la espera en esa cola.

6.1.4. Acceso orientado a conexión y no orientado a conexión

El tema de la conectividad se centra alrededor del protocolo de transporte que utiliza un cliente para acceder a un servidor.Ya se ha comentado que TCP/IP implementa ambos tipos de conexión, TCP y UDP.

Esta terminología se suele aplicar a los servidores cuando sería más preciso si lo restringiéramos a los protocolos de aplicación, dado que la elección del tipo de conexión a utilizar depende del protocolo de aplicación. En los siguientes puntos analizaremos ambos tipos de conexión un poco más a fondo.

6.1.5. Servidores orientados a conexión

La verdadera ventaja de una aproximación orientada a conexión reside en la sencillez a la hora de programar. En particular, dado que el protocolo de transporte se hace cargo de los paquetes perdidos y los que llegan fuera de orden, el servidor no tiene por qué preocuparse de estos detalles. Un servidor orientado a conexión maneja y usa conexiones. Éste acepta una conexión entrante de un cliente, y entonces se establece una conexión por donde se enviarán todos los datos. El servidor recibirá por este canal peticiones del cliente y enviará sus oportunas respuestas. Finalmente, el servidor cierra la conexión después de completar la interactuación.

Mientras permanece abierta una conexión, TCP proporciona todos los mecanismos de seguridad necesarios. Pero los servidores orientados a conexión también tiene desventajas. Los diseños orientados a conexión requieren de un socket separado para cada conexión, mientras que los diseños no orientados a conexión permiten la comunicación con múltiples *hosts* a través de un único socket. La creación de los sockets y el manejo de la conexión puede ser especialmente importante dentro de un sistema operativo que debe estar funcionando siempre sin abusar de los recursos. En el caso de aplicaciones triviales, el costo de usar un modelo TCP a uno UDP es bastante. En el primero es necesario usar más sockets y por tanto más estructuras y buffers que no son más que más recursos del sistema. Estos recursos del sistema además pueden ser mal utilizados en el caso de que las conexiones de los clientes fallasen, es decir, si el cliente se viene abajo y el servidor no lo advierte y mantiene el socket y la conexión abiertos. Además no se debe olvidar que los servidores deben ser diseñados para que estén operando siempre, por periodos de tiempo muy grandes, sin abusar de los recursos del sistema.

6.1.6. Servidores no orientados a conexión

Los servidores no orientados a conexión también tienen sus ventajas y desventajas. Como ya es sabido, el protocolo de transporte UDP no proporciona una seguridad de los datos transmitidos por lo que es necesario crear nuestros propios mecanismos de seguridad y fiabilidad en la transmisión en nuestro protocolo de aplicación.

Uno de los extremos (el lado servidor o el cliente) debe tomar la responsabilidad de controlar este problema. Normalmente, los clientes toman la responsabilidad de la retransmisión de las peticiones si no se recibe una respuesta por parte del servidor. Si el servidor necesita dividir su respuesta en varios paquetes es necesario también implementar el consiguiente mecanismo de retransmisión de forma adecuada.

El lograr una seguridad y fiabilidad a través de los mecanismos de *timeout* ya comentados en el capítulo 1 no es una tarea muy fácil. De hecho requiere ser un considerable experto en el diseño de protocolos. Dado que TCP/IP opera en un entorno Internet donde los retardos extremo a extremo cambian muy rápidamente, el uso de unos valores de *timeout* fijos no funciona. Muchos programadores aprenden esta lección cuando trasladan sus aplicaciones de una red de área local (la cual tiene retardos considerablemente menores y con pocas variaciones) a una red tan grande como es Internet (la cual tiene grandes retardos con una gran variación). Para acomodarse a un entorno Internet, la estrategia de la retransmisión debe ser adaptativa. Así las aplicaciones deben implementar un esquema de retransmisión tan complejo como el usado por TCP. Por tanto se suele animar a los programadores novatos que utilicen un protocolo de transporte orientado a conexión.

Otra consideración a la hora de elegir uno u otro modelo de conexión se centra en si el servicio diseñado necesita hacer un *broadcast* o *multicast*. Dado que TCP ofrece una comunicación punto a punto, éste no puede suministrar operaciones de *broadcast* o *multicast*; tales servicios requieren de UDP. En la práctica en muchas situaciones se intentan evitar operaciones de *broadcast* dentro de lo posible.

6.1.7. Cuatro tipos básicos de servidores

Los servidores como hemos visto pueden ser iterativos o concurrentes y pueden usar protocolos de transporte orientados a conexión o no orientados a conexión. Por tanto, se pueden agrupar cuatro tipos básicos:

Iterativo No-Orientado a Conexión	Iterativo Orientado a Conexión
Concurrente No-Orientado a Conexión	Concurrente Orientado a Conexión

Estos cuatro tipos se irán viendo en apartados posteriores con más detalle.

6.1.8. Tiempo de proceso de las peticiones

En general, los servidores iterativos tan sólo satisfacen los protocolos de aplicación más triviales dado que hacen que cada cliente espere su turno en vez de ser tratados a la vez.

Se define el tiempo de proceso de una petición (t_{pp}) por parte del servidor al tiempo total que tarda el servidor en manejar una única petición solamente y se define el tiempo de respuesta observado por el cliente (t_{roc}) como el retardo total entre la vez que envió la petición al servidor y cuando éste responde. Es obvio que el tiempo observado por un cliente nunca podrá ser menor que el tiempo que tarda en procesar el servidor una petición. Sin embargo, si el servidor tiene una cola de peticiones que manejar, el tiempo de respuesta observado por el cliente puede ser mucho más grande que el tiempo de proceso del servidor de una única petición.

Los servidores iterativos manejan una única petición a un tiempo. Si llegase otra petición mientras el servidor está ocupado atendiendo una petición existente, el sistema introduce en una cola a las nuevas peticiones. Una vez que el servidor haya terminado de procesar una petición, mira en la cola y extrae la siguiente petición si es que existe alguna. Si N es la longitud media de la cola de peticiones, el tiempo de respuesta observado para

una petición que llega será aproximadamente N/2+1 veces el tiempo de proceso del servidor de una petición. Como se observa el tiempo se incrementa en proporción a N y es por esto que algunos servidores iterativos restrinjan N a un valor tal como 5. Es más, si un servidor tiene K clientes y estos clientes envían R peticiones por segundo, el tiempo de proceso de peticiones por parte del servidor deberá de ser de menos de 1/KR segundos por petición. Si el servidor no puede alcanzar este promedio la cola de las peticiones provocará un overflow. Para evitar por tanto el *overflow* en servidores que tiene un largo tiempo de proceso de las peticiones se deberá usar un esquema concurrente y dejar a un lado el iterativo.

En la siguiente figura se aclarará los tiempos comentados anteriormente para su mejor comprensión.

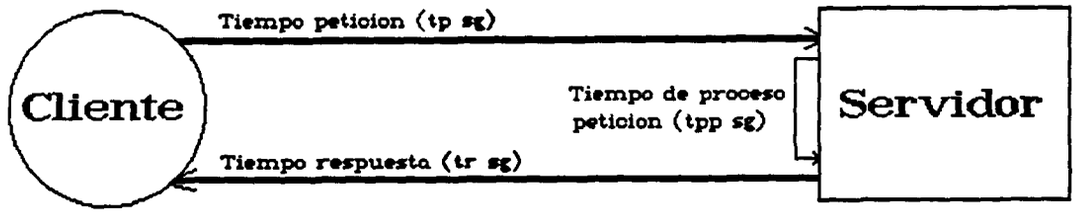


Figura 6.1.

El tiempo observado por el cliente es:

$$t_{roc} = t_p + t_{pp} + t_r$$

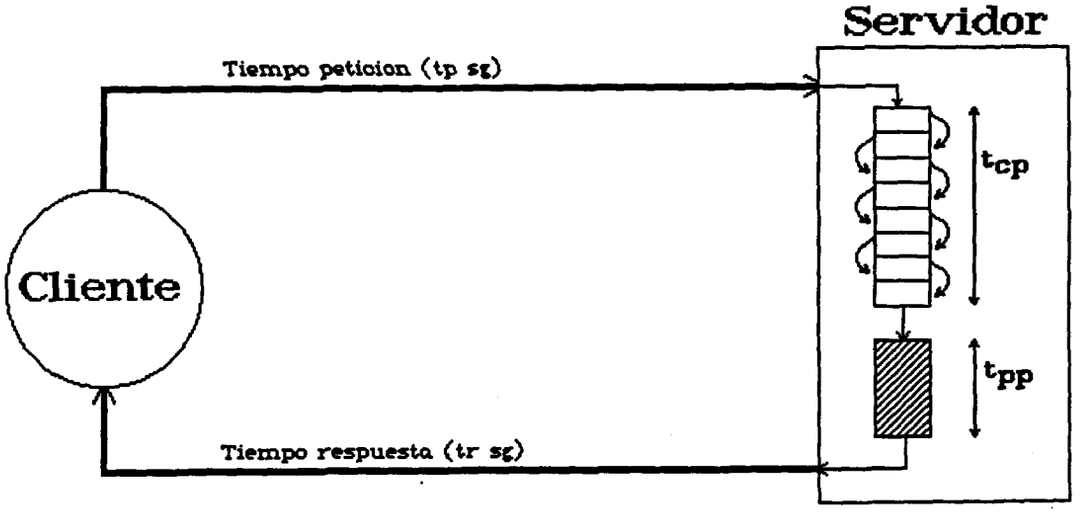


Figura 6.2.

t_{cp}: Tiempo en la cola de peticiones

t_{pp}: Tiempo de proceso de una petición

Cuanto mayor sea la cola mayor será t_{cp} y por tanto mayor es t_{roc}. En estos casos, si la cola es de N elementos, $t_{roc} = (\frac{N}{2} + 1) \cdot t_{pp}$

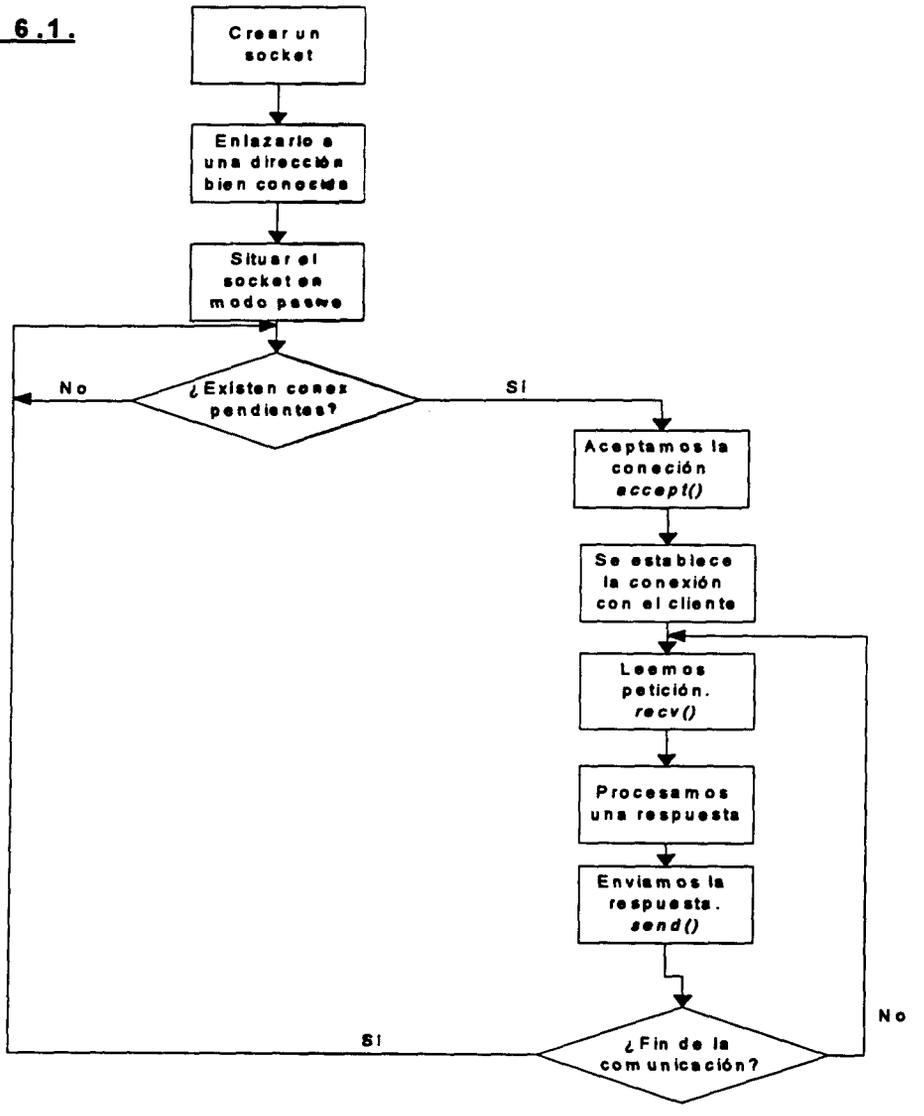
6.1.9. Algoritmos de los servidores iterativos

Un servidor iterativo es el más sencillo de diseñar, programar, depurar y modificar. Normalmente los servidores iterativos trabajan mejor con servicios simples cuyo acceso sea a través de un protocolo no orientado a conexión.

6.1.10. Algoritmo de un servidor iterativo y orientado a conexión

El siguiente algoritmo es el de un servidor accedido vía TCP. En los siguientes párrafos se aclarará cada una de las secciones de este algoritmo.

Algoritmo 6.1.



6.1.10.1. Enlazarse a una dirección utilizando INADDR_ANY

Un servidor necesita crear un socket y enlazarlo a un puerto bien conocido para el servicio que ofrece. Como los clientes, los servidores usan *getportbyname()* para dado un nombre de un servicio averiguar el correspondiente número de puerto bien conocido que tiene asignado. Claro está que estos puertos obtenidos por medio de esta función son para los servicios definidos en Internet. Nosotros podemos crear un servicio que no este reconocido en el mundo y le asignamos un número de puerto que identificará a nuestro nuevo servicio que hemos creado.

La función *bind(SOCKET s, const struct sockaddr FAR *addr, int namelen)* especifica un punto final de comunicación para un socket. Hace uso de tres parámetros.

Uno es el socket al que vamos a enlazar[#] , el siguiente parámetro es la dirección a la que vamos a enlazar el socket y el último no es más que la longitud de la estructura donde almacenamos la dirección especificada. Como vemos, esta función hace uso de la estructura *sockaddr_in*, la cual contiene un número de puerto y una dirección IP. De esta forma, *bind()* no puede especificar el puerto de protocolo usado sin especificar también una dirección IP. Desafortunadamente, seleccionar una dirección IP específica desde la cual el servidor atenderá y aceptará las conexiones puede causar alguna dificultad. Para hosts que tengan una única conexión de red, la elección es obvia porque el host tiene sólo una dirección IP. Sin embargo, *gateways* y *multi-homed* (multi-homed significa un host con varias direcciones diferentes) tiene múltiples direcciones. Si el servidor especifica una dirección particular cuando enlaza al socket, el socket no aceptará comunicaciones de clientes que accedan a esa máquina por medio de otra dirección IP.

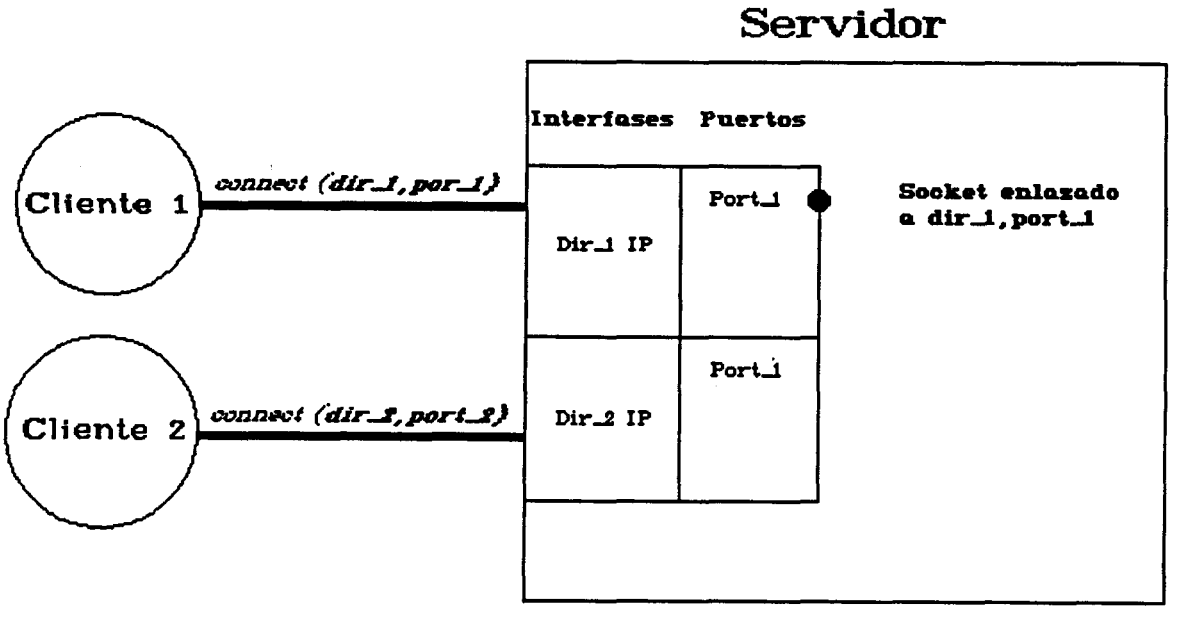


Figura 6.3.

El cliente 1 accede al servicio pero el cliente 2 aunque accede al mismo servidor no logra contactar con el servicio dado que accede a través de otra interfase y el servicio esta enlazado tan solo a la dirección Dir_1.

[#] la acción de enlazar un socket no es más que asignarle un nombre. Un nombre no es más que una dirección completa que lo identifique. Por ejemplo, en TCP/IP sería el par (IP,port)

Para resolver el problema la interfase socket define una constante especial, INADDR_ANY, que puede ser usada en lugar de una dirección IP. El usar esta constante evita tener que poner todas las direcciones IP de que dispone el host, lográndose de esta forma que se acepten llamadas desde todas las "puertas de entrada" definidas en el host.

6.1.10.2. Socket en modo pasivo

La llamada de un servidor TCP a la función *listen()* sobre un socket determinado, sitúa a éste en un modo pasivo. Dicha función acepta un argumento que es la longitud de la cola sobre dicho socket para peticiones.

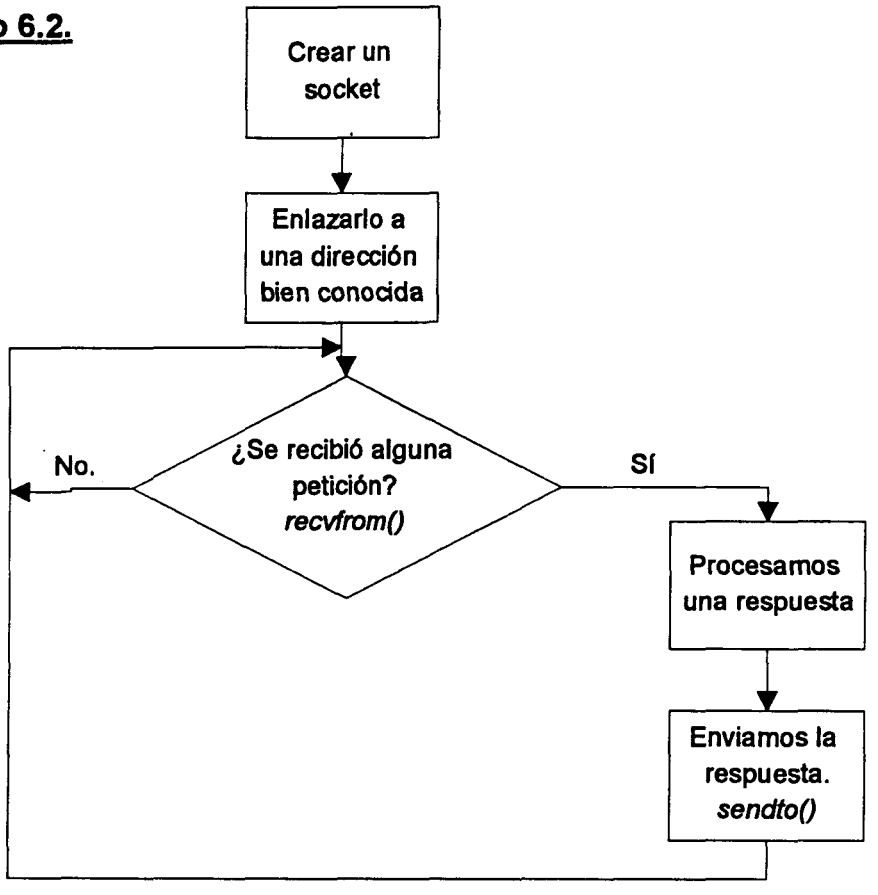
6.1.10.3. Aceptar una conexión

Un servidor TCP llama a la función *accept()* para obtener la siguiente petición de la cola de peticiones. La llamada retorna el descriptor de un socket el cual debe ser usado para llevar a cabo la comunicación. Una vez que es aceptada la conexión, el servidor usa las funciones *send()* y *recv()* sobre el socket creado para interactuar con el cliente según el protocolo de aplicación diseñado. Cuando el servidor termina cierra la conexión mediante la función *closesocket()*.

6.1.11. Algoritmo de un servidor iterativo no orientado a conexión

El algoritmo es el que se muestra a continuación. y como en el caso anterior se comentarán las particularidades en los subsiguientes apartados.

Algoritmo 6.2.



6.1.11.1. Formando una dirección de respuesta

La interfase socket proporciona dos maneras de especificar la dirección de un punto final. Después de que un cliente llama a *connect()* éste puede usar *send()* para enviar datos dado que la estructura de datos interna del socket contiene la dirección del punto final. Un servidor no orientado a conexión no puede usar la función *connect()* dado que hacer esto significa restringir la comunicación del socket con un cliente remoto específico por lo que el servidor no podría usar el socket otra vez para datagramas de varios clientes arbitrarios. Es por esto por lo que un servidor no orientado a conexión hace uso de un socket no conectado. El servidor genera de forma explícita una dirección de respuesta y usa la función *sendto*() para especificar los datos que serán enviados y la dirección a donde irán. Esta función tiene la siguiente apariencia:

ret_code = sendto (s, message, len_message, flags, to_addr,to_addr_len)

donde ***s*** es un socket no conectado[##] , ***message*** es la dirección de un buffer donde se encuentra los datos a enviar, ***len_message*** especifica la longitud de dicho buffer, ***flags*** es un argumento donde se especifican ciertas opciones de control, ***to_addr*** es un puntero a una estructura *sockaddr_in* donde se almacena la dirección a donde se enviarán los datos y por último ***to_addr_len*** es la longitud en bytes de dicha estructura especificada en el parámetro anterior.

La interfase socket proporciona también un forma fácil de obtener por parte de un servidor no orientado a conexión la dirección de un cliente: el servidor obtiene la dirección del cliente al que debe mandar una respuesta a partir de la petición de éste. Es decir, la dirección del cliente se encuentra en la petición de éste. La función *recvfrom()* se encarga de dicho trabajo. Tiene dos buffers, uno en el que se almacena los datos de la petición del cliente y otro donde se almacena la dirección del cliente que hace la petición.

ret_code = recvfrom (s, data_buf, len, flags, from_addr, from_addr_len);

El servidor para generar una respuesta a esta petición de un cliente utiliza la dirección almacenada en el buffer apuntado por ***from_addr***.

6.1.12. Algoritmos de servidores Concurrentes

La razón fundamental por la que se introduce la concurrencia en los servidores es debido a la necesidad de proporcionar un menor tiempo de respuesta a múltiples clientes.

En los servidores concurrentes aparecen los términos *Master/Slave*. Si bien es posible que un servidor alcance algo de concurrencia usando un único proceso, muchos de los servidores concurrentes usan múltiples procesos. La forma de proceder es crear un único proceso servidor maestro que comienza su ejecución inicialmente. El proceso maestro abre un socket en el puerto bien conocido, espera por la llegada de una petición por parte de un cliente, y crea un proceso servidor esclavo que es el encargado ahora de manejar la petición. El servidor maestro nunca se comunica directamente con un cliente, éste pasa la

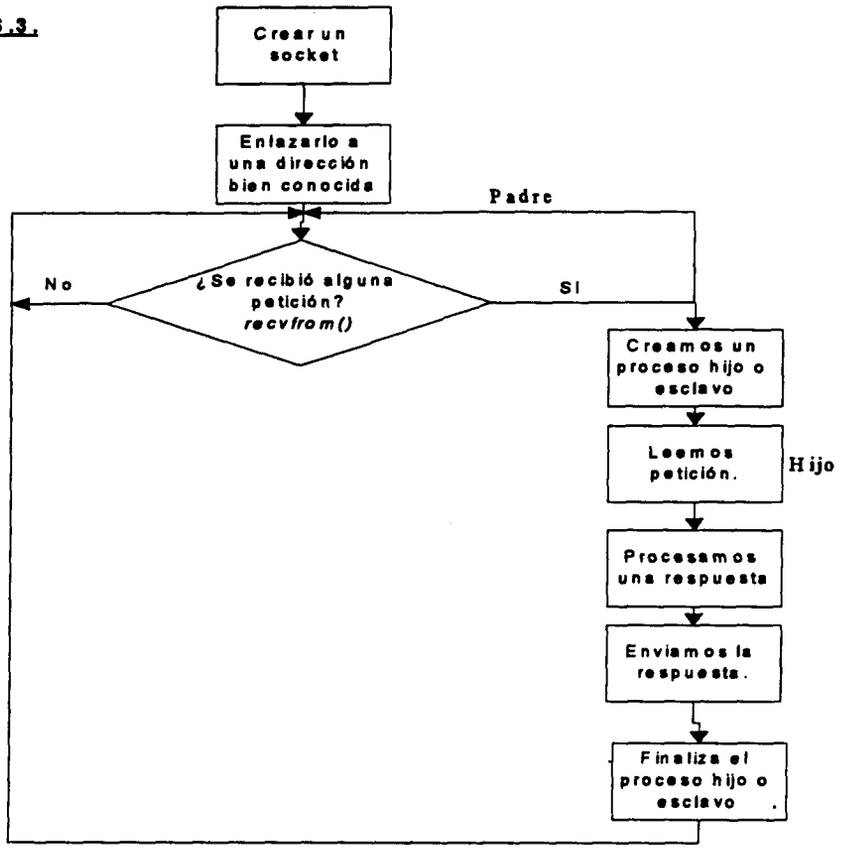
[##] cuando utilizo la expresión no conectado nos referimos a que el socket se creó bajo un modelo no orientado a conexión, es decir, sobre el socket s no se ha realizado ningún connect().

responsabilidad a un esclavo. Cada proceso esclavo maneja la comunicación con un único cliente. Después de que el esclavo haya formado una respuesta adecuada a la petición del cliente y se la envía, éste termina y se destruye. La siguiente sección explicará el concepto de maestro y esclavo con más detalle, y se verá como se aplica ambos enfoques de comunicación orientado a conexión y no orientado a conexión en servidores concurrentes.

6.1.13. Algoritmo de un servidor concurrente no orientado a conexión

Los programadores deben recordar sin embargo el costo exacto de la creación de procesos hijos que puede llegar a abusar de los recursos del sistema operativo y de la arquitectura subyacente, llegando a ser la operación muy cara. En el caso de un protocolo no orientado a conexión se debe considerar bastante bien si el costo de la concurrencia será mayor que la ganancia en velocidad por parte del servidor. De hecho, dado que la creación de procesos es caro, pocos servidores no orientados a conexión gozan de una implementación concurrente. El algoritmo de este tipo de servidor se muestra a continuación.

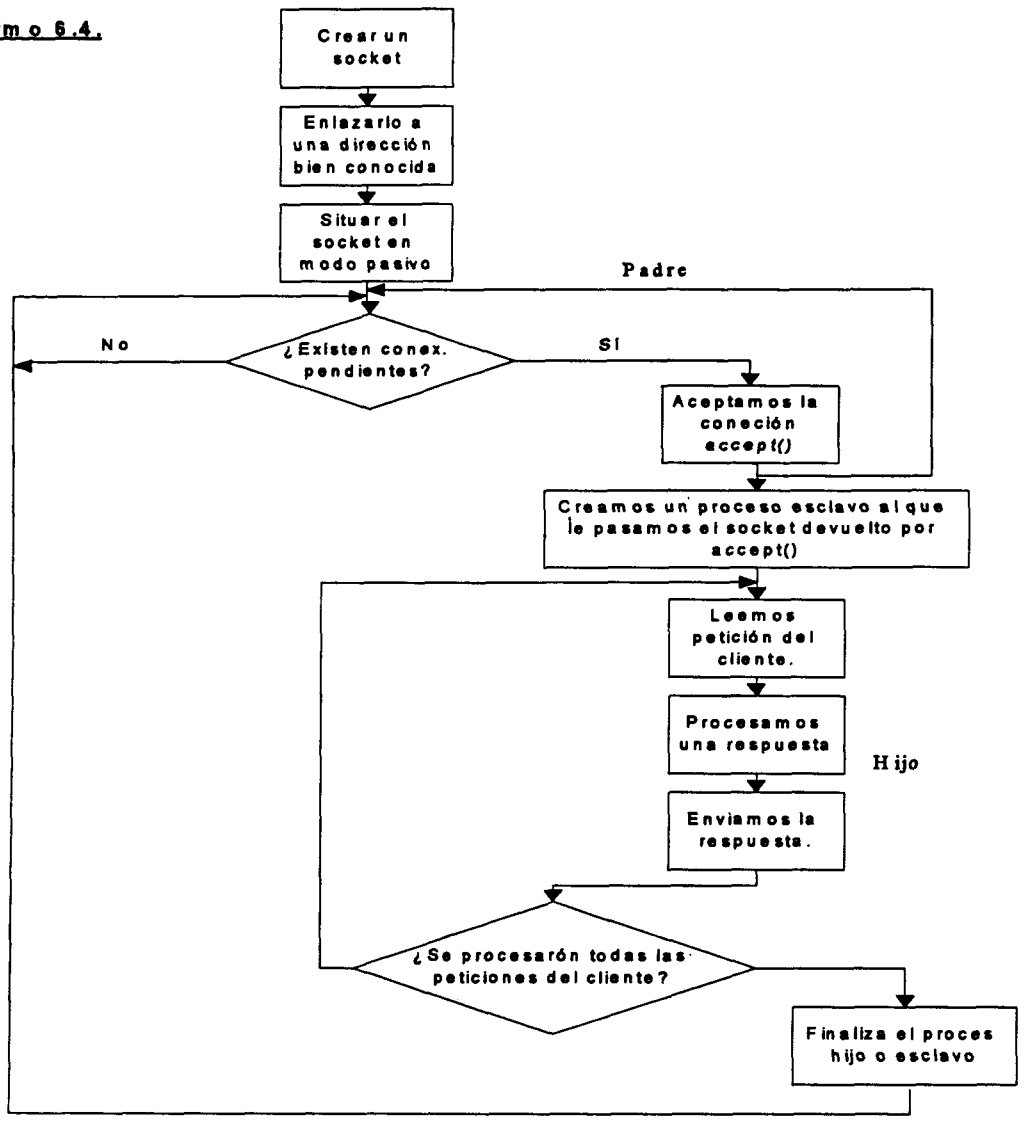
Algoritmo 6.3.



6.1.14. Algoritmo de un servidor concurrente orientado a conexión

Los protocolos de aplicación orientados a conexión usan la "conexión" como el paradigma básico para la comunicación. Permiten a un cliente establecer una conexión con un servidor, comunicarse a través de ella, y después desconectarse. En muchos casos la comunicación entre cliente y servidor maneja más de una petición: el protocolo permite a un cliente el mandar de forma repetida peticiones y recibir respuestas sin necesidad de terminar con la conexión o crear una nueva. Así, los servidores orientados a conexión implementan la concurrencia entre las conexiones más que entre las peticiones individuales. El algoritmo es el siguiente:

Algoritmo 6.4.



6.1.15. Concurrencia aparente usando un único proceso

El algoritmo anterior muestra que el servidor crea un nuevo proceso por cada conexión. En UNIX el proceso servidor maestro hace esto llamando a la sentencia fork(). Veamos ahora como es el algoritmo de una aplicación aparentemente concurrente pero que tan sólo tiene un proceso. Para ello se mantienen una serie de sockets abiertos para la comunicación entre los diferentes clientes. Para averiguar qué socket esta disponible para leer de él una petición o cual de ellos esta listo para poder enviar una respuesta se usa la función de la que ya se habló en un capítulo anterior llamada *select()*.

La forma de proceder es la siguiente. Primero se crea un socket principal cuyo descriptor es S. Este socket lo enlazamos a una dirección bien concida por los clientes. Después, y haciendo uso de la función *select*(), detectamos cuando hay datos esperando para ser leidos por ese socket. Esto se lleva a cabo de la siguiente forma. La función *select*() tiene dos parámetros de entrada que no son mas que dos matrices de descriptores de sockets. Los descriptores de sockets contenidos en cada matriz son aquellos de los que se desea tener noticias de cuando tienen datos para leer y cuando es posible enviar datos por ellos respectivamente. Por tanto y antes de llamar a *select*() debemos crear esas matrices del tipo *fd_set* contemplado en *winsock.h* y debemos incluir el socket S dentro de ella. Después llamamos a *select*() y esperamos hasta que *select*() devuelva el control al programa después de recibir datos por S. Si los datos han sido recibidos por S ejecutamos un *accept*() el cual retorna otro socket S' por el que llevaremos a cabo la comunicación con el cliente. Añadimos S' al conjunto *fd_set* antes creado y volvemos a llamar a *select*(). Si *select*() vuelve a devolver el control al programa entonces miramos si ha sido por la disponibilidad de S o de S'. Si fue por S, volvemos a aceptar la conexión y añadimos el nuevo socket S'' a la estructura *fd_set*. Si la disponibilidad fue detectada en S' entonces es que el primer cliente nos ha enviado una petición y por tanto la tratamos. De esta forma vamos añadiendo descriptores de sockets a la estructura *fd_set*, uno por cada cliente. Es necesario tener en cuenta que cuando el cliente cierra la conexión debemos extraer el descriptor del socket correspondiente de la estructura *fd_set*.

Esta filosofía de programación haciendo uso del select() no se suele utilizar bajo Windows Sockets ya que es mucho más apropiado utilizar métodos asíncronos y programación dirigida por mensajes.

6.1.16. Cuándo usar cada tipo de servidor

Servidores iterativos frente a concurrentes: Los servidores iterativos son más sencillos de diseñar, implementar y mantener pero los servidores concurrentes pueden proporcionar una respuesta más rápida a una petición de un cliente ya que éste no deberá

esperar a que se atiendan los clientes anteriores. Usar una implementación iterativa si el tiempo de proceso de las peticiones es pequeño.

Concurrencia real frente a la aparente: Un único proceso servidor debe manejar múltiples conexiones y usar operaciones asíncronas de entrada salida I/O; un servidor multiproceso permite al sistema operativo que proporcione la concurrencia automáticamente. Se recomienda el uso un único proceso servidor en el caso de que el servidor debe compartir o intercambiar datos entre las diferentes conexiones. Se recomienda usar una solución multi-proceso si cada esclavo puede operar de forma independiente o alcanzar un máximo de concurrencia (como por ejemplo en un miltiprocesador).

Servidores orientados a conexión frente a los no orientados a conexión: Dado que el acceso orientado a conexión significa usar TCP, esto implica fiabilidad en los datos transmitidos. Por el contrario un acceso no orientado a conexión implica que esta fiabilidad desaparezca y se tengan que implementar todos los mecanismo de corrección de errores y seguridad en la aplicación. Se recomienda el uso de una versión no orientada a conexión en aplicaciones que tengan implementados todos los mecanismo de fiabilidad y seguridad que proporciona TCP ó que cuyo ámbito sea una LAN (local area network) donde la fiabilidad es bastante mayor. Se recomienda usar una implementación orientada a conexión cuando el ámbito de nuestra aplicación se encuentre en redes de gran área (WAN). Nunca traslademos una aplicación servidora o cliente no orientada a conexión a una red tipo WAN sin comprobar primero que el protocolo de aplicación controla bien los problemas de fiabilidad y seguridad.

6.1.17. Sumario de los tipos de servidores

- *Iterativo no orientado a conexión.*

Estos son los servidores más comunes no orientados a conexión. Su uso se centra especialmente en aquellas aplicaciones triviales en las que la cantidad de proceso necesario para cada petición de un cliente es relativamente pequeña.

- *Iterativo orientado a conexión.*

Tipo de servidor menos común. Son usados para servicios que requieran poca cantidad de proceso pero que necesitan de una fiabilidad en el transporte que es dada por TCP, siendo un servicio orientado a conexión.

- *Concurrente no orientado a conexión.*

Es un tipo no muy común donde el servidor crea un nuevo proceso esclavo para manejar cada petición. En muchos sistemas, el costo añadido de la creación de procesos es mayor que la ganancia en eficiencia introduciendo la concurrencia.

- *Concurrente orientado a conexión.*

Es el tipo más general de servidor usado debido a la seguridad y fiabilidad que ofrece sobre los datos a través de redes WAN así como la capacidad de atender a varios clientes de forma concurrente. Existen dos implementaciones básicas: la más común usa procesos concurrentes para atender las conexiones; una implementación menos común es la que tan solo usa un único proceso y entradas/salidas asíncronas para manejar las múltiples conexiones. En una implementación de procesos concurrentes, el proceso servidor maestro crea un proceso esclavo para atender cada conexión. Al usar múltiples procesos hace que sea fácil de ejecutar un programa compilado separadamente para cada conexión en vez de escribir todo el código en un único y largo programa.

En la implementación de un único proceso, el proceso servidor atiende múltiples conexiones. Se logra una aparente concurrencia haciendo uso de operaciones de entrada/salida asíncronas. El proceso espera repetidamente por una entrada/salida en cualquiera de las conexiones que tiene abiertas y atiende las peticiones. Dado que un único proceso maneja todas las conexiones, éste puede compartir datos entre ellas. Sin embargo, dado que se trata de un único proceso, éste no puede manejar las peticiones más rápido que lo haría un servidor iterativo, incluso en un ordenador que tuviese múltiples procesadores. Por tanto, la aplicación debe necesitar compartir datos o que el tiempo de proceso sea para cada petición lo suficientemente pequeño para poder justificar este tipo de implementación.

6.1.18. El problema del *deadlock* en los servidores

El *"deadlock"* no es más que la parálisis del servidor o la entrada en un punto muerto debido a imperfecciones en su diseño. Es muy típico que algunos casos o posibles situaciones no se tengan en cuenta a la hora de diseñar el software y después provoquen grandes irregularidades en nuestro programa servidor. Por ejemplo, supongamos que un cliente se conecta a un servidor y no envía ninguna petición al servidor. El servidor estará bloqueado en la llamada *recv()* hasta que se reciban datos. Otro problema que puede suceder es cuando los clientes estan mal diseñados y generan peticiones pero no leen las respuestas. Esto puede llegar a producir que los buffer en el servidor utilizados para almacenar los datos de respuesta al cliente se llenen y produzcan otra situación de bloqueo. En ambos casos, si se trata de una implementación de múltiples procesos tan sólo se verá afectado y bloqueado aquel proceso esclavo encargado de atender a dicho cliente, pero si se trata de una implementación de un único proceso servidor, se bloqueará el servidor por completo.

6.2. Servidores iterativos (UDP)

6.2.1. Introducción

En los apartados anteriores se discutían algunos de los posibles diseños de un servidor, comparando las ventajas y desventajas de cada implementación. En este apartado se dará un ejemplo de una implementación de un servidor iterativo que usa un acceso no orientado a conexión. El ejemplo sigue el algoritmo 6.2. Posteriormente se continuará la discusión proporcionando ejemplos de las demás implementaciones ya comentadas.

6.2.2. Creación de un socket pasivo

Los pasos requeridos para crear un socket pasivo son similares a aquellos requeridos para crear un socket activo. En estos pasos se encuentran muchos detalles y requiere por parte del programa que busque a partir de un nombre de servicio el número del puerto bien conocido.

Para ayudar a simplificar el código del servidor, los programadores deben usar procedimientos para esconder los detalles de la creación de un socket. Como en los ejemplos de los programas clientes, el siguiente ejemplo utiliza dos procedimientos de alto nivel, *pasiveUDP* y *pasiveTCP*, que crean un socket pasivo y lo enlazan al puerto bien conocido del servidor. Cada servidor invoca una de estas funciones, según desee usar un acceso orientado a conexión o no. Ahora consideraremos el caso no orientado a conexión, *pasiveUDP*. Dado que ambos procedimientos tienen mucho en común, ambos llaman a un procedimiento de un nivel más bajo llamado *pasivesock*et para llevar a cabo este trabajo común.

Un servidor no orientado a conexión llama a la función *pasiveUDP* para crear un socket para el servicio que ofrece. Si el servidor necesita usar uno de los puertos reservados para los servicios bien conocidos (por ejemplo un número de puerto inferior a 1024), el proceso servidor debe tener unos privilegios especiales. Cualquier aplicación puede hacer uso de esta función para crear un socket para un servicio no privilegiado. *PassiveUDP* llama a la función passivesock para crear un socket del tipo UDP y retorna el descriptor de dicho socket.

Para hacer más fácil la comprobación y test de los programas cliente y servidor, passivesock localiza todos los puertos sumándoles un entero global llamado portbase. La importancia de esto es que permite que teniendo ya una versión en ejecución, sea posible ejecutar otra con algunas modificaciones y sobre otro puerto para no interferir en la versión que está siendo ejecutada con tan sólo cambiar el valor de portbase. Veamos la función passiveUDP.

```
int passiveUDP (char *service)
{
  /* service: Servicio asociado con el puerto deseado*/
 return passivesock(service, "udp", 0);
}
```

La función passivesock contiene los detalles de la creación de un socket, incluyendo el uso de portbase. Esta función acepta tres argumentos. El primero de ellos especifica el nombre del servicio, el segundo especifica el nombre del protocolo, y el tercero (usado sólo para sockets TCP) especifica la longitud deseada de la cola de peticiones de conexión. *Passivesock* crea un socket en modo datagrama o bien en modo stream, enlaza éste al puerto bien conocido para el servicio, y retorna el descriptor del socket.

Recordar que cuando un servidor enlaza un socket a un puerto bien conocido se debe especificar la dirección mediante la estructura *sockaddr_in*, la cual incluye una dirección IP además de un número de puerto. Passivesock usa la constante INADDR_ANY en vez de especificar un dirección local IP, permitiendo de esta forma que se pueda trabajar en host con una única dirección IP o con hosts *multi-homed* que tiene varias direcciones IP. El uso de esta constante significa que el servidor recibirá comunicaciones direccionadas a el puerto indicado y a cualquiera de las direcciones IP que tenga la máquina.

El código es el siguiente.

```
u_short htons(), ntohs();
u_short portbase=0;
int passivesock (char *service, char *protocol, int qlen)
{
struct servent *pse;
struct protoent *ppe;
struct sockaddr_in sin;
int s,type;

bzero((char *)&sin,sizeof(sin));
sin.sin_family=AF_INET;
sin.sin_addr.s_addr=INADDR_ANY;

/* Mapear el servicio a un número de puerto */
if (pse=getservbyname(service,protocol))
```

```
    sin.sin_port = htons(ntohs((u_short)pse->s_port)+portbase);
else if ((sin.sin_port = htons((u_short)atoi(service))) == 0) {
      wsprinterror("error en la especificación del servicio o número de puerto");
      return -1; }
/* Mapear el protocolo a su entero correspondiente */
if ((ppe=getprotobyname(protocol))==0) {
    wsprinterror("error en la especificación del protocolo");
    return -1;}
/* Usamos el protocolo selecionado para elegir un tipo de socket*/
if (strcmp(protocol,"udp")==0)
  type = SOCK_DGRAM;
else
  type = SOCK_STREAM;
/* Creamos el socket */
s=socket(PF_INET,type,ppe->p_proto);
if (s<0)
    wsprinterror("Fallo en la creación del socket");
    return -1;
/* Enlazamos el socket */
if(bind(s,(struct sockaddr *)&sin,sizeof(sin))<0) {
  wsprinterror ("socket mal enlazado");
  return -1;}
if (type==SOCK_STREAM && listen(s,qlen)<0) {
  wsprinterror("No se puede escuchar a través de este socket");
  return -1;
return s;
}
```

Tal y como se puede observar en el código, la función *passivesock()* acepta un parámetro llamado servicio que es de tipo carácter. En él se almacenará el nombre de un servicio bien conocido o bien un número de puerto en formato texto. Se procede primero llamando a la función *getservbyname()* para ver si el servicio indicado se encuentra o se

reconoce. Si el servicio no es reconocido entonces se debe entender que se ha especificado un número de puerto y no un nombre de servicio y por tanto se procede a la conversión a entero de la cadena de caracteres servicio. Si en la conversión se ve que no era un número se aborta la función y se retorna un valor de error. Por el contrario, si la conversión dio como lugar un valor diferente a cero se continua la ejecución normal de la función.

Veamos ahora el algoritmo de la función passivesock() para tener una idea un poco más general.

La siguiente figura ilustra la estructura de un proceso simple para un servidor iterativo y no orientado a conexión.

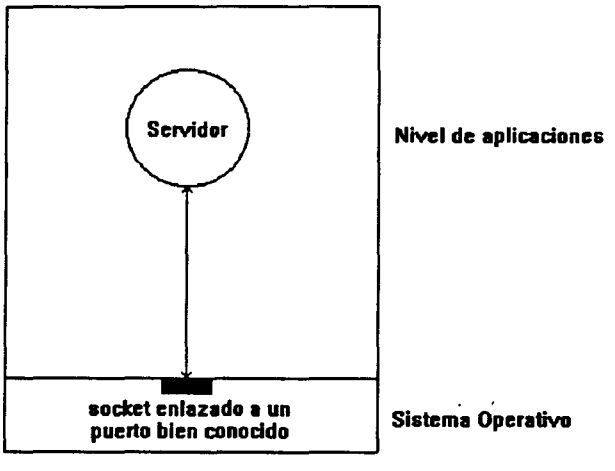


Figura 6.4.

El único proceso es ejecutado siempre. Usa un único socket pasivo que ha sido enlazado a un puerto bien conocido para el servicio que ofrece. El servidor obtiene una petición desde el socket, procesa una respuesta, y envía una respuesta al cliente usando el mismo socket. El servidor utiliza la dirección fuente en la petición como la dirección de destino en su respuesta como es lógico.

6.3. Servidores iterativos orientados a conexión

6.3.1. Introducción

El apartado anterior proporcionó un ejemplo de un servidor iterativo que usaba una conexión por datagramas. Este nuevo apartado muestra un servidor iterativo orientado a conexión que usa el protocolo TCP de la capa de transporte de TCP/IP.

6.3.2. Creando un socket pasivo TCP

En apartados anteriores mencionamos que un servidor orientado a conexión utiliza la función *passiveTCP* para crear un socket del tipo stream y enlazarlo a un puerto bien conocido. *PassiveTCP* acepta dos argumentos. El primero es una cadena de caracteres que especifica el nombre de un servicio el número del puerto que le corresponde. El segundo argumento contiene la longitud deseada de la cola de peticiones de conexiones sobre el socket. Si el primer argumento contiene un nombre de un servicio se debe de estar seguro de que este nombre de servicio se encuentre registrado en la base de datos accedida por la función *getservbyname*, ya que de otra forma se retornará un error. Sin embargo, si el primer argumento es un número en formato texto (ej: "79") será interpretado como un número de puerto y no un nombre de servicio.

La función pasive no reviste grandes misterios y su código es muy similar a su homóloga passiveUDP.

```
int passiveTCP (char *service, int qlen);
{
  return passivesock(service, "tcp", qlen);
}
```

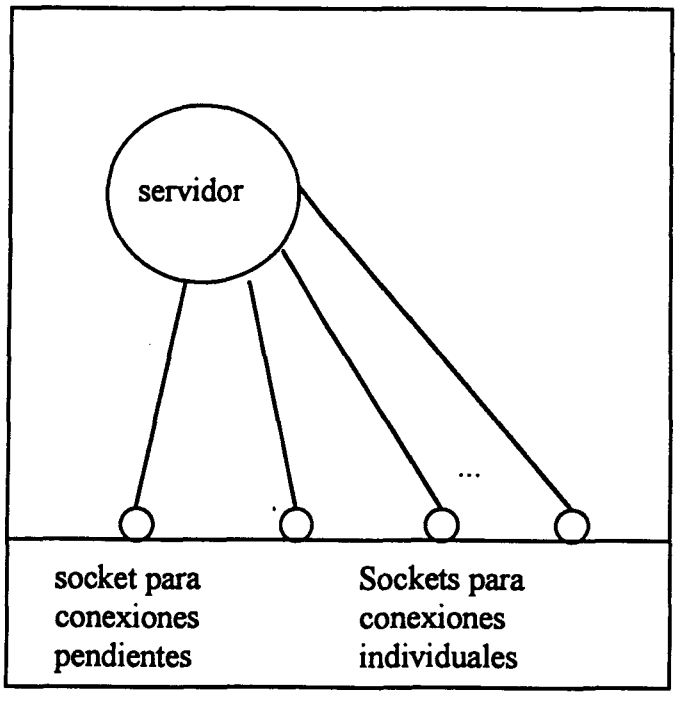
6.4. Servidores concurrentes monoproceso

6.4.1. Introducción

Hasta ahora hemos visto como, gracias a las facilidades del sistema operativo para crear procesos separados, la implementación de la concurrencia no fue muy complicado. Ahora veremos una idea bastante interesante a la hora del diseño que no es muy obvia: se verá cómo un servidor puede ofrecer una aparente concurrencia a los clientes usando un único proceso. Veamos por tanto ya la estructura de un proceso de este tipo.

6.4.2. Estructura de un servidor monoproceso TCP

La siguiente figura muestra la estructura de una forma muy general.



Como se puede ver en la figura, el servidor consta tan sólo de un único proceso desde el cual se abre un socket de escucha y a medida que llegan las peticiones de los clientes se va abriendo los sockets respectivos para atenderlos.

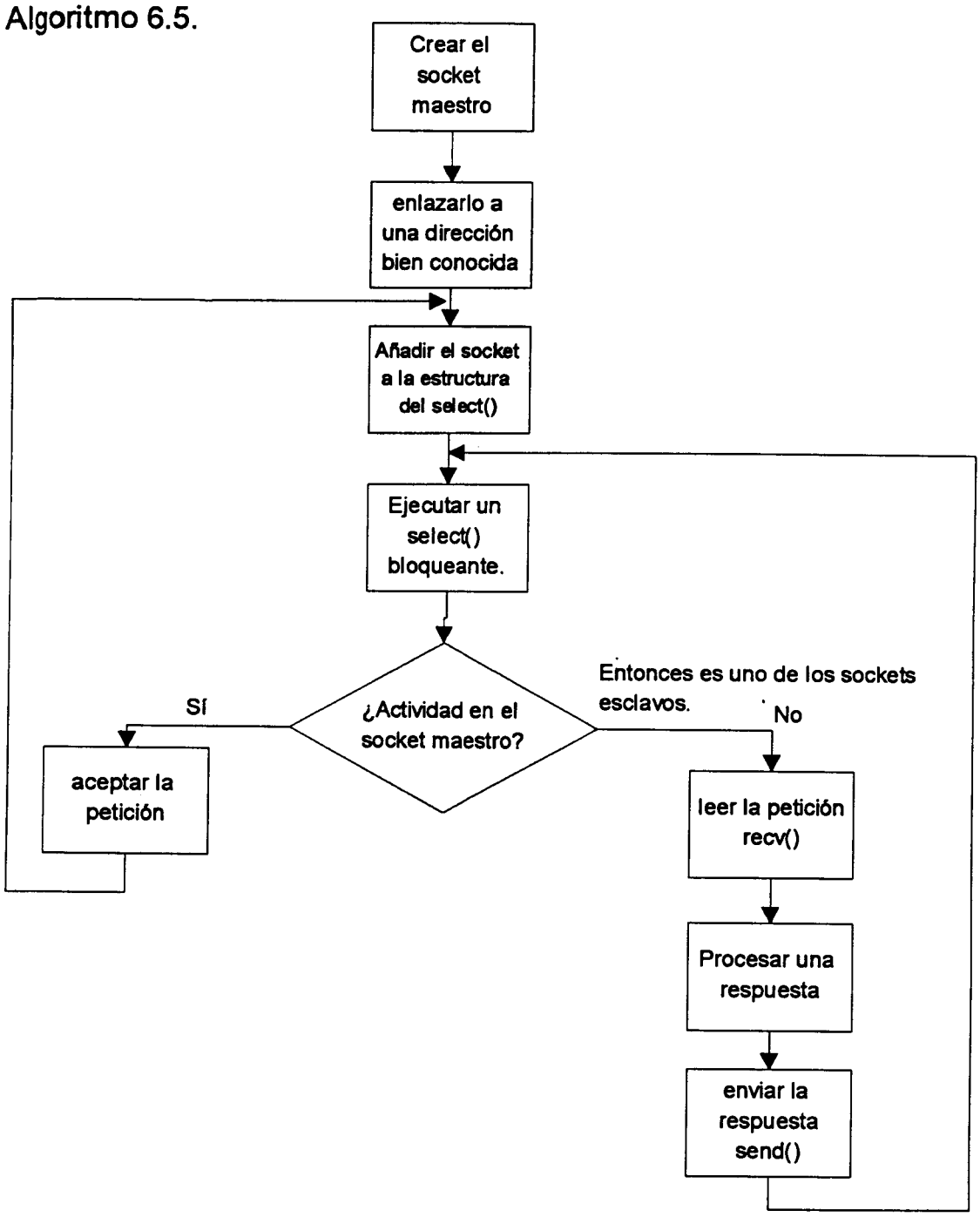
En esencia, un único proceso debe proporcionar las funciones de un proceso maestro y los esclavos. Por tanto, el proceso mantendrá abiertos una serie de sockets, con

uno de ellos enlazado a un puerto bien conocido, por el cual aceptará las conexiones. Los otros sockets corresponden a las conexiones sobre las cuales un proceso esclavo debería trabajar. En este diseño se utiliza la función *select()* . Esta función es capaz de devolver al programa el *status* de uno o más sockets, de tal forma que se puede averiguar si sobre un socket determinado existe la posibilidad de enviar o de recibir datos. Por tanto, nuestro programa servidor pasa a la función el conjunto de sockets como un argumento y espera la actividad en cualquiera de ellos. Cuando la función *select()* devuelve una respuesta, ésta pasa al programa una estructura donde se reflejan los sockets sobre los que es posible realizar una lectura o una escritura. Gracias a esta función, nuestro único proceso servidor puede advertir la llegada de peticiones por parte de los clientes o de datos de una conexión con un cliente.

Para distinguir operaciones de un esclavo o de un maestro, los servidores monoproceso utilizan el descriptor del socket. Si el descriptor corresponde al socket del maestro el servidor entonces lleva a cabo las acciones oportunas: llamaría a la función *accept()* en el socket para obtener una nueva conexión. Si el descriptor corresponde a un socket esclavo, el servidor lleva a cabo las operaciones correspondientes: llamará a *recv()* para obtener la petición y después la responderá con un *send()*.

El algoritmo de éste tipo de servidores se muestra a continuación:

Algoritmo 6.5.



6.5.6. Servidores multiprotocolo concurrentes

Ya se ha hablado de servidores de un único proceso e iterativos, pero si el servicio lo requiere, la idea se puede extender a un enfoque concurrente. Recordemos que para servicios cuyo tiempo de proceso de una respuesta era relativamente pequeño, una implementación iterativa era la adecuada, pero en aquellos servicios que necesitasen de un tiempo considerable de proceso de respuesta, vimos que el diseño más acertado era uno concurrente.

La verdad es que la idea de un servidor concurrente y multiprotocolo ya es algo complicada. En un diseño de este tipo participan varios algoritmos de los ya vistos. Habría que analizar si tratamos de forma concurrente los servicios proporcionados vía TCP o si por el contrario lo hacemos de forma concurrentemente en ambos protocolos, el TCP y el UDP. No es mi propósito el profundizar mucho en el diseño exacto sino dar una visión global de las posibles soluciones que podemos adoptar a la hora de construir software servidor.

6.6. Servidores Multiservicio

6.6.1. Introducción

Ya hemos visto como construir un servidor monoproceso que utiliza operaciones I/O asíncronas para aparentar una concurrencia entre varias conexiones y como servidores multiprotocolos suministran un servicio vía UDP y TCP. En este nuevo apartado combinaremos varios de los tipos de servidores vistos hasta ahora analizando sus implementaciones.

6.6.2. Servidores

En muchos de los casos, los programadores diseñan un servidor por servicio. Estos servidores esperan en un puerto bien conocido por peticiones para un único servicio. Por ejemplo, existe un servidor para el servicio DAYTIME, otro para el servicio ECHO,... Anteriormente vimos como una aproximación multiprotocolo ayudaba a conservar los recursos del sistema mejor. Las mismas ventajas que motivaron a los servidores

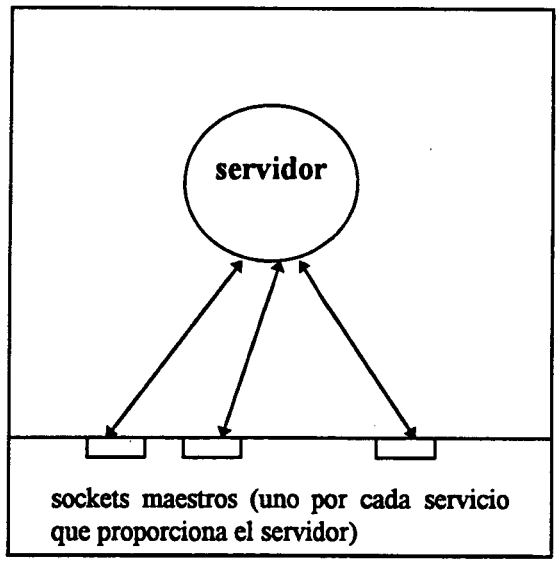
multiprotocolo han motivado la aparición de los servidores multiservicio, consolidando varios servicios sobre un mismo servidor.

Para darse cuenta del coste de la creación de un servidor por cada servicio que define TCP/IP pensemos en el número de procesos servidores que se deberían estar ejecutando en una máquina y muchos de estos procesos quizás no reciban una petición por parte de un cliente nunca. El englobar varios servicios en un único proceso servidor reduciría en gran medida el número de procesos abiertos en un máquina y por tanto reduciría el consumo de los recursos de dicha máquina.

Otra razón para englobar varios servicios en un único servidor se debe a que existen muchos servicios triviales (tales como el TIME, DAYTIME, ECHO,...) en los cuales la mayor parte del código se la lleva el tema del control de los sockets y no el proceso del servicio en sí. Cómo el control de los sockets es igual independientemente del servicio, el englobar todos estos servicios en un único servidor reduciría el código bastante.

6.6.3. Diseño Multiservicio no orientado a conexión

Los servidores multiservicio pueden usar ambos tipos de protocolos de transporte, el orientado a conexión y el no orientado a conexión. La siguiente figura ilustra una posible estructura para un servidor multiservicio UDP.



Como se muestra en la figura, un servidor iterativo, multiservicio y no orientado a conexión normalmente consiste en un único proceso que contiene todo el código necesario para suministrar los servicios. El servidor abre un conjunto de sockets UDP y los enlaza cada uno a su puerto bien conocido correspondiente según el servicio. Se usa entonces una pequeña tabla para mapear los sockets a los servicios. Para cada descriptor, la tabla graba la dirección de un procedimiento que computa el servicio ofrecido por cada socket. El servidor utiliza la función *select()* para esperar por la llegada de alguna petición sobre los sockets anteriormente creados.

Cuando llega un datagrama, el servidor llama al procedimiento apropiado para computar y enviar la respuesta.

6.6.4. Diseño Multiservicio orientado a conexión

Un servidor orientado a conexión y multiservicio bien puede seguir el algoritmo de uno iterativo. En principio, estos servidores desarrollan las mismas tareas que los servidores iterativos y orientados a conexión ya vistos. Para ser más precisos, el único proceso de un servidor multiservicio reemplaza a los procesos servidores maestros de un conjunto de servidores orientados a conexión.

Cuando se inicia la ejecución del código, el servidor multiservicio crea un socket para cada servicio que ofrece, los enlaza a sus puertos bien conocidos, y utiliza la función *select()* para esperar por una conexión entrante en alguno de los sockets creados. Cuando alguno de los sockets está listo para leer, o sea ha llegado una petición de conexión por parte de un cliente, el servidor llama a la función *accept()* para obtener una nueva conexión por donde atender al cliente. *Accept()* crea un nuevo socket. El servidor utiliza este socket creado por *accept()* para "hablar" con el cliente. Cuando termina de hablar con él procede a cerrar dicho socket. Así, detrás de cada socket maestro para cada servicio, el servidor tiene al menos otro socket adicional abierto en cualquier momento.

Como en el caso no orientado a conexión, el servidor mantiene una tabla de mapeos de forma que pueda decidir cómo tratar cada conexión entrante. Cuando un servidor comienza, crea los sockets maestros. Para cada socket maestro, el servidor añade una

entrada a la tabla de mapeados que especifica el número de socket y el procedimiento que implementa el servicio ofrecido por ese socket. Después de que haya alojado un socket maestro para cada servicio, el servidor llama a *select()* para esperar por una conexión. Desde que llega una conexión, el servidor utiliza la tabla para decidir cual de los procedimientos internos será llamado y por tanto suministrará el servicio deseado al cliente.

6.6.5. Servidor multiservicio orientado a conexión y concurrente

El procedimiento llamado por un servidor multiservicio cuando llega una petición de conexión puede aceptar y manejar la nueva conexión directamente (haciendo de esta forma el servidor iterativo), o puede crear un proceso esclavo para tratar dicha petición (haciendo al servidor concurrente). De echo, un servidor multiservicio puede elegir en tratar algunos servicios de forma iterativa y otros de forma concurrente; el programador no necesita elegir un único estilo para todos los servicios.

En una implementación iterativa, desde que finaliza un procedimiento su comunicación con el cliente, éste cierra la nueva conexión. En el caso concurrente, el proceso servidor maestro cierra la conexión tan pronto como se haya creado el proceso esclavo; la conexión permanece abierta con el proceso esclavo.

6.6.6. Diseño de un servidor multiservicio monoproceso

Esto es posible, aunque no muy común, manejar toda la actividad en un servidor multiproceso mediante un único proceso, utilizando un diseño tal y como se ha explicado ya. En estos diseños, en vez de crear un proceso esclavo para cada conexión, un único proceso servidor crea los sockets para cada conexión y los añade al conjunto de ellos visto por la función *select()*. Si uno de los socket maestros esta listo, el único proceso servidor llama a *accept()*; si uno de los sockets esclavos esta listo, conforma una respuesta, y llama a *send()* para enviar una respuesta al cliente.

Capítulo 7

El mundo WINDOWS

7.1. Introducción

¿Quién no conoce Windows? Todo el mundo que haya trabajado con un ordenador personal alguna que otra vez ha oído hablar de Windows, pero ¿qué es Windows? Algunos autores lo definen como un sistema operativo que a la larga intentará sustituir por completo al MS-DOS, otros lo definen como un extensión gráfica del DOS; en realidad lo menos importante para nosotros es el buscarle una definición. Más bien nos interesa comprender como trabaja, qué diferencias tiene respecto al DOS y cómo se programa en este entorno.

7.2. Multiproceso ó Multitarea

La primera gran diferencia con respecto al DOS es que Windows puede soportar varias aplicaciones ejecutándose a la vez. No nos confundamos y pensemos que Windows es multiproceso y que permite la ejecución paralela de varios procesos diferentes. Windows lo único que permite es el tener varios procesos abiertos a la vez, pero en realidad dado un instante de tiempo tan sólo se está ejecutando un único proceso. Windows es por tanto Multitarea pero no multiproceso.

7.3. Interfaz de usuario

Otra ventaja que tiene Windows respecto al DOS es que ofrece una interfaz de usuario estándar. Esto quiere decir que la forma de interactuar por parte del usuario con los programas Windows es igual e independiente de la aplicación que se ejecute. Sin embargo en el DOS, para cada programa el usuario debe aprender un conjunto de órdenes diferentes.

En la siguiente figura se muestra el entorno Windows, en el cual aparecen una serie de ventanas dentro de las cuales están dispuestos nuestros programas de aplicación.

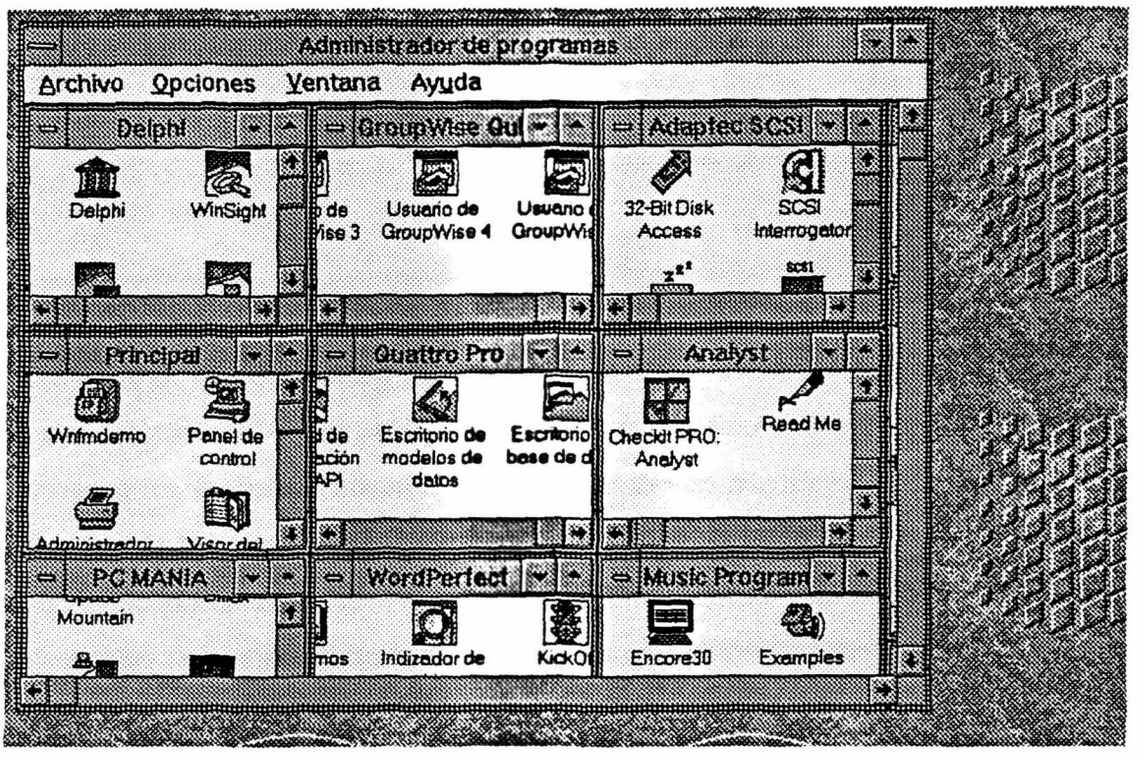


Figura 7.1.

7.4. Mensajes

Otra gran diferencia que ofrece Windows respecto al MS-DOS es que los programas de aplicación no se controlan de una forma secuencial sino que están conducidos por sucesos. Es decir, dependiendo de lo que ocurra se ejecutará una parte del programa u otra. Un suceso bien puede ser el pulsar un botón del ratón o pulsar una tecla. Todos estos sucesos Windows los dirige a la aplicación por medio de mensajes. Nuestra aplicación por tanto tan sólo debe limitarse a tratar mensajes. Esta es la gran filosofía de la programación en Windows.

7.5. ObjectWindows y la API

Por otro lado, la programación en Windows se puede analizar desde dos puntos de vista. Uno mirando al estilo antiguo de programación haciendo uso de la API (Application Program Interface) o bien haciendo uso de ObjetWindows que no es más que una biblioteca de clases que permiten programar en Windows partiendo de la programación orientada a objetos. Quizás el método más adecuado sea el segundo debido a la característica de Windows respecto a que es un sistema gobernado y conducido por sucesos. También es posible realizar un programa orientado a objetos haciendo uso de la API sin necesidad de utilizar ObjectWindows. En esta pequeña introducción se darán unas ideas de ambas formas de programar en Windows aunque se profundizará un poco más en la programación orientada a objetos haciendo uso de ObjectWindows debido a la novedad y la gran aceptación que tiene entre los principiantes debido a su sencillez en el código.

7.6. Un ejemplo API

Para entender la filosofía de la programación haciendo uso de la API procederemos con un ejemplo sencillo. Advertir que en la programación con ObjectWindows se pueden usar las funciones de la API, aunque el objetivo principal de ObjectWindows es ocultar al programador el API de Windows.

7.6.1. Los ficheros de cabecera

En nuestros programas en C tendremos como primeras líneas los ficheros llamados de inclusión ó cabeceros. En éstos ficheros se definen todos los tipos de datos que se van a usar y los prototipos de las funciones. Los prototipos de las funciones no son más que las declaraciones de las funciones definiendo el valor que retornan y sus parámetros. Respecto a los tipos de datos que se usan en Windows hay que ser bastante estrictos y usar los que corresponden. En Windows existen muchos tipos de datos creados, con su nombre especial. Es decir, en el fichero de inclusión podemos tener definido el tipo HWND como un entero. En nuestro programa podemos crear una variable del tipo entero o bien declararla del tipo

HWND y funcionará igual de bien en ambos casos. De todas formas se aconseja que se use el tipo HWND y que no se declare como un entero para asegurar la portabilidad y compatibilidad del código con futuras versiones; en versiones futuras de Windows que sean más potentes pueden definir al tipo HWND como un entero largo y no como un entero. Si en nuestro programa hubiésemos definido nuestras variables de ese tipo como HWND este cambio se solucionaría incluyendo el nuevo fichero de inclusión de la nueva versión de Windows donde se definiría el tipo HWND como un entero largo, sin tener que retocar el programa. De la otra manera, tendríamos que acudir a nuestro código y redefinir a mano todas las definiciones de variables realizadas con los tipos de datos que han cambiado de una versión a otra del Windows.

Por tanto, nuestro principal fichero de inclusión es:

#include <windows.h>

7.6.2. Prototipos de las funciones

Continuando con la escritura de nuestro programa definimos el prototipo de nuestra función de ventana así como los prototipos de las funciones que diseñemos en nuestro programa. Esta función (el procedimiento de ventana) será la encargada de tratar todos los eventos que sucedan en dicha ventana. Será el procedimiento que maneje y controle la ejecución del programa.

long FAR PASCAL **WndProc** (HWND, unsigned, WORD, LONG);

La primera palabra **long** indica el tipo de dato que devuelve la función. Las dos siguientes palabras FAR PASCAL indican que la referencia a la función es FAR y que se tratará su acceso de manera análoga a como se hace en Pascal. Por último el resto es el nombre de la función y sus parámetros.

7.6.3. La función WinMain()

Ahora se va a declarar una función muy importante y necesaria en todo programa. Su significado es semejante al de la función main() en un programa normal de C. Su nombre en Windows es WinMain. En ella se definen una serie de parámetros que ahora se explicarán.

int PASCAL **WinMain** (HANDLE hInstance, HANDLE hPrevInstance,

LPSTR lpszCmdLine, int nCmdShow)

El primer parámetro retorna un valor del tipo HANDLE (Manejador) que no es más que un número que identifica el número de copia del programa. Es decir, se puede ejecutar varias veces el reloj de Windows y tener varias copias de éste abiertas a la vez. Pues para poder identificar a cada copia cada vez que se ejecuta una nueva el sistema retorna un HANDLE de la copia. Este HANDLE identifica a cada copia en ejecución del mismo programa de forma unívoca. El siguiente parámetro que nos retorna es el HANDLE de la copia anterior si es que existe alguna copia del mismo programa ya abierta antes de ejecutar ésta. Los otros dos parámetros más de momento no son muy importantes. En lo que si hay que hacer hincapié es en que siempre que se haga un programa Windows se debe crear esta función WinMain y los parámetros deben ser siempre éstos. El programador no puede jugar con estos parámetros y poner ni quitar ninguno.

7.6.4. La clase de ventana

Lo que sigue a continuación es bastante importante también. Es el tema de las ventanas. Como bien es sabido por todos la ventana en Windows es la pieza fundamental. Una ventana se puede considerar como un objeto con unas características determinadas. Pues bien, las características de la ventana se deben definir ahora haciendo uso de una estructura de datos llamada WNDCLASS y que está definida en el fichero *Windows.h*. Definimos por tanto una variable llamada *claseVentana* del tipo WNDCLASS.

WNDCLASS claseVentana;

7.6.5. ¿Dónde se reciben los mensajes?

Otra variable que debemos definir ahora y también muy importante en Windows es una variable del tipo **MSG**. O sea, una variable donde se puedan almacenar los mensajes recibidos por la aplicación. El tipo MSG es una estructura que tiene dos campos muy importantes llamados *Wparam* y *Lparam*. Dependiendo del tipo de mensaje recibido en cada uno de éstos campos se almacenará información detallada del suceso que ha provocado dicho mensaje. Ya por último definimos la variable *hWnd* (Handle Window ó manejador de ventana). Esta variable va a hacer que después de crear una ventana, mediante el valor guardado en esta variable podamos hacer referencia a dicha ventana. Es decir, si queremos mandar un mensaje a una ventana la forma de decirle a Windows a qué ventana es mediante su identificador o manejador de ventana correspondiente.

```
MSG     msg;
HWND hWnd;
```

Como hemos dicho antes, antes de poder crear una ventana es necesario definir algunos parámetros asociados con dicho tipo de ventana. Para ello hacemos uso de la variable del tipo WNDCLASS. Un detalle a tener en cuenta es que Windows mantiene una única copia en memoria de algunas variables que pueden ser usadas de forma conjunta por varias copias de la misma aplicación en ejecución, con el fin de ahorrar memoria. Una de estas variables son las del tipo WNDCLASS. Sólo es necesario definir las características de la ventana en la primera instancia que se ejecuta y no se debe definir en ejemplarizaciones sucesivas debido a que las características de la ventana se copian en una zona de memoria a la que después acceden el resto de ejemplares en ejecución. Es por ello que en la siguiente línea de código exista una sentencia condicional para ver si existe alguna instancia previa en ejecución.

```
if (!hPrevInstance)
    {
claseVentana.style = CS_HREDRAW

        . // Se definen el resto de campos de la estructura
```

```
// Una vez completada la estructura es necesario registrarla para
// que Windows tome nota de ella.
// Si la función para registrarla falla se retorna un valor de Falso.
if (!RegisterClass (&claseVentana))
return FALSE;
} // Fin de la sentencia IF
```

7.6.6. Creación de una ventana

Ya hemos registrado la clase de ventana en la cual hemos definido algunas características de la ventana. Nos resta ahora crear la ventana con esas características. Para ello hacemos uso de la función de la API llamada **CreateWindow**. Esta función retorna un manejador de la ventana que almacenaremos en la variable que hemos definido antes con tal propósito. No entraremos en detalles sobre los parámetros que acepta tal función y por tanto serán omitidos en el código. Por último es necesario mostrar la ventana. Cuando se crea no se dibuja en la pantalla ni mucho menos. Es necesario llamar a la función **ShowWindow** a la cual, como uno de sus parámetros le pasamos el manejador de la ventana para decirle qué ventana debe mostrar. Finalmente se llama a una función **UpdateWindow** que actualiza la ventana; por ejemplo tras moverla en la pantalla.

```
hWnd = CreateWindow (..... );
ShowWindow (hWnd,nCmdShow);
UpdateWindow (hWnd);
```

7.6.7. El bucle de mensajes

Llegamos ahora a uno de los puntos claves de Windows, el bucle de mensajes. Como ya se comentó antes, en Windows el flujo del programa es controlado por los mensajes. No existe un orden de secuencia preestablecido a la hora de ejecutar las sentencias, sino que dependiendo de los mensajes que Windows despache a la aplicación, se ejecutará una acción u otra. Cada aplicación tiene una cola de mensajes asociada. El

bucle de mensajes va despachando los mensajes de esta cola según van llegando. Pero veamos la forma del bucle de mensajes.

```
while (GetMessage (&msg, NULL, 0 ,0))
    {
TranslateMessage (&msg);
DispatchMessage (&msg);
    }
```

La función **GetMessage** retorna un valor true mientras haya mensajes en la cola. Después y dentro del bucle, la función **TranslateMessage** traduce los códigos del teclado y los códigos de caracteres a un código estandar ANSI. Y la función importante es **DispatchMessage** que es la encarga de despachar el mensaje al procedimiento de ventana correspondiente.

Por último se añade en el final de nuestra función WinMain la siguiente línea que hace retornar el valor guardado en wParam del último mensaje recibido.

```
return msg.wParam;
} // Fin de WinMain
```

7.7. El procedimiento de ventana

Ahora procedemos a definir nuestro procedimiento de ventana en el cual se trataran todos los mensajes que reciba nuestra ventana.

```
long FAR PASCAL WndProc (HWND hWnd,unsigned mensaje,
                         WORD wParam, LONG lParam)
{
/* Variables locales si existen */
```

Ahora viene lo interesante del tema. Mediante una sentencia *switch* según el mensaje recibido en la variable mensaje se tomarán unas acciones u otras en nuestro procedimiento de ventana.

```
switch (mensaje)
    {
case WM_PAINT:      //Este mensaje se recibe cada vez que se debe
                    //volver a dibujar la ventana porque se haya
                    //movido o superpuesto otra encima,...
return 0;
    case WM_DESTROY:      //Se recibe cuando se cierra la ventana.
        PostQuitMessage(0); //Provoca una salida NULL en la función
                            //GetMessage y por tanto se rompe el bucle de
                            //mensajes y termina la aplicación.
return 0;
} // Fin de la sentencia switch.
```

Como se puede apreciar en este procedimiento de ventana tan sólo se tratan dos mensajes, el WM_PAINT y el WM_DESTROY. Pero existen muchos otros mensajes que deben ser tratados y que el programador puede olvidarse de ellos dándole la responsabilidad de su tratamiento a una función también bastante importante llamada **DefWindowProc**. Esta función trata todos los mensajes que no hayamos tratado nosotros en nuestra sentencia switch.

```
    DefWindowProc (hWnd, mensaje, wParam, lParam);
} // Fin de WinProc.
```

Resumiendo, el programador dota a la ventana creada de un comportamiento particular mediante el tratamiento que le da a los mensajes que recibe dicha ventana. Este programa tan sólo dibuja una ventana en Windows y permite que se mueva, ampliarla, minimizarla y cambiarle el tamaño.

Esta claro que las aplicaciones Windows van más allá de este simple ejemplo. También es posible generar menús desplegables y cajas de diálogo para introducir datos o

mostrar mensajes informativos en la ventana. El diseño de menús y el de las cajas de diálogo se hacen en otro fichero diferente al del código. Este fichero se llama de recursos y su extensión es .RC. En él definimos iconos, menús, cajas de diálogo, cursores,... Para el diseño de todo esto es bastante recomendable el hacer uso de la utilidad que Borland C++ dispone para este fin. Los detalles a la hora de generar menús y cajas de diálogo no se verán aquí ya que la finalidad de esta introducción es la de aclarar el concepto de la programación bajo Windows que se basa en mensajes.

7.8. Programación Orientada a Objetos

7.8.1. Introducción

Este tipo de programación se basa en el concepto de objeto. *Un objeto es un ente lógico donde además de incluirse tipos de datos, variables, registros, etc... se permite definir funciones o métodos que gobernarán al objeto.* Éstos métodos serán los encargados de dar vida y caracterizar el comportamiento del objeto.

Objeto = Datos + Métodos

El lenguaje C++ representa el resultado de los esfuerzos realizados para proporcionar las ventajas de la programación orientada a objetos a un lenguaje clásico, muy extendido en la programación de sistemas. Se trata de una extensión del lenguaje C que representa claras influencias del lenguaje Simula, que a su vez puede considerarse como el precursor de los lenguajes OOP (Oriented Object Programming).

7.8.2. Clases

El lenguaje C++ entiende por **clase** un tipo de datos definido por el programador que contiene toda la información necesaria para construir un objeto de dicho tipo y el conjunto de operaciones que permiten manejarlo, los métodos. Las clases de C++ le permiten disfrutar de los beneficios de la modularidad.

La construcción de las clases se hace de tal manera que se asegura el encapsulamiento de los objetos. El lenguaje C permite utilizar los términos **union** (unión) y **struct** (estructura) para declarar una estructura de datos. Estos identificadores siguen siendo válidos en C++. De hecho, la construcción de clases se puede realizar con ellos, aunque los programadores suelen preferir el nuevo identificador **class**. Ambas opciones permiten definir una clase como un conjunto de datos o atributos más una serie de métodos. En la nomenclatura del C++, cada uno de los componentes de una clase se denomina **miembro**. Los datos son *data-members* y los métodos *function-members*.

7.8.2.1. Niveles de acceso a las clases

La forma en la que C++ incorpora la ocultación de la información está ligada a la definición de las clases o estructuras. Para entender las diferencias entre las tres alternativas es preciso explicar previamente otra innovación que incorpora el C++; los **niveles de acceso**. El acceso a cada uno de los miembros (atributos o métodos) de una clase o estructura puede pertenecer a uno de los siguientes niveles:

- **Público (public).** Un miembro público puede utilizarse en cualquier lugar donde se tenga acceso a la clase.
- **Privado (privado).** Un miembro privado sólo puede utilizarse en métodos declarados dentro de la misma clase.
- **Protegido (protected).** Un miembro protegido puede utilizarse en métodos declarados dentro de la misma clase y en métodos de clases descendientes de ellas.

En la definición de una clase, los niveles de acceso pueden especificarse de forma explícita o implícita. El nivel por defecto es diferente, en este último caso, según se empleen las declaraciones **class**, **union** o **struct**.

- En una clase definida con **struct** todos los miembros son, por defecto, públicos.
- En una clase definida con **union** todos los miembros son públicos y este nivel de acceso no puede modificarse.
- En una clase definida con **class** todos sus miembros son, por defecto, privados.

7.8.2.2. Un Ejemplo

Vamos a poner un ejemplo. Supongamos que deseamos construir la clase **Punto**, que defina el concepto de un punto en dos dimensiones. Dicha clase deberá contener información sobre la ordenada y la abcisa del punto y el color con que se va a representar. En cuanto a los métodos, necesitaremos una función que nos dé la ordenada del punto, otra que nos diga el valor de la abcisa y otra que nos dé el color. También necesitaremos una que nos dibuje el punto en la pantalla, otra que lo mueva a coordenadas nuevas y otra que le cambie el color. La implementación podría realizarse así:

```
struct Punto {
protected:
   int X, Y;
   int Color;
public:
   int DameX() {return X;}
   int DameY() {return Y;}
   int DameColor() {return Color;}
   void PonColor (int NuevoColor);
   void Mueve (int NuevoX, int NuevoY);
   void Dibuja (void);
};
....
....
// Definición de las funciones miembro.
Punto::PonColor(int NuevoColor)
{
   Color = NuevoColor;
};
Punto::Mueve(int NuevoX, int NuevoY)
{      X = NuevoX;
       Y = NuevoY;
};
```

```
Punto::Dibuja(void)
{
   putpixel (X,Y,Color);
};
```

En el listado anterior se observará que hemos definido la clase **Punto** empleando el identificador **struct**. También podríamos haberlo definido con **class**, puesto que hemos especificado los accesos de forma explícita. Puesto que hemos utilizado **struct**, no habría hecho falta declarar los miembros públicos como tales, ya que lo son por defecto. Por otro lado, si hubiéramos utilizado el término **class**, sí hubiera sido necesario especificar los públicos pero no los privados.

Otra innovación que presenta el C++ es la posibilidad de definir funciones asociadas a una estructura. En el ejemplo vemos que la clase **Punto** contiene métodos.

7.8.3. La herencia

Un concepto que es bastante importante en la OOP (Programación Orientada a Objetos) es la de la herencia. Con ella podremos heredar las características de un objeto ya existente. Por poner un ejemplo práctico, supongamos un objeto llamado vehículo. En él podemos tener variables que almacenen el número de ruedas así como la matrícula de un vehículo, y un par de métodos o funciones que sirvan para poder introducir estos datos. Podemos más adelante crear otro objeto llamado coches y otro llamado camiones. En ambos objetos los datos del número de ruedas y matrícula se pueden omitir su declaración si especificamos que estos dos objetos nuevos hereden las características del objeto vehículo. Es decir, podemos construir una estructura jerárquica donde la raíz es un objeto muy general y a partir de él vamos particularizando según sea necesario, heredando las características del objeto inmediato superior. Por tanto, mediante la herencia un objeto puede adquirir las propiedades de otro.

Como ya hemos dicho, un objeto se compone de datos y métodos. Cuando en Pascal queremos crear una variable tipo registro primero debemos definir el tipo registro y después declarar la variable de ese tipo. Aquí sucede algo similar. Primero debemos definir la clase, que viene a ser como el tipo del objeto, y después declaramos un objeto de esa clase. Por tanto, dentro de la definición de la clase es donde vamos a definir todos los datos y métodos que tendrán los objetos más tarde declarados bajo esa clase. Veamos otro ejemplo:

```
class cola {
   int q[100];   // Datos privados de la clase.
   int sloc,rloc;
public: //Definición de métodos públicos.
   void init();
   void qput();
   void qget();
   friend int amiga(); //Método o función amiga.
};
// Declaración e implementación de los métodos de la clase.


// Creación de un objeto llamado prueba del tipo cola.
cola prueba;
```

En este ejemplo existen varias palabras extrañas y falta la implementación de los métodos de la clase cola. Como se puede observar la clase no es más que una estructura donde se engloban datos y funciones. Analicemos otra vez las palabras claves private y public ya que su comprensión es bastante importante. Gracias a estas dos palabras se va a poder gobernar dentro de una clase que procedimientos podrán ser invocados desde otra clase y qué variables podrán ser accedidas por métodos de otras clases. Por defecto en una clase definida con la palabra class todo es privado, es decir, si no se dice lo contrario las variables definidas sólo podrán ser accedidas por los métodos propios de la misma clase y los métodos de la clase no se podrán heredar y no podrán ser invocados desde otra clase.

Existe otra palabra clave (protected) que permite que los métodos puedan ser heredados pero no puedan ser invocados desde otra clase.

Otra palabra clave que se encuentra en este código es la palabra *friend*. Mediante esta palabra declaramos a una función que no es de la clase como una función amiga. De esta forma, esta función amiga puede acceder a los datos privados de la clase donde ha sido definida como amiga.

Resumiendo, el formato general de definición de una clase es:

```
class nombre_de_la_clase {
   //datos y funciones privadas
public:
   //datos y funciones públicas
} lista_nombre_objetos_de_esta_clase;
```

La lista de nombre de objetos de esta clase se puede omitir y declarar los objetos más adelante como: nombre_de_clase OBJETO1,....,OBJETOn

```
//Ahora viene la definición de los métodos.
tipo nombre_de_la_clase::nombre_función (parámetros)
{
   Cuerpo de la función.
};
```

El operador :: llamado de ámbito sirve para asociar el método a una clase determinada. De esta forma podemos utilizar el mismo nombre de función para diferentes clases.

Una vez creado el objeto y definido sus métodos, para invocar una de sus funciones desde el programa principal se debe hacer:

```
{ ....
   nombre_del_objeto.método (parámetros);
  ....
}
```

7.8.4. Funciones constructoras y destructoras

Otro tema importante en las clases y objetos es el de las funciones constructoras y destructoras. Una función constructora es una función que se ejecuta tras la creación de un objeto, en su declaración.

```
cola objeto1;    // Declaración de objeto1 de la clase cola.
```

En esta función se llevan a cabo todos los mecanismos de inicialización que el objeto necesite. Por ejemplo, si estamos haciendo uso de la memoria de forma dinámica mediante el uso de punteros, pues como fase de inicialización podría ser la de reservar memoria. Por el contrario la función destructora es la encargada de llevar a cabo todos las acciones pertinentes antes de destruir un objeto. Siguiendo el ejemplo de los punteros, la función destructora del objeto podría ser la encargada de liberar la memoria antes reservada.

Para entender mejor el tema de las funciones constructoras y ver un ejemplo de la herencia veamos el siguiente ejemplo. Supongamos que hay dos clases base o padres B1 y B2. Ahora creamos una clase D1 derivada de B1 y B2 como:

```
class D1: public B1, public B2
```

Supongamos ahora que las funciones constructoras y destructoras de cada clase lo único que hacen es el escribir por pantalla "constructora de .." y "destructora de ..". Pues entonces la salida al siguiente programa sería:

```
void main() {
   D1 clase1; // Creamos un objeto llamado clase1.
   return 0;
};
```

SALIDA:

Constructor B1

Constructor B2

Constructor D1

Destructor D1

Destructor B2

Destructor B1

Como se puede observar el programa no hace nada, sólo declara un objeto de la clase D1.

Como se ha podido observar la sintaxis para declarar una nueva clase y que herede las propiedades de otra es la siguiente:

class nombre_clase **:** acceso1 clasebase1,...,acceso n clasebase n

El operador acceso indica el tipo de acceso dentro de la nueva clase de los datos y métodos heredados de la clase correspondiente.

A la hora de declarar las funciones constructoras y destructoras de una clase se hace igual que la declaración de los demás métodos con la salvedad que la función constructora tiene como nombre el nombre de la clase y la destructora tiene el nombre de la clase precedido de una tilde.

A menudo la función constructora tiene parámetros. Entonces la declaración de un objeto sería como sigue:

cola objeto1(parámetros);

Es en la declaración del tipo de objeto donde se ejecuta el constructor y al abandonar el bloque o cuando la función devuelve el control es cuando se ejecuta el destructor.

7.8.5. Un ejemplo de programación OOP usando la API

Antes de entrar en la programación con ObjectWindows veamos como es posible el realizar un programa orientado a objeto que utilice la API. Con este ejemplo se pretende aclarar que programación orientada a objeto en Windows no es sinónimo de ObjectWindows. Como se podrá apreciar es muy parecido en cuanto a contenido respecto al primer ejemplo que tratamos. Lo único que se ha hecho es declarar una serie de clases donde agrupamos los tipos de datos y funciones que usábamos en el primer ejemplo. Después, inicializando los datos miembro de cada clase y teniendo cuidado con el orden en que llamamos a las funciones creamos el siguiente programa:

```
#include <windows.h>
#include <stdlib.h>
#include <string.h>

// Definición del prototipo de la función de ventana WndProc.
extern "C" {
LONG FAR PASCAL WndProc (HWND hWnd, UINT messg, WPARAM wParam,
                                        LPARAM lParam);
            }
class Main  // Declaramos la clase principal.
        {
public:
   // Definimos los datos y funciones públicas de esta clase.
   static HANDLE hInst; // Número instancia actual
   static HANDLE hPrevInst; // Instancia anterior si hubo.
   static int nCmdShow;
   static int BucleMensajes (void); // Función Bucle de mensajes
};
```

```
// Inicializamos las variables de la clase a cero para evitar problemas.
HANDLE Main::hInst = 0;
HANDLE Main::hPrevInst = 0;
int Main::nCmdShow = 0;

// Declaramos el cuerpo de la función BucleMensajes
        int Main::BucleMensajes (void)
{
   MSG lpMsg; // Variable donde almacenaremos los mensajes.
   // Bucle de mensajes.
   while (GetMessage (&lpMsg,NULL,0,0)) {
   TranslateMessage (&lpMsg);
   DispatchMessage (&lpMsg);
   } // Fin del mientras.
   return lpMsg.wParam;
} // Fin de la función BucleMensajes.
class Window
{
   protected:

// Definimos este dato como protegido que significa que sólo es
// accesible por los miembros de esta misma clase y por los miembros
// de una clase derivada de ésta.
   HWND hWnd;
   public:
   // Función que devuelve el manejador de la ventana.
   HWND ObtenerManejador (void) {
```

```
    return hWnd;
        }
    // Función que muestra la ventana.
    BOOL Mostrar (int nCmdShow) {
    return ShowWindow (hWnd,nCmdShow);
        }
    // Función que actualiza la ventana.
    void Actualizar (void) {
    UpdateWindow (hWnd);
        }
};

class MainWindow : public Window
{ // Creamos una clase derivada de Window a la cual añadimos un par de
// funciones.
   private:
    static char szProgName[];
    public:
    // Esta función es la misma que la vista en el ejemplo anterior
    // cuando queríamos registrar la clase de ventana.
    static void RegistrarClaseVentana (void) {
    WNDCLASS wcApp;
       wcApp.lpszClassName=szProgName;
       wcApp.hInstance=Main::hInst;
       wcApp.lpfnWndProc=WndProc;
wcApp.hCursor=LoadCursor(NULL,IDC_ARROW);
```

```
    if (!RegisterClass (&wcApp))
    exit (FALSE);

     }

  MainWindow(void)  // Función constructora.
{ // Nuestra función constructora se encargará de crear la ventana.
                                    hWnd=CreateWindow (szProgName,
                                    "Título de la ventana",
                                    WS_OVERLAPPEDWINDOW,
                                    CW_USEDEFAULT,
                                    CW_USEDEFAULT,
                                    CW_USEDEFAULT,
                                    CW_USEDEFAULT,
                                    (HWND) NULL, (HWND) NULL,
                                                        Main::hInst,
                                    (LPSTR) this);
                                    if (!hWnd)
                                    exit (FALSE);
                                    Mostrar (Main::nCmdShow);
                                    Actualizar();
                                    } // Fin de RegistrarClaseVentana.

}; // Fin de la clase MainWindow.
char MainWindow::szProgName[]="ProgName";
LONG FAR PASCAL WndProc (HWND hWnd, UINT messg, WPARAM      wParam,
LPARAM lParam)
{ // Procedimiento de ventana que es igual al diseñado en el primer ejemplo.
   switch (messg)
       {
```

```
    case WM_DESTROY:
    PostQuitMessage(0);
        break;
        default:
return DefWindowProc (hWnd,messg,wParam,lParam);
    } // Fin de la sentencia switch.
} // Fin de la definición de WndProc.
int PASCAL WinMain (HANDLE hInst, HANDLE hPrevInst, LPSTR lpszCmdLine, int
nCmdShow)
{
// Definición de las variables miembro fuera de la declaración de la clase Main.
Aquí se les dan los valores retornados por Windows
    // donde se especifican las instancias actual y anterior.
        Main::hInst=hInst;
        Main::hPrevInst=hPrevInst;
        Main::nCmdShow=nCmdShow;

// Como en el primer ejemplo, si no exíste ninguna instancia previa
    // en ejecución pues se pasa a registrar la clase de la ventana.
    if (!Main::hPrevInst) {
    MainWindow::RegistrarClaseVentana();
        }

    // Declaramos un objeto MainWnd de la clase MainWindow
    // Tras su declaración se ejecuta su función constructora llamada
    // MainWindow().
    MainWindow MainWnd;
    // Por último ejecutamos el bucle de mensajes.
```

```
    return Main::BucleMensajes();
} // Fin de WinMain.
```

7.8.6. Un ejemplo con ObjectWindows

Bueno, es hora ya de entrar en nuestro primer ejemplo de programación en Windows usando ObjectWindows. Este es el ejemplo más difundido en toda la bibliografía consultada. El siguiente programa lo único que hace es representar una ventana en Windows, permitiendo maximizarla, moverla, cambiarle el tamaño y cerrarla. La gran ventaja de ObjectWindows es que todas estas funciones de maximizar la ventana ó moverla quedan ocultas al programador y es ObjectWindows el encargado de esa tarea. Analicemos por tanto el código tal y como hicimos en la aplicación API. Como primera línea tenemos un fichero de inclusión en el cual como ya se ha comentado se definen tipos de datos y prototipos de funciones. El fichero de inclusión en este caso se denomina OWL.h. En él va incluido también el Windows.h por lo que no va ha ser necesario el incluir dicho fichero.

```
#include <owl.h>
```

En la biblioteca de clases ObjectWindows existen dos bastantes importantes. Una de ellas es TApplication la cual se encarga de crear y gobernar una aplicación, y la otra se llama TWindow y es la encargada de crear y controlar las ventanas en Windows. Por tanto, para crear una aplicación Windows debemos hacer uso de TApplication y para poder crear una ventana deberemos usar la clase TWindow. Como se puede intuir será necesario del concurso de ambas clases en toda aplicación Windows.

Por consiguiente, como primer paso crearemos nuestra clase que heredará las propiedades de TApplication y que llamaremos MiApp.

```
class TMiApp : public Tapplication
// Los nombres de las clases por norma suelen empezar con la letra T.
{
public:
```

```
TMiApp (LPSTR Aname, HANDLE hInstance, HANDLE hPrevInstance, LPSTR
lpCmdLine, int  nCmdShow) : Tapplication (Aname, hInstance, hPrevInstance,
lpCmdLine, nCmdShow)
{
   // Aquí van los datos y funciones que queramos
   // introducir dentro de nuestra función constructora.
   // En este primer caso no definimos ninguno.
};
```

La primera función declarada es la constructora de nuestra clase. Como la clase es heredada, es necesario especificar la función constructora de la clase heredada, con sus parámetros, después de nuestra función constructora TMiApp() separada mediante dos puntos.

Los parámetros de esta función son los mismos que los de la función **WinMain** ya explicada y por tanto no entraremos en ellos otra vez. En esta clase definimos una función virtual. Pero ¿qué es una función virtual? El declarar una función como virtual permite al programador el redefinir dicha función en una clase derivada o heredada y dar una nueva versión de ésta, distinta a la declarada en la clase base. La función aquí declarada como virtual es una función miembro de Tapplication y siempre será necesario redefinirla en nuestros programas con ObjectWindows.

```
virtual void InitMainWindow();
}; // Fin de la declaración de la clase TMiApp.
```

Llegados a este punto definimos una clase TMiVentana que hereda las propiedades de la clase base TWindow y nos va a permitir crear y gobernar una ventana. Como se podrá ver en la declaración sólo llamamos al constructor de la clase. Con esto ya habremos dotado a la ventana con la posibilidad de maximizarse, cerrarse, cambiar de tamaño, y el programador ¡ni se entera! Eso es debido a que todo el trabajo lo realiza ObjectWindows. Más adelante y para dotar a la ventana con un comportamiento particular uno deberá definir funciones miembro de esta clase que realicen tareas específicas sobre la ventana en función

de los eventos ocurridos y que son comunicados a la aplicación en forma de mensajes. Estas funciones se suelen denominar de respuesta. Una función de respuesta no es más que una función que se invoca en respuesta a un mensaje recibido. Este tratamiento sustituye por completo a la sentencia *switch*() vista en el otro modelo de programación haciendo uso de la API.

```
class TMiVentana : Twindow
{
public:
TMiVentana (PTWindowsObject APArent, LPSTR Atitle)
:Twindow (Aparent, Atitle) {
// Aquí van los datos y funciones particulares del constructor
// de nuestra clase TMiVentana.
};
```

Una vez terminada la definición de las clases se pasa a la implementación de las funciones miembro de cada clase si es que existen. En nuestro caso tan sólo tenemos una función miembro declarada como virtual y se llama InitMainWindow().

```
void TMiApp :: InitMainWindow() {
MainWindow = new TMiVentana (NULL, "Programando en Windows");          //
NULL indica que se trata de la ventana principal
// y el siguiente parámetro es el título de la ventana.
};
```

Con esta función creamos un objeto del tipo ventana que es guardado en la variable *MainWindow* del objeto aplicación.

Y ahora como en todo programa Windows viene la función WinMain().

```
int PASCAL WinMain (HANDLE hInstance, HANDLE hPrevInstance, LPSTR
lpCmdLine, int nCmdShow)
```

```
{
 TMiApp MiAplica ("nombre apli.", hInstance, hPrevInstance, lpCmdLine, nCmdShow);
 MiAplica.Run();
 return MiAplica.Status;
};
```

La primera línea no es más que la creación de un objeto llamado MiAplica del tipo TMiApp. Como se puede observar es necesario especificar en la declaración los parámetros de la función constructora.

Seguidamente se llama a una función miembro de TApplication que ha sido heredada en MiAplica que se denomina **Run()**. Esta función es la encargada de ejecutar dentro de ObjectWindows nuestra aplicación. Por último, la función miembro **Status** retorna el estado de la aplicación.

Esto es a grandes rasgos el diseño de una aplicación en ObjectWindows. Para dotar a la ventana de un comportamiento particular debemos implementar una serie de funciones respuesta a mensajes. El diseño ahora, a partir de un programa Windows, no será más que ir añadiendo e implementando estas funciones respuesta según nos convenga, además de crear cajas de diálogo, cuadros de mensaje, etc...

Para terminar veamos un esquema general sobre un programa Windows básico.

Ficheros de inclusión

```
#include <owl.h>

Crear una clase derivada de TApplicationclass TMiApp : public TApplication

{  public:
TMiApp (LPSTR Aname, HINSTANCE hInstance, HINSTANCE
PrevInstance, LPSTR lpCmdLine, int nCmdShow) : TApplication
```

```
(AName,hInstance, hPrevInstance, lpCmdLine, nCmdShow) { };
virtual void InitMainWindow();
};
```

Crear una clase derivada de TWindow

```
_CLASSDEF (TMiVentana)
class TMiVentana : public Twindow
{
```

Miembros datos

```
public:
TMiVentana (PTWindowsObject Aparent, LPSTR Atitle);
```

Declaración de las funciones miembro};

Constructor de la clase derivada de TWindow

```
TMiVentana :: TMiVentana (PTWindowsObject Aparent, LPSTR Atitle)
    : Twindow (Aparent, Atitle) {
```

Incluir código en el constructor si es necesario

```
        };
```

Definición de las funciones miembro de TMiVentanaCrear la ventana principal

```
void TMiApp :: InitMainWindow () {
MainWindow = new TMiVentana (NULL, "título de la ventana");
};
```

Función principal del programa: **WinMain**

```
int PASCAL WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR
lpCmdLine, int nCmdShow)
{
```

```
TMiApp MiApp ("Nombre App.", hInstance , hPrevInstance, lpCmdLine,
nCmdShow);
MiApp.Run();

return MiApp.Status;
} // Fin de la aplicación.
```

Como se puede observar en una aplicación ObjectWindows no existe bucle de mensajes. Bueno, esto no es del todo cierto, en todo programa Windows existe un bucle de mensajes. Lo que ocurre es que ya esta implementado en ObjectWindow y por tanto el programador no debe preocuparse de el.

En este primer programa, en la función constructora de TMyWindow no se ha incluido ningún procedimiento, tan sólo se ha dejado vacía { }. Cuando deseemos crear controles, deberemos declararlos en este lugar, en la función constructora de TMyWindow.

Otro detalle que no se ha visto en este primer ejemplo es cómo se tratan los mensajes. Es de suponer, al no ver a primera vista el bucle de mensajes, que ObjectWindows se encarga de esta tarea. El programador tan sólo deberá crear funciones de respuesta a los mensajes que él desee, dando de esta forma un comportamiento particular a la ventana. La definición de estas funciones respuesta se hacen también en el constructor de TMyWindow y son algo parecido a lo siguiente:

virtual void Respuesta1(RTMessage)=[ID_FIRST+ID_STATUS];

El mensaje ID_STATUS puede ser enviado a la ventana tras la pulsación de un botón por ejemplo. La creación de botones, y demás controles, así como la creación de menús y cuadros de diálogo no es tarea muy complicada y tan sólo me remito a cualquier libro de programación en Windows con ObjectWindows.

Capítulo 8

Especificación Windows Sockets

8.1. Introducción

Ya hemos visto gran parte de los detalles de la programación en red y bajo Windows. Ahora tan sólo resta el analizar de forma detenida la especificación W.S.

En esta especificación se comentan de forma muy sencilla los conceptos del modelo Cliente/Servidor, datos fuera de banda, broadcasting, sockets bloqueantes y no-bloqueantes, tratamiento de errores, etc.. Estos conceptos son ampliados en el presente capítulo de forma que sean comprensibles para aquellas personas que se introduzcan en el tema. Así, en este capítulo no se pretende hacer ni mucho menos una traducción literal de la especificación, ni inclusive un listado alfabético de todas las funciones y rutinas descritas en la especificación. Tan sólo se realiza una enumeración de todas las funciones que contempla la especificación dando una breve descripción de ellas con el fin de tener una visión global de lo que permiten y de las posibilidades funcionales de esta especificación.

Para analizar los detalles de cada función, como son los parámetros, posibles valores de retorno, errores que pueden darse, etc..., se recomienda acudir a la especificación ya que se ha diseñado con ese fin, como manual de consulta. En este capítulo sólo se pretende ampliar conceptos que no quedan claros en dicha especificación, así como realizar un breve comentario sobre todas las rutinas de forma general.

También y antes de continuar con este capítulo debemos dejar claro que parte de la programación con sockets ya se ha visto en temas anteriores, como crear una dirección, conectarse a un host, intercambiar datos. Así, todos los algoritmos discutidos anteriormente son válidos ahora y no serán repetidos en este capítulo.

8.2. ¿Cómo usar W.S.?

Desde el punto de vista del programador, para poder hacer uso de las funciones de Windows Sockets es necesario disponer del fichero Winsock.DLL. Es una libreria de enlace dinámico de Windows en la que se recoge la implementación de todas las funciones descritas en la especificación. Junto con esta librería de enlace dinámico se suministra un fichero de cabecera imprescindible para la programación. Este fichero de cabecera se llama "Windsock.h" si trabajamos en un entorno de programación C ó C++, en el que se encuentran todas las definiciones de los tipos de datos contemplados en la especificación así como las declaraciones de las funciones de W.S.

Además, para que un programa W.S. funcione es necesario que exista algún tipo de pila de protocolos cargada previamente, y que además la detecte correctamente Windsock.DLL.

Un ordenador antes de poder realizar una conexión con el exterior a través de W.S. debe tener un software cargado previamente encargado de interactuar con la red. Este software se suele estructurar en un modelo de capas. Las capas superiores hacen uso de las inferiores proporcionando una independencia de una capa a otra.

Hasta aquí todo parece estar bastante claro pero es necesario advertir un detalle; la librería Winsock.DLL es implementada por cada vendedor de software de red. Por ejemplo, la librería Winsock.DLL para TCP/IP de un vendedor como puede ser 3Com Corp. es diferente que la que pueda proporcionar Frotier Tech. o IBM. La especificación no dice nada al respecto de cómo deben ser implementadas sus funciones por los vendedores, tan sólo dicta cómo deben funcionar.

Pues bien, un esquema típico en la configuración del ordenador para poder funcionar con W.S. suele ser la siguiente. Primero se conecta una tarjeta de red al ordenador. Después se carga el software de la API del Driver Hardware que bien puede ser *Packet Driver, NDIS, ODI,..* Este software será el encargado de interactuar directamente con la tarjeta de red, haciendo que el acceso sea independiente de la tarjeta de red instalada. Es decir, el *Packet Driver* aisla las particularidades de cada tarjeta de red y proporciona una serie de

funciones comunes a las subsiguientes capas de software que se monten después. Así como ya se dijo, el acceso al *Packet Driver* será igual e independientemente de la tarjeta de red utilizada. Después se monta el software correspondiente al conjunto de protocolos que se desee utilizar. En nuestro caso son el conjunto llamado TCP/IP. Finalmente, si todo ha ido bien, ya es posible utilizar los W.S.

Destacar que Winsock.DLL se encuentra en la capa más alta y que hace uso de los servicios directamente del software TCP/IP que se ha cargado previamente.

La forma y cómo se instala este software, tanto el *Packet Driver* como la pila de protocolos, viene detallada junto con el paquete de software adquirido. Aquí supondremos que partimos ya de una configuración que nos permita salir al exterior haciendo uso de TCP/IP.

8.3. ¿Qué es un socket bloqueante y no-bloqueante?

Una vez que estemos en disposición de hacer uso de los W.S. sin problemas nos resta enfrentarnos a las dificultades que trae consigo la programación con sockets.

Ya hemos visto los posibles algoritmos en el diseño de aplicaciones Cliente/Servidor, pero resta analizar los detalles de algunas funciones utilizadas anteriormente y que no han sido comentadas. Además es necesario introducir nuevos conceptos que no se han visto en los capítulos anteriores y que son específicos de Windows.

La especificación en sí no reporta mucha información en lo que a la programación concierne. Como se ha comentado anteriormente, esta información se ha obtenido de otras fuentes basadas en el modelo de programación con sockets bajo el entorno UNIX.

Uno de los conceptos que nos deben quedar más claros es el de socket bloqueante y socket no bloqueante.

Cuando nosotros llamamos a una función, cualquiera que sea, y ésta invierte un periodo de tiempo arbitrario o muy largo en retornar un valor, nuestro programa se dice que queda bloqueado en ese punto. Este detalle a veces carece de importancia pero aquí cobra una notable transcendencia. Supongamos el siguiente código en el lenguaje MatLab:

```
t=0:0.1:100;
y=sin(100*pi*t);
Y=FFT(y);
plot (abs(Y));
```

La función FFT() tarda normalmente bastante tiempo en retornar un resultado si el número de muestras de la señal "y" es grande. Debido a éste, nuestro programa que se ejecuta de una forma secuencial, parece bloquearse durante ese tiempo en la instrucción Y=FFT(y). Como se trata de un simple cálculo del cual se quiere ver su representación gráfica, al usuario no le importa este bloqueo, o por lo menos no influye en el desarrollo normal del programa.

Sin embargo, supongamos ahora una situación en un programa enfocado hacia una aplicación de red. Supongamos que poseemos dos puntos finales de comunicación, es decir, tenemos dos sockets abiertos, s1 y s2, para recibir datos por ellos. En nuestro programa, primero leemos de s1 y después leemos los datos del otro.

```
recv(s1,buffer1,sizeof(buffer1),0);
recv(s2,buffer2,sizeof(buffer2),0);
```

Además supongamos que en el primer socket no hay datos que leer, es decir, no se han recibido datos por s1, y que por el segundo socket ya hay datos esperando para ser leídos. ¿Qué ocurriría? Bueno, básicamente, nuestro programa quedaría bloqueado en el primer *recv()* sobre s1 esperando que le llegasen datos por s1 para leer, mientras que los datos que llegaron por el segundo socket no serán leídos hasta que lleguen datos por el primero. Es decir, un bloqueo aquí esta provocando que no se estén leyendo datos procedentes de otro canal. Debido a esta situación y otras muchas que se pueden presentar, el tema de los bloqueos cobra suma importancia en el diseño de tales programas.

El bloqueo trae consigo otro problema. Si ahora nos trasladamos a un entorno multitarea cooperativa como es Windows, un bloqueo en una aplicación arrastraría a todo el sistema Windows a bloquearse. Sin embargo, en un entorno multiproceso como es UNIX esto no sería un gran problema ya que el bloqueo de un proceso no afectaría a otros procesos que se estuviesen ejecutando en ese momento de forma concurrente. Para evitar que un bloqueo de este tipo llevase al sistema Windows a un bloqueo general, la especificación W.S. recoge la implementación de un bucle de mensajes propio, en el cual se entra cuando una función de la especificación W.S. produce un bloqueo. Es necesario recalcar que la especificación dice que se entra en dicho bucle de mensajes sólo cuando los bloqueos son producidos por funciones de la especificación W.S.. Si por cualquier motivo una función que no es de W.S. provocase bloqueo, nunca se entraría en este bucle de mensajes.

Así, de esta forma, se seguirán despachando los mensajes aún cuando una operación bloqueante de W.S. este en curso. Gracias a esto, no se arrastraría al sistema Windows a un bloqueo general. Cabe entonces preguntarse ahora ¿qué tipo de bloqueo puede producirse si se siguen despachando mensajes y el curso de la aplicación puede continuar? Pues bien, la especificación W.S. lo que prohibe, y por tanto no se permite, es que la aplicación llame a una función de W.S. cuando existe ya una bloqueante en curso. Esta premisa se añadió a la especificación dado que era bastante difícil controlar de una manera segura una serie de situaciones.

Para ver más claro el porque la especificación no permite hacer una llamada a una función de W.S. cuando existe una bloqueante en curso, veamos la siguiente situación. Supongamos que llegamos a un punto en el código en el que se realiza una llamada *recv(s1,...)*. Esta llamada además produce bloqueo porque no existen datos para leer. Ahora el bucle de mensajes de Windows Sockets despacha un mensaje de la cola que deriva a una parte del código de nuestro programa en la cual se encuentra un sentencia *close(s1)* y mientras tanto llegan datos por s1. Esta función cierra el socket. Aquí se produce ya una situación no deseada. Por un lado se ha llamado a *recv()* para leer los datos del socket y antes de que fuesen leídos se ha cerrado el socket. Debido a estas situaciones y otras más

que se pueden dar, no se permite llamar a cualquier función de W.S. cuando exista una bloqueante en curso. Tan sólo son contempladas en la especificación dos funciones que pueden ser invocadas en estas situaciones y que ayudan al programador a tratar estas situaciones.

Una de ellas es *WSAIsBlocking()* la cual informa si existe alguna operación bloqueante en curso, y la otra es *WSACancelBlockingCall()* que se encarga de cancelar la última llamada a W.S. que está produciendo bloqueo.

Por tanto, y como norma general, cuando escribamos un programa, en la parte del código referente a la finalización de la aplicación, deberemos comprobar si existen llamadas bloqueantes en curso y en tal caso, antes de salir de la aplicación deberemos llamar a *WSACancelBlockingCall()* para cancelarlas. También es muy importante que antes de salir de una aplicación todos los sockets utilizados sean cerrados..

8.4. ¿Qué es el BlockingHook?

Analicemos ahora un poco en detalle el bucle de mensajes que implementa la especificación W.S. Dentro de una posible implementación de la especificación W.S. una operación bloqueante que no puede ser completada de forma inmediata se trata como sigue: primero, tras la llamada a la función que producirá bloqueo la DLL inicia la operación correspondiente a esta función invocada. Después entra en un bucle en el cual despacha cualquier mensaje Windows (permitiendo al procesador inclusive derivar a otra tarea si es necesario) para finalmente chequear si la operación iniciada por la DLL ha finalizado. En caso de que la operación haya finalizado o que se haya invocado a la función *WSACancelBlockingCall()*, la función bloqueante se termina reportando un resultado.

Por tanto, en este bucle se analiza si existe algún mensaje en la cola de la aplicación. En tal caso se despacha. Después, y si no existen más mensajes que despachar, se mira si se ha invocado a la función *WSACancelBlockingCall()* o si la operación bloqueante ha terminado, en cuyo caso se abandona el bucle de mensajes implementado por W.S. El código sería algo parecido a lo siguiente:

```
for (;;) { // Bucle infinito.
        // Control de mensajes
        while (DefaultBlockingHook));

        // Mira si se ha cancelado la operación bloqueante
        if (opperation_cancelled())
           break;
        // Mira si la operación bloqueante ha finalizado
        if (opperation_completed())
             break;
      }


BOOL DefaultBlockingHook(void) {

MSG msg; // Variable donde almacenamos el mensaje a despachar
BOOL ret; // Valor que retornará la función.

// Extraemos el siguiente mensaje de la cola.
// ret=TRUE si existe mensaje
// ret=FALSE si no habia mensaje en la cola.

ret = (BOOL) PeekMessage(&msg,NULL,0,0,PM_REMOVE);

// Si existía mensaje y se extrajo correctamente, lo
// depachamos.
if (ret) {
             TrasnlateMessage(&msg);
             DispatchMessage(&msg);
      }
```

```
// Retornamos el valor ret
return(ret);}
```

Como se puede observar por el código, el bucle de mensajes implementado por W.S. es infinito (for (;;){ ..}) Dentro de este bucle primero se despachan todos los mensajes en la cola de la aplicación si es que existe alguno. Una vez despachados todos los mensajes presentes se pasa a ver si se ha cancelado la operación bloqueante o si ésta ha finalizado. En ambos casos, tanto si se ha cancelado como si se ha finalizado la operación bloqueante se abandona el bucle infinito.

Este bucle de mensajes es bastante sencillo y en la mayoría de los casos no es necesario modificarlo. Pero existen casos en los que el programador desee modificar este bucle de mensajes y diseñar uno propio. Para tales situaciones la especificación W.S. introduce la función *WSASetBlockingHook()*. Gracias a esta función el programador puede implementar una función de control de mensajes propia y hacer que ésta sea usada por W.S. Para ello debe pasarle a la función *WSASetBlockingHook()* un puntero a la función diseñada. Entonces W.S. dejará de usar la función Blocking-Hook por defecto (*DefaultBlockingHook()*) para empezar a usar la que le hemos especificado mediante el puntero. Si en cualquier momento se desea restaurar la función por defecto de procesado de mensajes de W.S. tan sólo hay que llamar a la función *WSAUnhookBlockingHook()*.

Por último para finalizar con el apartado de bloqueante y no bloqueante añadir que el adjetivo de bloqueante en la especificación se aplica a sockets y no a funciones. Es decir, se suele hablar de sockets bloqueantes y sockets no bloqueantes, aunque realmente el bloqueo es producido por determinadas funciones de W.S. En la especificación se enumeran todas las funciones que pueden causar bloqueo mediante un asterisco. Se debe entender que en un socket bloqueante todas estas funciones bloqueantes pueden producir bloqueo, como es lógico.

Un socket por defecto actúa en modo bloqueante y sólo actuará como no bloqueante si se conmuta su estado mediante la llamada a la función *WSAAsyncSelect()* o mediante *ioctlsocket()* utilizando como comando FIONBIO.

A la hora de diseñar aplicaciones se deberá por tanto elegir uno de los dos modelos, bloqueante ó no-bloqueante. Cuando nos decidimos por la versión bloqueante se suele decir que la aplicación diseñada usa llamadas síncronas mientras que en una versión no-bloqueante se suele denominar a la aplicación diseñada como asíncrona. La diferencia de un modelo a otro es bastante sustancial. Además la especificación W.S. recomienda encarecidamente el diseño de aplicaciones asíncronas dado que su forma de trabajar se ajusta muy bien al modelo de programación en Windows.

8.5. Funciones de la especificación

La especificación W.S. se puede observar como un conjunto de funciones divididas en dos grandes grupos. El primero es el conjunto de funciones que son heredadas del entorno UNIX, origen éste de la interfase SOCKET. El segundo es un nuevo conjunto de funciones específicas diseñadas con el fin de aprovechar el modelo de programación basado en mensajes que soporta Windows.

El conjunto de funciones heredadas del entorno UNIX es el siguiente:

- *accept()* *
- *bind()* *
- *closesocket()* *
- *connect()* *
- *getpeername()* *
- *getsockname()* *
- *getsockopt()*
- *htonl()*
- *htons()*
- *inet_addr()*
- *inet_ntoa()*
- *ioctlsocket()*
- *listen()*
- *ntohl()*
- *ntohs()*
- *recv()* *
- *recvfrom()* *
- *select()* *
- *send()* *
- *sendto()* *
- *setsockopt()*
- *shutdown()*
- *socket()*

Además existen otro grupo de funciones heredades del entorno UNIX y que son las denominadas de base de datos.

gethostbyaddr() * *gethostname()* *getprotobynumber()*

gethostbyname() * *getprotobyname()* * *getservbyname()* *

getservbyport() *

Y por último siguen las funciones específicas para Windows incluidas en esta especificación. Como se puede observar el parecido con algunas de las ya citadas anteriormente es muy grande y la única diferencia es en su filosofía de trabajo.

WSAAsyncGetHostByAddr()	*WSAAsyncSelect()*	WSASetBlockingHook()
WSAAsyncGetHostByName()	WSACancelAsyncRequest()	WSASetLastError()
WSAAsyncGetProtoByName()	WSACancelBlockingCall()	WSAStartup()
WSAAsyncGetProtoByNumber()	WSACleanup()	WSAUnhookBlockingHook()
WSAAsyncGetServByName()	*WSAGetLastError()*	
WSAAsyncGetServByPort()	*WSAIsBlocking()*	

Resumiendo, la especificación consta de una serie de funciones que se pueden agrupar como sigue:

- Funciones heredades de UNIX:
  - Funciones Base de datos
  - Otras Funciones
- Funciones específicas de Windows.

8.6. Funciones WSA...()

La especificación para poder controlar los bloqueos y aprovechar la programación conducida por mensajes en la que se basa Windows ha introducido esta serie de funciones específicas. Una de las más importantes es la función *WSAAsyncSelect()*.

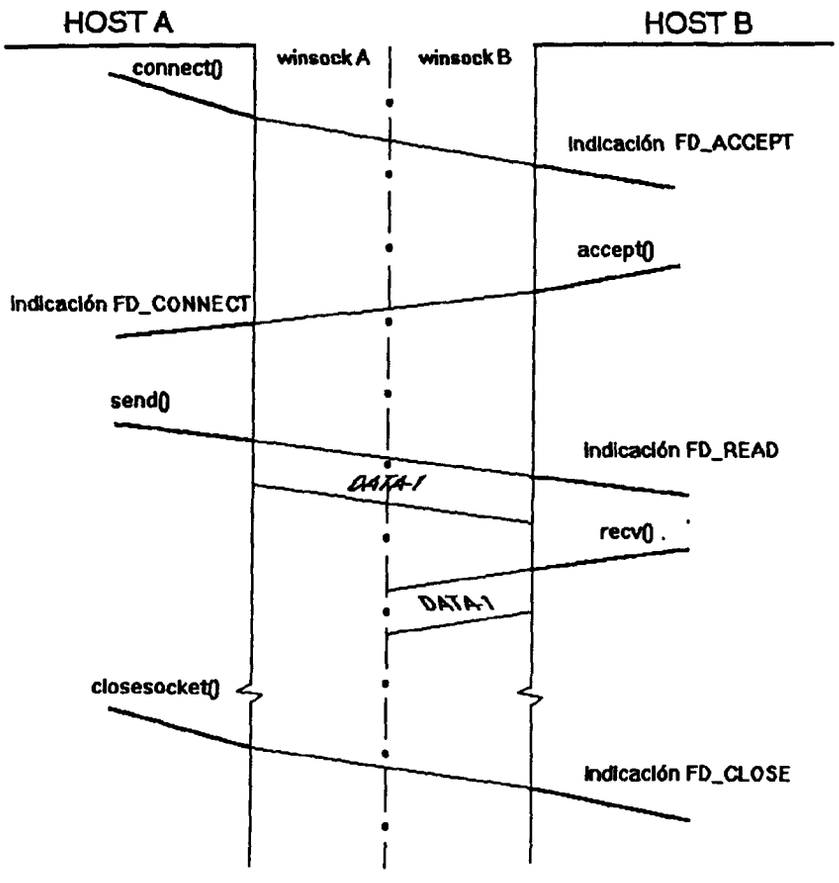
Ya hemos visto que intentar leer datos de un socket sobre el cual todavía no se ha recibido información produce un bloqueo. Para el programador sería necesaria la existencia de una función que reportara o informase sobre la disponibilidad de un socket para leer datos sobre él, asegurando por ejemplo, que una llamada a la función *recv()* no producirá bloqueo.

Lo mismo ocurre cuando se intentan enviar datos por un socket. Esta operación igualmente puede provocar bloqueo y por tanto se hace necesaria una función que informase también sobre la disponibilidad de las subcapas para realizar un envío de datos.

En definitivas cuentas, todas aquellas funciones que necesiten interactuar con la red para determinar un resultado pueden producir un bloqueo debido a la falta de disponibilidad de la red. Es decir, cuando queremos enviar datos por un canal a través de un socket, los datos pueden ser enviados inmediatamente o no, esto depende del estado de la red y de los buffers locales de los que dispone Winsock. Por tanto, gracias a la función *WSAAsyncSelect()* se puede pedir a la capa de transporte que nos informe cuando es posible un determinado conjunto de sucesos en la red. Dentro de estos sucesos cabe destacar los dos ya comentados de escritura y lectura, pero existen otros más que se describen en la especificación y que son también importantes.

Además de comprobar la disponibilidad de Escritura y Lectura como ya se ha comentado, esta función puede informar sobre la llegada de datos fuera de banda, sobre el establecimiento de una conexión tras una llamada *connect()*, si se ha cerrado un socket, y en los modelos orientados a conexión también informa si es posible realizar un *accept()* sin peligro de bloqueo; nos informa que ha llegado una petición por el socket de escucha y por tanto se puede realizar un *accept()* para aceptarla sin que se produzca bloqueo. Es necesario advertir desde ahora que cuando el programador haga uso de esta función, *WSAAsyncSelect()*, automáticamente el socket pasará de modo bloqueante a no-bloqueante. Es decir, la llamada a esta función trae consigo la conmutación del socket en no-bloqueante como efecto lateral.

Para ver un poco más claro este apartado, he recurrido a un esquema donde representamos dos posibles hosts que entran en comunicación.



En el siguiente esquema se representa gráficamente el secuenciamiento de una serie de primitivas con el objetivo de aclarar el tema de los eventos en la función *WSAAsyncSelect()*.

Como vemos en el esquema existen dos Hosts entre los que se va ha establecer una conexión. Todas las indicaciones que se muestran se traducen en mensajes que se envían a la aplicación para comunicarle el suceso en sí. Por ejemplo, comencemos desde el Host B a analizar todas las indicaciones que aparecen. La primera corresponde a la indicación de un suceso de red FD_ACCEPT. Este suceso ocurre como se puede observar cuando llega una petición de conexión por parte de otro Host. El nombre que la especificación le ha dado a este suceso, FD_ACCEPT, viene de la idea de que tras recibir nuestra aplicación una notificación de un suceso FD_ACCEPT, se puede ejecutar un *accept()* sin ningún tipo de problema de bloqueos. Pero para nosotros no significará más que una indicación de que existe una petición de conexión pendiente. Ahora el Host B puede aceptarla o no. En nuestro esquema la aceptamos. Tras este *accept()* se establece ya por fin la conexión. La

siguiente indicación que observamos en el lado del Host B se trata de una indicación correspondiente a un envío de datos por parte del Host A. Este suceso se denomina dentro de W.S. como FD_READ. Viene a significar que tras recibir una notificación de un FD_READ nuestra aplicación puede ejecutar un *recv*() sin ningún tipo de problemas de bloqueo y recibir los datos. Ejecutamos por tanto un *recv*() para leer los datos y continuamos. La última indicación recibida es la correspondiente a una desconexión. Este suceso viene reflejado en W.S. como FD_CLOSE y nos indica que se ha ejecutado un *closesocket*() o *shutdown*() en cualquiera de los dos Hosts. En el ejemplo suponemos que el cierre de la conexión lo lleva a cabo el Host A, pero bien podría haberlo realizado el Host B y no por ello dejaría de recibir la notificación.

Vayamos ahora al lado del Host A para analizar un tipo de evento que no hemos visto en el lado del Host B. Este evento es la confirmación del establecimiento de la conexión. Gracias al evento FD_CONNECT es posible pedir la confirmación del establecimiento de la conexión.

Aquí hemos visto cuándo se producen ciertos eventos de red, pero nuestra aplicación no tiene por qué saber de todos ellos. Es decir, en nuestra aplicación por ejemplo tan sólo nos interesaría tener notificación de los eventos FD_READ y FD_CONNECT y no de los eventos FD_CLOSE y FD_ACCEPT. Para ello, la especificación permite que en la función WSAAsyncSelect() se especifiquen el conjunto de eventos de red que se desean tener en cuenta. Esta claro que si no hemos seleccionado el evento FD_XXXX nunca tendremos noticias de él aunque suceda. El conjunto de eventos seleccionados depende ya del programador y del diseño propio de la aplicación.

Como ya hemos visto, *WSAAsyncSelect*() es una función que activa un mecanismo de mensajes para notificar ciertos sucesos de red. La implementación de esta función en cuanto a la notificación de sucesos se dice que es "***level triggered***". Esto no es más que mientras un suceso de red siga latente, se seguirán despachando mensajes para notificarlo. Por ejemplo, el Host A envía 1000 bytes de datos. Por su parte, en el Host B *WSAAsyncSelect*() despacha un mensaje a la aplicación notificando que hay datos para leer. La aplicación del Host B lee tan sólo 500 bytes, quedando otros 500 bytes sin leer. Siguen

existiendo por tanto datos que leer y *WSAAsynSelect*() despacha otro nuevo mensaje a la aplicación indicando que hay datos para leer. Aquí se puede presentar un problema de comprensión en el funcionamiento de *WSAAsyncSelect*(). Puede dar la impresión que esta función mientras haya datos que leer esta enviando de forma continuada un mensaje de notificación a la aplicación. Esto realmente no es así. Lo que ocurre es lo siguiente: desde que llegan los datos para leer por primera vez, *WSAAsyncSelect*() despacha un mensaje de notificación. *WSASyncSelect*() no volverá a despachar más mensajes notificando que hay datos para leer, hasta que se haya producido una lectura completa o parcial de esos datos. Si en la lectura de esos datos no se han podido leer todos, *WSAAsyncSelect*() despachará otra vez el mensaje, pero sólo si se ha llamado a *recv*() o *recvfrom*() para leer los datos y no se han podido leer todos los datos que han llegado. Esta manera de funcionar es bastante razonable y lógica pero es conveniente aclararla.

Lo mismo ocurre por ejemplo con el suceso FD_ACCEPT. A un Host pueden llegarle varias peticiones de conexión. WSAAsyncSelect() indicará este suceso mediante un mensaje a la aplicación. *WSAAsyncSelect*() no notificará más mensajes de este tipo hasta que no se haya ejecutado un *accept*() para aceptar la conexión entrante.

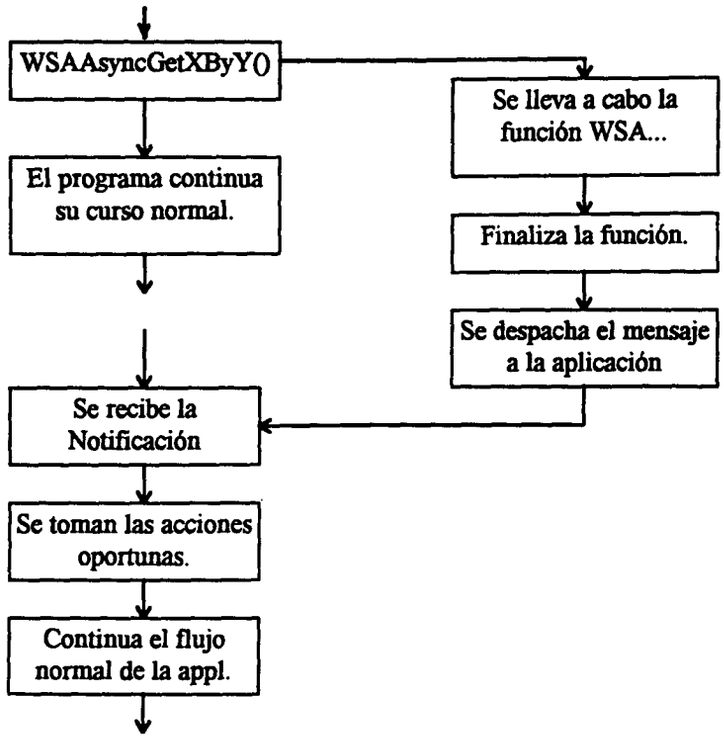
También existen otras funciones como son las *WSAAsyncGetXByY()* que son denominadas funciones de base de datos. Gracias a estas funciones se puede obtener información variada sobre las direcciones del host, el número de puerto de servicio correspondientes a un servicio determinado, etc... Como se puede observar son iguales que las funciones heredadas de UNIX pero con la salvedad que éstas aprovechan los mecanismos de Windows basados en los mensajes. El uso de estas funciones no obliga necesariamente a conmutar el estado del socket bloqueante a no-bloqueante. La única función que altera el estado del socket es *WSAAsyncSelect()*. Son unas funciones que actúan de forma asíncrona pero no afectan el estado del socket.

Así, cuando se hace una llamada a este tipo de funciones, la función devuelve un manejador de tarea asíncrona y comienza a realizar su tarea. Mientras esta función lleva a cabo su tarea, el programa continua su curso y por tanto no se produce ningún bloqueo. De cara al programador es como si se ejecutase paralelamente la función *WSAAsyncGetXByY()* y nuestro programa. Si en cualquier momento se quiere suspender o detener la ejecución de

esta tarea paralela, es necesario indicar mediante su manejador de tarea asíncrona qué tarea es la que se desea abortar. Para entenderlo mejor hagamos un paralelismo con el S.O. UNIX y los procesos "*background*". Estas funciones se comportan como un proceso UNIX ejecutado en el "*background*" (de fondo). Cuando se lanza el proceso en el *background*, el S.O. UNIX devuelve un **pid** (*process identifier*) que identifica al proceso. Si por cualquier motivo queremos abortar o finalizar este proceso deberemos llamar a la función *kill()* y decirle mediante el pid correspondiente, qué proceso del *background* queremos destruir. El manejador de tarea asíncrona se puede comparar con el pid devuelto por la función *fork()* de UNIX al crear un proceso paralelo.

Pero ¿qué ocurre cuando finaliza la función asíncrona? Como el tiempo que puede invertir la tarea asíncrona en llevarse a cabo es arbitrario, el lugar donde se encuentre nuestra aplicación es uno desconocido. ¿Cómo puede entonces nuestra aplicación darse cuenta que la tarea asíncrona finalizó y tomar las acciones oportunas? La solución está en los mensajes de Windows.

Cuando nosotros lanzamos una tarea asíncrona utilizando una función del tipo *WSAAsyncGetXByY()* debemos indicar un mensaje y una ventana. Entonces cuando finalice la tarea, se enviará el mensaje especificado a la ventana indicada. De esta forma la aplicación tomará cuenta de que la tarea asíncrona ha finalizado y por tanto puede leer ya los resultados almacenados en el buffer correspondiente. El funcionamiento se puede ver de forma gráfica en la siguiente ilustración:



Veamos ahora el funcionamiento de *WSAAsyncSelect()* que es bastante simple. En Windows hemos visto que quizás el objeto más importante es la ventana. Una aplicación se compone de una o varias ventanas por la que va discurriendo la aplicación. También hemos visto el carácter de Windows sobre los mensajes. Por tanto, a la hora de que una función comunique o informe un suceso cualquiera a nuestra aplicación, es necesario que se haga mediante un mensaje. Además se debe especificar a dónde será enviado este mensaje, es decir, será necesario especificar la ventana de la aplicación a la que mandaremos este mensaje. Como vemos la filosofía de todo este conjunto de funciones es la misma; cuando finalizan lo comunican a la aplicación mediante un mensaje enviado a una ventana determinada.

La función que ahora nos ocupa debe saber entonces a qué ventana enviar el mensaje, qué mensaje enviar y qué conjunto de sucesos de red quiere que se comprueben sobre el socket determinado.

Así la función como primer parámetro acepta el descriptor del socket sobre el que se van ha analizar todos estos eventos. En segundo lugar se especifica el manejador de una ventana Windows. La función comunicará el evento a esta ventana enviándole un mensaje.

El mensaje que será enviado es especificado en el tercer parámetro. Y por último, se especifica el evento o el conjunto de eventos del que se quieren tener noticias.

Analicemos ahora una serie de situaciones a modo de ejemplo para aclarar un poco mejor esta función.

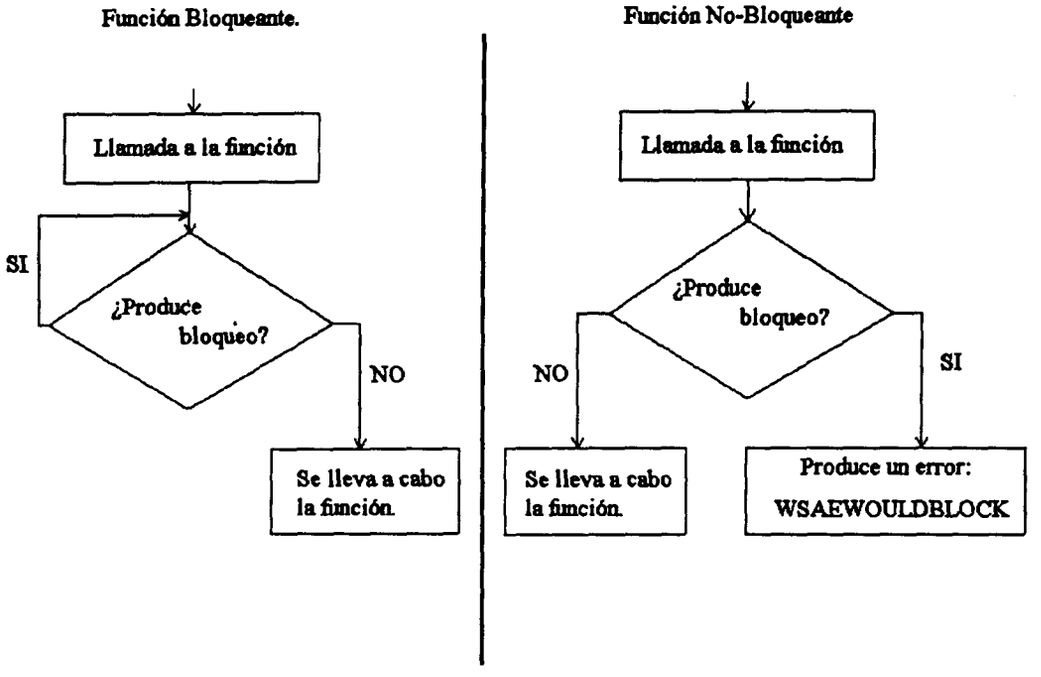
Supongamos que hemos creado un socket, lo hemos conectado a un servidor y estamos esperando su respuesta. El programa para poder leer la respuesta debe llamar a *recv()*. Si la respuesta tarda un tiempo el programa quedará bloqueado en este punto. Por tanto, sería interesante llamar a la función *recv()* cuando ya hubiesen llegado los datos. Para ello el programador puede optar por la opción de no ejecutar un *recv()* para leer la respuesta del servidor y ejecutar un *WSAAsyncSelect(s,hWnd,msg,FD_READ)*. De esta forma, el programa continuaría su curso natural y cuando llegasen los datos, el mensaje msg sería enviado a la ventana hWnd para comunicar el evento FD_READ sobre el socket s. Es entonces en ese momento cuando ejecutamos un *recv(s,...)* para leer la respuesta del servidor, teniendo ya la seguridad de que la lectura será inmediata y no existirá ningún bloqueo.

Antes de seguir con el siguiente ejemplo hagamos un resumen de lo visto hasta ahora sobre estas funciones específicas de Windows ya que es un tema bastante importante.

Cuando un socket es creado, por defecto es bloqueante. Si ejecutamos cualquier función de las señaladas con un * es muy posible que nuestro programa quede bloqueado en ese punto hasta que se realice la tarea de la función llamada. Sin embargo, si hacemos uso de la función *WSAAsyncSelect()* el socket pasará a comportarse como no bloqueante y estas funciones ya no producirán bloqueo a la aplicación.

Es decir, si llamamos a una función de W.S. en esta situación pueden ocurrir dos cosas. La primera es que esta función no se pueda llevar a cabo inmediatamente, o sea, produce bloqueo. En este caso W.S. reportará un error indicando que dicha función produce bloqueo, esta función no será ejecutada y continuará el flujo normal de la aplicación. O puede ocurrir que no produzca bloqueo y por tanto se lleve a cabo la función

inmediatamente sin problemas. Para ver de forma esquemática la diferencia de evolucionar una función bloqueante y otro no-bloquenate se ha incluido el siguiente esquema:



Siguiendo con los ejemplos, veamos qué ocurre cuando se desea que se informe de varios sucesos de red sobre un mismo socket. Se agrupan como sigue:

```
WSAAsyncSelect(s,hWnd,msg, FD_READ | FD_WRITE |...);
```

Un detalle muy importante es que no se permite que se envíen diferentes mensajes para comunicar diferentes eventos sobre un mismo socket. Es decir, la siguiente secuencia no sería correcta:

```
WSAAsyncSelect (s, hWnd, msg1, FD_READ);
WSAAsyncSelect (s, hWnd, msg2, FD_WRITE);
```

El intentar enviar un mensaje msg1 para notificar la disponibilidad de lectura sobre s y enviar otro mensaje diferente msg2 para reportar la disponibilidad de escritura sobre s no se permite. Si hacemos varias llamadas a *WSAAsyncSelect()* sobre un mismo socket tan sólo tendrá efecto la última llamada y las demás se anularán. Es decir, en el ejemplo mostrado, la

última llamada a *WSAAsyncSelect()* anularía a la primera, y tan sólo se reportaría la disponibilidad de escritura sobre s mediante el mensaje msg2.

Hemos visto que para notificar a la aplicación ciertos eventos ocurridos sobre un socket se hace mediante un mecanismo de mensajes. Estos mensajes son despachados a la cola de la aplicación como cualquier otro mensaje Windows. Debido a esto, es posible recibir mensajes almacenados en la cola, notificando ciertos eventos sobre un socket, después de haber anulado la función *WSAAsyncSelect()*. Es decir, especificando en el parámetro *lEvent* de la función un valor 0 se consigue que se cancele la notificación de eventos sobre el socket determinado. Pero esto no impide que la aplicación reciba mensajes anteriores que estaban almacenados en la cola; por tanto es necesario que nuestra aplicación tenga esto muy en cuenta.

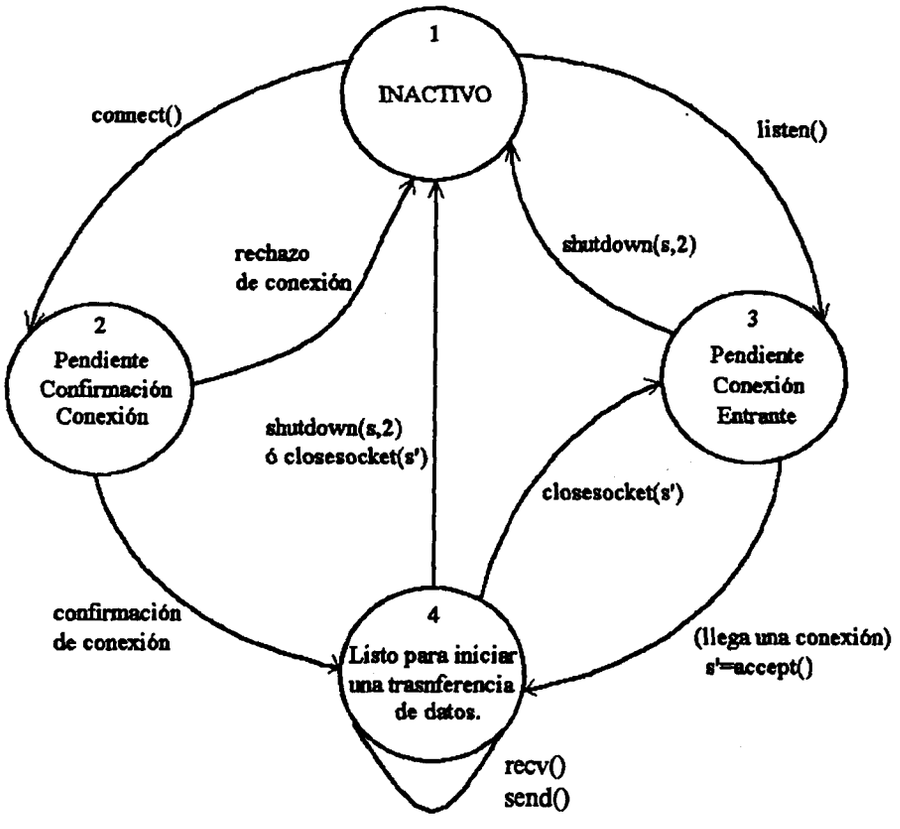
Otro detalle importante que se comenta en la especificación W.S. es el siguiente. Cuando se tiene un socket en modo de escucha y a través de éste se acepta una conexión, la función *accept()* retorna un nuevo socket que tiene las mismas propiedades que el utilizado para la escucha.

Pues bien, si sobre el socket de escucha se ha ejecutado un *WSAAsyncSelect()* con una serie de eventos, el socket reportado por *accept()* tendrá activada la notificación de los mismos eventos y mediante el mismo mensaje debido a que ha heredado las propiedades del socket de escucha. Por tanto, si se desease cambiar los eventos sobre el socket retornado por *accept()* sería necesario ejecutar un nuevo *WSAAsyncSelect()* indicando los nuevos eventos.

Hasta aquí todo parece claro. Pero es ahora donde hay que darse cuenta de un detalle bastante significativo. Entre la función *accept()* y la posible llamada a *WSAAsyncSelect()* para cambiar los eventos del socket retornado por *accept()*, existe un periodo de tiempo durante el cual el socket reportado por *accept()* puede recibir notificación de los eventos heredados del socket de escucha.

La solución que propone la especificación a esta situación es la de asignar un evento al socket de escucha que no pueda ocurrir sobre un posible socket aceptado. Este evento es el FD_ACCEPT. Sobre un socket de escucha se recibirá notificación de que existe una petición de conexión pero sobre el socket aceptado no será posible recibir esta notificación ya que tal y como dice la especificación, un socket retornado por *accept()* no debería ser usado para aceptar otras conexiones. Además, el socket retornado por *accept()* ya está conectado al host remoto y no es posible recibir un FD_ACCEPT.

Ahora para entender las ideas del modelo orientado a conexión, veamos un diagrama de estados en el cual se puede encontrar un socket del tipo SOCK_STREAM. La figura es la siguiente:



En este diagrama de estados se presenta de una manera abreviada las posibles transiciones de un socket en modo SOCK_STREAM.

Primero hay que señalar que es algo complicado reflejar todas las posibles iteracciones de un estado a otro debido a su gran número. La intención de este diagrama es la de comprender un poco mejor el funcionamiento de una comunicación a través de un socket. A nivel de aplicación, que es donde no estamos moviendo siempre, sería muy necesario el añadir un estado "error", al cual se podría llegar desde cualquier otro estado si al intentar ejecutar una primitiva se produciese algún tipo de error. Exísten 44 tipos de errores diferentes contemplados en la especificación de W.S. Todo este conjunto de errores se puede dividir en dos grupos bien diferenciados; uno donde se contemplan todos los errores debidos a posibles anomalías que puediesen darse al fallar la red o las capas inferiores, y otro donde se incluyen todos aquellos errores debido a un mal uso de las primitivas, ya sea por un mal secuenciamiento de éstas (ej: no se puede intentar conectar un socket antes de crearlo) o por un posible error en algunos del los parámetros de las primitivas. El cómo se tratan estos errores y qué se decide hacer ante su presencia es muy particular de cada aplicación y tan sólo un pequeño conjunto de estos errores tienen un tratamiento igual independientemente de la aplicación. Debido a ésto no se considera oportuno el añadir este estado al diagrama ya que sería imposible representar todas las posibles interacciones y no nos aclararía nada.

Pasemos por tanto a explicar el diagrama en cuestión. Se parte de que exíste ya un socket creado; S. Este socket inicialmente se encuentra en un estado inactivo, no se puede hacer nada con él en este estado. Desde este estado exísten dos posibilidades, o bien irse al estado 2 o bien al estado 3. Si nos fijamos bien, en esta bifurcación es donde se decide si vamos a actuar como un servidor o como un cliente. Si nos acordamos de los capítulos Cliente/Servidor veremos esta situación más clara.

Empecemos por el modelo cliente. El socket S del estado inactivo puede pasar al estado 2 "pendiente confirmación de conexión" mediante la primitiva connect(). El socket está ahora esperando la confirmación del host remoto a su petición de conexión. En el momento que se reciba esta confirmación, el socket pasará al estado 4 y ya podrá iniciar una trasnferencia de datos con el host remoto. Desde este estado, la aplicación cliente puede cerrar la conexión mediante un shutdown(S,2) y volver al estado 1. Esta claro que el host remoto se trata de un servidor al que nos hemos conectado como un cliente.

Por otro lado, está la versión servidora. Partimos de nuevo del estado 1. Si recordamos algo del diseño de aplicaciones servidoras veremos que uno de los pasos que hay que dar es la de dejar al socket en modo pasivo. Esto se hace gracias a la primitiva listen() que se encarga de establecer una cola de peticiones. Pues mediante esta primitiva se accede al estado 3, donde se espera a la llegada de conexiones. Mientras no lleguen conexiones se permanecerá en este estado a no ser que se desee finalizar y se ejecute un shutdown() con lo que volveríamos al estado 1. En la versión servidora es necesario jugar con dos sockets, ya que uno S, el maestro, es el encargado de escuchar las peticiones, y otro S', el esclavo, es el encargado de atenderlas. Tras un accept() se crea el esclavo y se pasa al estado 4. En este estado ahora se pueden hacer transferencias de información a través de S' y no de S. Para salir de este estado y volver al de escucha es necesario destruir el socket esclavo S'. Sin embargo, si deseamos finalizar todo, también será necesario eliminar S' con closesocket(). Sólo la aplicación determinará hacia que estado desea volver, si al 1 o al 3, ya que la misma primitiva puede dar lugar a dos estados diferentes.

8.7. Breve análisis de las funciones contempladas en la especificación

Veamos ahora todas las funciones que se encuentran en la especificación dando una breve explicación de su funcionalidad.

↳ accept(*SOCKET s, struct sockaddr FAR* addr, int FAR* addrlen*)

La primera función que aparece es *accept()*. Esta función extrae la siguiente petición de conexión pendiente en la cola de peticiones establecida mediante la función *listen()*. Se suele aplicar en esquemas servidores para ir obteniendo o extrayendo las peticiones de los clientes.

Esta función al extraer una conexión pendiente crea un nuevo socket por el que se llevará a cabo la comunicación. Si no existen conexiones pendientes esta función producirá

bloqueo hasta que llegue alguna. Si por el contrario el socket es no-bloqueante, esta función retornaría el típico error WSAEWOULDBLOCK si no existiesen peticiones pendientes de extraer de la cola, avisando de esta forma que su ejecución produciría un bloqueo.

↳ bind(*SOCKET s, const struct sockaddr FAR* name, int namelen*)

La siguiente función es *bind()*. Su traducción literal es enlazar, y el significado que tiene enlazar un socket no es más que el de atribuirle o asignarle un nombre para que pueda ser identificado por otros programas.

Desde el punto de vista del programador, la función *bind()* es la encargada de asignarle una dirección determinada y un número de puerto a un socket. Se suele emplear en los esquemas servidores para asignar una dirección y puerto bien conocidos al socket por el que se esperan recibir las peticiones de conexión. También es aplicable en esquemas clientes aunque no es muy común. La razón de que no se apliquen en esquemas clientes es porque un cliente no debe utilizar una dirección y puerto bien conocidos. Pero esto no libra al socket cliente de enlazarse a una dirección local. Lo que ocurre es que existe otra función que tiene como efecto lateral el enlazamiento del socket a una dirección arbitraria y válida del sistema. Esta función es *connect()* la cual antes de iniciar una conexión observa si el socket por el que se desea establecer la conexión tiene asignado una dirección local. En caso de que no tenga asignado ninguna dirección, es decir, no se ha llevado a cabo ningún *bind()* sobre este socket, la función asigna una dirección local arbitraria a este socket de forma que ocupe un número de puerto no privado o reservado por el sistema, no utilice un puerto de un servicio estándar y que no utilice un número de puerto que está siendo utilizado actualmente por otro socket.

Suele ser muy interesante que sea el sistema el que elija la dirección local del socket en un programa cliente ya que libera al programador de la tarea de buscar un número de puerto que cumpliese los requisitos antes enumerados.

↳ closesocket(*SOCKET s*)

Esta función junto con la función *shutdown()* serán tema de análisis posteriormente debido a su similitud en significado. Ambas funciones cierran una conexión pero lo hacen de formas muy diferentes.

↳ connect(*SOCKET s, const struct sockaddr FAR* name, int namelen*)

Continuamos ahora con la función *connect()*. Como bien indica su nombre, esta función se encarga de establecer una conexión. Parece obvio que su ámbito se restrinja a modelos orientados a conexión pero la especificación permite realizar llamadas a *connect()* en modelos no orientados a conexión. En este último caso no se establece una conexión sino que se fija en el host local una dirección destino por omisión, así cuando se ejecuten *send()* o *recv()* se utilizará la dirección especificada en *connect()*.

Otro detalle importante de esta función es que si el socket especificado para conectarse no está enlazado previamente a una dirección, *connect()* lo enlazará a una dirección arbitraria tal y como se ha comentado con anterioridad.

↳ getpeername(*SOCKET s, struct sockaddr FAR* name, int FAR* namelen*)

Esta función se encarga de reportarnos el nombre o dirección del host remoto al que nos hemos conectado. Sólo es válida en modelos orientados a conexión, y dado un descriptor de un socket, que previamente se ha conectado con un host remoto, esta función retorna el nombre de dicho host.

↳ getsockname(*SOCKET s, struct sockaddr FAR* name, int FAR* namelen*)

Una función que a primera vista parece algo absurda suele cobrar importancia en algunos diseños. Esta función se encarga de reportar el nombre o dirección del socket especificado.

A veces, y sobre todo en esquemas clientes donde no importa demasiado la dirección que se le asigna al socket local, se suele enlazar el socket dejando que sea W.S. el

que elija la dirección de forma que no interfiera con direcciones ya utilizadas, privadas o que sean de servicios estándar. Recordemos que esto se conseguía como un efecto lateral de la función *connect()*.

Pues bien, en ciertas ocasiones es necesario obtener la dirección que le asignó el sistema al socket, y es aquí donde entra en juego la función *getsockname()*. Por poner un ejemplo: supongamos que diseñamos un programa cliente y cuando establecemos una conexión queremos mostrar al usuario las direcciones locales y remota. Pues para mostrar la dirección local es necesario actuar haciendo uso de *getsockname()*, ya que de otra forma no la podríamos obtener.

↳ g( )ckopt(*SOCKET s, int level, int optname, char FAR* optval, int FAR* optlen*)

La siguiente función, *getsockopt()*, se encarga de reportar el valor actual de una opción asociada con el socket indicado, sea éste del tipo que sea y esté en el estado que esté. Las opciones que soporta esta función están listadas en la especificación y ya se verán más adelante.

↳ Funciones de conversión

Ahora aparecen una serie de funciones encargadas de transformar un número de orden de red a orden de host y viceversa. Como ya se ha comentado, el orden en que se codifican los bits de un número difiere de unas máquinas a otras y por ejemplo la codificación en un PC es distinta que la utilizada por Internet. Debido a esta situación se hace necesario el uso de estas funciones, ***htonl(u_long hostlong)***, ***htons(u_short hostshort)***, ***ntohl(u_long netlong)***, ***ntohs(u_short netshort)***. Su significado se puede analizar como sigue:

Por ejemplo, la primera convierte un entero largo de orden de host a orden de red.

host_to_network_long()	htonl()
host_to_network_short()	htons()
network_to_host_long()	ntohl()
network_to_host_short()	ntohs()

Existen dos pares de funciones iguales, una para convertir enteros largos y otra para convertir enteros cortos.

Siguiendo con el tema de las conversiones, ya hemos visto que una dirección Internet suele darse en forma de punto y formato decimal y no binario. Pero también puede darse ya como un número que corresponde a la representación binaria de ésta (aunque no es lo normal) y se desee obtener su equivalente en formato punto tipo carácter. Para este fin la especificación ha incluido la siguiente función: *inet_ntoa()*.

↳ ioctlsocket(*SOCKET s, long cmd, u_long FAR* argp*)

La siguiente función, *ioctlsocket()*, se encarga de ejecutar ciertos comandos sobre un socket. Estos comandos son tres, FIONBIO, FIONREAD y SIOCATMARK.

El primero activa y desactiva el modo bloqueante y no-bloqueante del socket. El segundo se encarga de fijar la cantidad de datos que pueden ser leídos de forma automática desde el socket. Si éste es del tipo SOCK_STREAM entonces la función retorna la cantidad de datos total que podrían ser leídos con un único *recv()*; normalmente coincide con la cantidad de datos pendientes de ser leídos. Si por el contrario el socket es del tipo SOCK_DGRAM la función retorna el tamaño del primer datagrama pendiente en la cola.

Y por último, el comando SIOCATMARK determina si todos los datos fuera de banda han sido leídos o no. Este comando tan sólo es aplicable a sockets del tipo SOCK_STREAM que han sido configurados para la recepción de datos fuera de banda en línea. Si no existen datos fuera de banda para ser leídos, la operación retorna un valor lógico de TRUE. En cualquier otro caso retorna FALSE y el siguiente *recv()* o *recvfrom()* obtendrá todos o parte de los datos que preceden a la "marca".

EL sistema emisor de datos fuera de banda contemplado en esta especificación se basa en la utilización de un sólo canal para datos tipados y datos normales. Es decir, la idea de utilizar un canal paralelo para los datos fuera de banda no existe y se envían los datos fuera de banda "mezclados" con los datos normales. Para poder distinguirlos dentro de una

trama de datos se establecen una serie de "marcas" que son detectadas por W.S. Cuando se detecta una marca, la función anterior con el comando SIOCATMARK devuelve un valor FALSE. Es necesario advertir que al realizar un *recv()* o *recvfrom()* tan sólo se recibirá o bien datos fuera de banda o bien datos normales, pero NUNCA mezclados.

↳ listen(*SOCKET s, int backlog*)

Ahora viene una función muy importante en los esquemas servidores. Se trata de la función *listen()*. Esta función es la encargada de establecer un tamaño máximo para la cola de peticiones de conexión. Es decir, en los esquemas servidores es muy interesante limitar el número de usuarios que pueden acceder a él. Para este fin se ha desarrollado esta función, que limita el número de conexiones que puede aceptar. En cuanto se llena esta cola, subsiguientes peticiones por parte de clientes serán rechazadas indicándoles el error WSAECONNREFUSED.

↳ recv(*int s, char FAR* buf, int len, int flags*)

Hemos llegado ya a la función *recv()*. Como su nombre indica se encarga de recibir los datos a través de un socket. Es necesario que el socket esté conectado debidamente para que esta función actúe correctamente. Esta función retorna tanta información como este disponible hasta agotar el buffer suministrado. En caso de que no existan datos para leer y el socket este en modo bloqueante, esta función provocará bloqueo hasta que se reciban los datos. Esta función se utiliza en sockets del tipo SOCK_STREAM pero puede ser usada en sockets del tipo SOCK_DGRAM si previamente se ha indicado alguna dirección por defecto mediante la función *connect()* tal y como ya se comentó.

↳ recvfrom(*int s, char FAR* buf, int len, int flags, struct sockaddr FAR* from, int FAR* fromlen*)

La función *recvfrom()* es muy parecida a la anterior pero su uso se centra más en modelos no orientados a conexión. Gracias a esta función, es posible recibir un datagrama y obtener además la dirección del que lo envió. Es muy útil cuando se están recibiendo datagramas de diferentes fuentes, para de esta forma poder identificar el "dueño" del datagrama recibido.

↳ select(*int nfds, fd_set FAR* readfds, fd_set FAR* writefds, fd_set FAR* exceptfds, const struct timeval FAR* timeout*)

Una función de gran importancia es la función *select()*. Se encarga de reportar al programa si un socket determinado esta disponible para leer, escribir, recibir datos tipados o algunas condiciones excepcionales de error. El funcionamiento de esta función es como sigue. Por si sola provoca bloqueo pero tiene un mecanismo que permite controlar este bloqueo. Como argumentos se le pasa el conjunto de descriptores para analizar la disponibilidad de lectura, el conjunto del ellos para la de escritura y el conjunto para opciones excepcionales como son los datos fuera de banda. Estos conjuntos de sockets se le pasan haciendo uso de una estructura llamada fd_set. Además se le pasa como último parámetro un timeout a través de una estructura timeval. Todas estas estructuras están contempladas en el fichero de cabecera winsock.h.

Pues bien, la función *select()* produce bloqueo hasta que alguno de los sockets cumpla la condición seleccionada o hasta que expida el timeout especificado. Pero existen dos casos extremos que comento ahora. Si no se especifica ningún timeout, es decir, el parámetro timeout es NULL, entonces la función *select()* bloqueará indefinidamente hasta que alguno de los sockets cumpla la condición.

Y si se especifica como timeout el par {0,0} la función *select()* no bloqueará y reportará un resultado de inmediato. Esta última opción sirve para hacer "polling" a través de los diferentes sockets en uso.

↳ sent() y sendto()

Siguiendo el orden establecido en la especificación, ahora vienen las funciones *send()* y *sendto()* cuyo significado es el mismo y se diferencian entre ellas igual que *recv()* y *recvfrom()*. Advertir tan sólo que la opción de realizar un broadcast no es posible más que en modo datagrama, es decir, con un socket del tipo SOCK_DGRAM. Para ello se utiliza *sendto()* especificando como dirección IP, INADDR_BROADCAST, y sin olvidar especificar el número de puerto sobre el que será recibido el broadcast.

↳ setsocktopt(*SOCKET s, int level, int optname, const char FAR* optval, int optlen*)

Veamos ahora la función *setsockopt()*. Gracias a ella vamos a poder habilitar la transmisión de mensajes broadcast, recibir datos fuera de banda dentro de los datos normales, especificar el tamaño del buffer de recepción, permitir que se pueda enlazar un socket a una dirección usada ya por otro socket, especificar el tamaño del buffer de recepción, desactivar el algoritmo de *Nagle*, etc... Todas estas opciones están detalladas en la especificación y destacar tan sólo dos.

La primera sobre la opción SO_REUSEADDR que permite usar una misma dirección y puerto por varios sockets. Esto es posible debido a que una conexión queda caracterizada por las direcciones de ambos extremos, permitiéndose que dos sockets disfruten de una misma dirección local siempre que la dirección remota sea diferente. De esta manera no existirá ambigüedad ninguna sobre de quién son los datos recibidos, ya que sólo habrá que analizar de qué host remoto provienen para saber a qué socket local pertenecen.

La segunda se trata sobre el algoritmo de *Nagle*. Este algoritmo se encarga de evitar enviar muchos paquetes de reducido tamaño. Para ello se espera hasta que se complete un paquete entero antes de ser enviado y no realizar un envío por cada *send()* que ejecuta la aplicación. Pero a veces y según el protocolo de aplicación es necesario que se envíe estos paquetes pequeños para un correcto funcionamiento y se hace necesario desactivar este algoritmo. Este algoritmo se desactiva mediante la opción TCP_NODELAY.

La especificación advierte que no se debe desactivar el algoritmo de Nagle a menos que sea muy necesario debido a su gran impacto sobre la red y su funcionamiento.

↳ socket(*int af, int type, int protocol*)

Llegamos ya a la función *socket()*. Esta función es la encargada de crear un socket de un tipo, SOCK_STREAM o SOCK_DGRAM, y bajo una familia de direcciones específica (en el caso de internet es AF_INET), haciendo uso de un protocolo determinado. Respecto al tipo de protocolo utilizado, se suele especificar un valor 0 para que sea W.S. el que seleccione el protocolo adecuado con las opciones anteriores. Por ejemplo, para un

socket del tipo SOCK_STREAM el protocolo a nivel de transporte utilizado es el ya comentado TCP.

Ahora vienen las denominadas funciones de base de datos. Analizaremos tan sólo las funciones heredadas del UNIX ya que las nuevas funciones *WSAAsyncGetXByY()* para Windows son iguales y tan sólo difieren en su funcionamiento que ya fue explicado.

↳ gethostbyaddr(*const char FAR* addr, int len, int type*)

La primera función es *gethostbyaddr()*. Esta función retorna información acerca de un host a partir de su dirección Internet. La información se retorna bajo una estructura hostent en la que se agrupa el nombre oficial del host, un conjunto de alias del host, el tipo de dirección y su tamaño y una lista de posibles direcciones del host.

↳ gethostbyname(*const char FAR* name*)

La segunda función es *gethostbyname()* y su misión es idéntica a la anterior pero con la salvedad que ahora especificamos el nombre del host y no su dirección internet. La estructura que obtenemos es la misma, hostent.

La función *gethostname()* retorna el nombre del host local. Se suele utilizar para obtener la dirección IP del host local. Para ello primero obtenemos el nombre del host local haciendo uso de esta función y con el nombre obtenido y la función *gethostbyname()* obtenemos su dirección IP.

↳ getprotobyname(*const char FAR* name*)

La cuarta función contemplada en la especificación es *getprotobyname()* y sirve para obtener el número asignado a un determinado protocolo a partir de su nombre. Esta función devuelve una estructura protoent en la que se especifica el nombre oficial del protocolo, una serie de alias y el número asignado a dicho protocolo.

↳ getprotobynumber(*int number*)

La función inversa a la anterior es *getprotobynumber*() y no es más que obtener el nombre de un protocolo a partir de su número. La estructura utilizada es la misma que anteriormente.

Existen una serie de servicios estándar en Internet como pueden ser FTP, GOPHER, ... que tiene asignados unos número de puerto concreto.

↳ getservbyname(*const char FAR* name, const char FAR* proto*)

Pues bien, para ayudar al programador en la tarea de obtener dichos puertos surgió la función *getservbyname*() que dado un nombre de servicio y el tipo de protocolo utilizado para acceder a él, retorna una estructura del tipo servent donde se registra entre otros datos el número de puerto que tiene asignado ese servicio. Es bastante típico que antes de conectarnos a un servicio bien conocido, pidamos mediante esta función que nos sea reportado el número de puerto donde actúa ese servicio.

↳ getservbyport(*int port, const char FAR* proto*)

Y como siempre aparece la función inversa. Ahora se trata de la función *getservbyport*() que nos retorna el nombre del servicio bien conocido que tiene asignado el puerto indicado. Se devuelve también la información a través de una estructura servent, en la que aparece entre otros datos el nombre oficial del servicio que trabaja sobre el puerto especificado.

Ya hemos terminado con las funciones heredadas del UNIX. Ahora entramos en las funciones específicas de Windows.

↳ WSAAsyncSelect(*SOCKET s, HWND hWnd, unsigned int wMsg, long lEvent*)

Omitiendo las funciones de base de datos que son idénticas a las ya comentadas, llegamos a la función *WSAAsyncSelect()* que ya ha sido comentada anteriormente.

↳ WSACancelAsyncRequest(*HANDLE hAsyncTaskHandle*)

Por tanto, pasamos a la función WSACancelAsyncRequest(). Como ya vimos, las versiones asíncronas de las funciones de base de datos reportan un ATH (Async. Task Handle) o manejador de tarea asíncrona. Cuando nosotros lanzamos una tarea de forma asíncrona podemos querer detenerla antes de que finalice, ya sea por cuestiones de finalización del programa o por otras razones. Es en este punto donde entra en juego la función *WSACancelAsyncRequest*(). La llamada a esta función indicando el ATH de una tarea específica provoca que la tarea sea finalizada de forma abrupta; en otras palabras, se destruye la tarea.

Dentro de las funciones de destrucción de tareas existe otra que se encarga de cancelar cualquier llamada bloqueante de W.S. La función es ***WSACancelBlockingCall***(). Por ejemplo, si hacemos un *recv()* y produce bloqueo y el usuario desea abandonar la aplicación, es necesario llamar a esta función para cancelar el *recv()* que se había ejecutado anteriormente y que esta produciendo bloqueo. Otra función que se suele utilizar conjuntamente con ésta es la ***WSAIsBlocking()*** que determina si existe alguna función W.S. bloqueante en curso. En caso afirmativo, esta función se puede anular mediante ***WSACancelBlockingCall***().

↳ WSACleanup(*void*)

Ya estamos llegando al final de las funciones contempladas en la especificación. La que continua es *WSACleanup*() se encarga de desinstalar winsock.DLL. Es necesario advertir que cualquier socket conectado será reseteado si se llama a esta función y que cualquier información pendiente de ser enviada en los buffers será enviada sin problemas. Existe la función *WSAStartup*() que se encarga de inicializar winsock.DLL y prepararlo para su uso. Pues bien, con WSAStartup() se comienza cualquier programa Winsock y con *WSACleanup*() se finaliza. Estas funciones son obligatorias y además por cada llamada a WSAStartup() se debe hacer después una llamada a *WSACleanup*().

↳ WSAGetLastError(*void*)

Hasta ahora no hemos hablado nada de los errores que se pueden producir dentro de una aplicación y su tratamiento. El tratamiento difiere mucho de un error a otro pero la

forma de obtener los códigos de errores es siempre la misma. La especificación dispone de una función que se encarga de reportar el código del último error ocurrido. Esta función es *WSAGetLastError()*. Si bien la respuesta de esta función es un número que identifica a un error, de cara al programador sería interesante realizar una relación biunívoca entre código de error numérico y una cadena de caracteres que describa el error. De esta forma se mostraría al usuario una descripción del error y no un simple código numérico de error que no le serviría de nada.

Dentro del tratamiento de errores la especificación incluye una función más que permite fijar el código de error que será retornado por una subsiguiente llamada a *WSAGetLastError()*. Esta función es *WSASetLastError*(). Advertir que cualquier llamada a una función W.S. después de *WSASetLastError*() sobreescribirá otro tipo de error.

Las funciones ***WSASetBlockingHook***() y ***WSAUnhookBlockingHook***() ya han sido comentadas anteriormente y por tanto serán omitidas ahora.

↳ **WSAStartup(*WORD wVersionRequired, LPWSADATA lpWSAData*)**

Resta finalmente analizar la función *WSAStartup*(). Esta función debe ser siempre la primera función que se llame en cualquier programa que utiliza W.S. Esta función es la encargada de negociar con nuestra DLL la versión que será utilizada. La DLL soportará una serie de versiones de Windows Sockets. Soportará desde una mínima hasta una máxima. Si la versión requerida por *WSAStartup*() es mayor que la mínima de la DLL entonces todo funcionará correctamente.

8.8. Estructuras de datos importantes bajo Windows Sockets

Para poder fromarse una idea bastante buena de las funciones, deberemos analizar también las estructuras de datos que son usadas por las funciones Windows Sockets. Estas estructuras se encuentran detalladas en el fichero de cabecera de Windows Sockets; alguna de ellas ya se ha visto en capítulos anteriores.

La primera estructura que analizaremos es fd_set. Esta estructura se utiliza para pasarle a la función *select()* el conjunto de sockets que se quieren que sean chequeados para lectura, escritura, ... Tiene la siguiente definición:

```
typedef struct fd_set {
        u_short fd_count;   /* Número de sockets incluidos */
        SOCKET fd_array[FD_SETSIZE]  /* Matriz */
} fd_set;
```

Como se puede observar esta estructura consta de dos campos. En el primero se especifica el número de sockets que contiene la matriz. En el segundo se encuentra la matriz de 64 elementos. Como el array es una estructura estática de 64 elementos, deberemos especificar de alguna forma el número de sockets que hay dentro de ese array, por lo cual aparece el primer campo.

Otra estructura utilizada en la función *select()* es timeval. Tiene la siguiente forma:

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

Esta estructura ya se ha visto anteriormente y por tanto los comentarios sobre ella sobran, al igual que con las estructuras hostent, servent y protoent.

Ahora sigue la estructura in_addr. Esta estructura se compone de una unión. Una estructura del tipo union en C++ es como una estructura normal salvo que en cada instante sólo puede tener uno de sus campos en uso. Es decir, si tiene 4 campos en los que se definen 4 tipos diferentes de datos, en realidad sólo estará activo un tipo a la vez.

La estructura se define como sigue:

```
struct in_addr {
        union {
                struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;
                struct { u_short s_w1,s_w2; } S_un_w;
                u_long S_addr;
            } S_un;
        #define s_addr  S_un.S_addr
        #define s_host  S_un.S_un_b.s_b2
        #define s_net   S_un.S_un_b.s_b1
        #define s_imp   S_un.S_un_w.s_w2
        #define s_impno S_un.S_un_b.s_b4
        #define s_lh    S_un.S_un_b.s_b3
};
```

Además en el fichero de cabecera se definen una serie de datos de esta estructura de entre los que cabe destacar el siguiente:

```
#define s_addr S_un.S_addr    /* Valor usado por aplicaciones TCP/IP */
```

Así, la estructura in_addr se compone de una unión de tres elementos. Por tanto, podremos definir una dirección IP de tres formas diferentes aunque como ya se ha dicho la más utilizada es la representación como un u_long.

Esta estructura aparece dentro de otra estructura que es la que utilizaremos para definir una dirección final de forma completa. Esta otra estructura es:

```
struct sockaddr_in {
        short sin_family;  /* Especifica la familia de direcciones del protocolo */
                           /* En caso de Internet y TCP/IP utilizaremos el AF_INET
*/
        u_short sin_port; /* Especifica el número de puerto donde nos vamos a
conectar */
        struct in_addr sin_addr;   /* Dirección IP */
        char sin_zero[8];  /* 8 bytes libres */
};
```

Otra estructura utilizada es WSAData. Esta estructura nos retorna cierta información sobre la pila de protocolos TCP/IP instalada al inicializar Windows Sockets mediante la función WSAStartup(). Su forma es la siguiente:

```
typedef struct WSAData {
        WORD            wVersion;
        WORD            wHightVersion;
        char            szDescription[WSADESCRIPTION_LEN+1];
        char            szSystemStatus[WSASYS_STATUS_LEN+1];
        unsigned short  iMaxSockets;
        unsigned short  iMaxUdpDg;
        char FAR*       lpVendorInfo;
} WSADATA;
```

Analicemos ahora cada campo de esta estructura.

wVersion: Es la versión de la especificación Windows Sockets que la DLL espera que utilice el usuario.

wHighVersion: Es la mayor versión que la DLL puede soportar. Normalmente conincidirá con wVersion.

szDescription: Es una cadena de caracteres acabada con un carácter NULL sobre la que la DLL copia una descripción de la implementación de Windows Sockets, incluyendo la información e identificación del vendedor. El texto que puede alcanzar hasta 256 caracteres debe contener caracteres y por tanto los vendedores deben tener cuidado de no incluir en ella caracteres de formato y control ya que el posible uso que se le de a este campo es imprimirlo como un mensaje de estado al cargar Windows Sockets con WSAStartup().

szSystemStatus: Es también una cadena de caracteres acabada con el carácter NULL donde la DLL copia información sobre el estado o la configuración. La DLL deberá

utilizar este campo sólo si la información es importante de cara al usuario. Este campo no se debe considerar como una extensión del campo szDescription.

iMaxSockets: Es el número máximo de sockets que un único proceso puede abrir.

iMaxUdpDg: Es el tamaño en bytes de la longitud máxima de un datagrama UDP que puede ser recibido o enviado.

lpVendorInfo: Es un puntero lejano a un estructura específica del vendedor. La definición de esta estructura en caso de que exisitese va más allá de la definición de la especificación Windows Sockets 1.1.

La siguiente estructura y que esta relacionada con la especificación de una dirección es sockaddr. Esta estructura es la más general que contempla Windows Sockets en cuestión de direcciones. Como Windows Sockets pretende soportar varios tipos de protocolos, deberá por tanto también soportar diferentes tipos de direcciones. Dentro de cada protocolo existirá una representación diferente de las direcciones. Debido a esto, se creo la siguiente estructura en la cual en su primer campo se especifica el tipo de familia de direcciones que estamos usando y en el segundo campo se dejan 14 bytes para la escritura de la dirección en su formato correspondiente y según la familia de direcciones del protocolo que se desee usar.

```
struct sockaddr {
        u_short sa_family;
        char sa_data[14];
};
```

Otra estructura que mencionaremos es *linger*. Es bastante simple y su significado se verá en la explicación de closesocket() y shutdown().

8.9. Diferencias entre *closesocket()* y *shutdown()*

Esta funciones sirven para cerrar una conexión o parte de ella. La primera, *closesocket()*, realiza un cierre total de la conexión, liberando todos los recursos asociados a un socket incluyendo al propio socket. La segunda, *shutdown()*, se encarga de deshabilitar la recepción de datos o de emisión o de ambas cosas. A diferencia con la función *closesocket()*, *shutdown()* no libera los recursos asociados con el socket, mantiene el descriptor del socket válido.

La especificación desaconseja la reutilización de un socket cerrado mediante un *shutdown()* aunque no prohibe que cualquier vendedor de software implemente esta opción en su DLL.

Además, *shutdown()* no tiene efecto sobre las capas inferiores y todos los datos que lleguen serán aceptados y en ningún caso se generará un paquete de error ICMP. Esta función sólo tiene efecto a nivel de aplicación.

8.10. Algunos Errores de Interés

Existen como ya se ha dicho anteriormente 44 tipos de errores diferentes y cuyo listado se encuentra en la especificación. Aquí tan sólo nos vamos a referir a algunos errores que suelen ser comunes o que deben ser tenidos en cuenta.

La idea de realizar este listado se debe a que no existe en la especificación ningún anexo donde a demás de recoger los tipos de errores se dé una explicación breve de cada uno de ellos.

Veamos por tanto los errores, indicando su identificativo y un breve comentario.

WSAEINTR: Nos indica que la llamada que producía bloqueo ha sido interrumpida mediante la función *WSACancelBlockingCall*().

WSANOTINITIALIASED: Avisa de que se ha intentado ejecutar alguna función de la especificación sin previamente haber llevado a cabo una correcta carga de W.S. mediante *WSAStartup*().

WSAENETDOWN: La implementación de W.S. ha detectado que el subsistema de red ha fallado y no se encuentra operativo.

WSAEFAULT: Este error suele aparecer cuando se ha indicado la longitud de un buffer como parámetro y éste es demasiado pequeño para poder almacenar el resultado de la función.

WSAEINPROGRESS: Advierte que ya existe una llamada bloqueante en curso y por tanto no se puede llamar a ninguna otra función de W.S.

WSAEINVAL: No se ha ejecutado un *listen()* previamente al *accept()*. Este error es reportado por la función *accept()* cuando anteriormente a ésta no se ha ejecutado un *listen()*.

WSAENOBUFS: No se dispone de espacio en los buffers.

WSAENOTSOCK: El descriptor pasado a la función no se trata de un socket.

WSAEOPNOTSUPP: El socket referenciado no es del tipo SOCK_STREAM y por tanto no soporta esquemas orientados a conexión.

WSAEWOULDBLOCK: Cuando un socket esta marcado como no bloqueante y se intenta ejecutar una función que va ha producir bloqueo, se obtiene este error.

WSAEADDRINUSE: Este error nos comunica que la dirección especificada a la función *bind()* ya está en uso. Como vimos, es posible asignar la misma dirección a varios sockets pero es necesario antes configurar esta situación haciendo uso de la función *setsockopt()* con la opción SO_REUSEADDR.

WSAEAFNOSUPPORT: Este error nos indica que la familia de direcciones indicada no soporta el protocolo elegido. Este error se suele dar en la función *bind()*.

WSAEADDRNOTAVAIL: La dirección indicada no esta disponible desde la máquina local. Este error se reporta al ejecutar la función *connect()*.

WSAECONNREFUSED: Este error indica que el intento de conexión ha sido rechazado por la máquina remota.

WSAEDESTADDREQ: Se ha omitido el parámetro de la dirección destino y es necesario indicarlo.

WSAEISCONN: Indica que el socket especificado ya está conectado actualmente y que por tanto no es posible conectarlo de nuevo. Si se desea conectarlo a otro lugar es necesario cortar la primera conexión con *shutdown()* o *closesocket()*.

WSAENETUNREACH: El host no puede acceder a la red en ese instante en el que se ejecutó la función W.S.

WSAETIMEOUT: En el intento de conectarse con un host remoto se ha excedido el *timeout* fijado por W.S. sin establecerse la conexión.

WSAENOPROTOOPT: Tras la ejecución de *setsockopt()* se puede dar este error indicando que la opción indicada no existe o que no es aplicable al socket. Por ejemplo, la opción SO_BROADCAST no es aceptada en un socket SOCK_STREAM y las opciones SO_ACCEPTION, SO_DONTLINGER, SO_KEEPALIVE, SO_LINGER y SO_OOBINLINE no son soportadas por sockets del tipo SOCK_DGRAM.

WSAENOTCONN: El socket no está conectado todavía y es necesario para que se ejecute correctamente la función que el socket este conectado.

WSAESHUTDOWN: Este error aparece cuando se intenta recibir o enviar datos después de haber efectuado un *shutdown()* sobre el socket cortando el flujo de datos en el sentido de recepción, emisión, o en ambos sentidos.

WSAEMSGSIZE: El datagrama ha sido demasiado grande como para poder ser almacenado en el buffer y se ha truncado.

WSAECONNABORTED: El circuito virtual establecido se ha abortado debido a un timeout o fallo de la red.

WSAECONNRESET: El circuito virtual ha sido reseteado por el extremo remoto.

WSAENETRESET: La conexión debe ser reseteada debido a que W.S. la ha tirado abajo.

WSAENOBUFS: W.S. reporta un problema de *deadlock* en los buffers.

WSAESOCKNOSUPPORT: El socket especificado no es del tipo soportado por la familia de direcciones indicada.

WSAEPROTOTYPE: Indica que el tipo de protocolo indicado no es válido dentro de la familia de direcciones especificado.

WSAHOST_NOT_FOUND: Indica que el host indicado no se ha encontrado. Este error se da en las funciones de base de datos.

Existen todavía más errores pero considero que los más comunes e importantes los he comentado arriba.

8.11. Modelos Cliente/Servidor bajo Windows

Es hora de comentar de forma muy breve posibles implementaciones y restricciones que pueden darse al trasladar las ideas de los modelos Cliente/Servidor, comentadas en los capítulos anteriores, del entorno UNIX al entorno Windows.

En el capítulo de la programación en Windows vimos que éste no es un sistema multiproceso sino que era un sistema multitarea y no una multitarea muy efectiva sino que era una multitarea sin prioridades. Es decir, nosotros podemos implementar la multitarea en un sistema monoprocesador asignando franjas de tiempo a cada tarea abierta tal y como vimos. El sistema operativo es el que en última instancia conmuta de una tarea a otra cuando se ha acabado la franja de tiempo correspondiente. En Windows esto no ocurre así, quién decide cuándo conmutar a otra tarea es la propia tarea que está siendo ejecutada en ese instante. Hasta que una tarea no "suelte" el control del sistema, otras tareas que estén abiertas no podrán cogerlo. Analizando de una forma más detallada podemos resolver que en última instancia quién conmuta de una tarea a otra en Windows es el propio usuario.

De esta propiedad, arriba comentada, se pueden obtener conclusiones sobre algunas restricciones que nos ofrece esta multitarea. En primer lugar no vamos a poder jugar con la posibilidad de crear procesos paralelos para atender a varios clientes de forma concurrente, ya que en cuanto cambio de tarea la tarea que abandoné queda "muerta" sin recibir mensajes hasta que el usuario vuelva a interacturar sobre ella.

En principio parece que lo dicho anteriormente es aceptable y correcto pero si pensamos más a fondo sobre esta forma de operar, podremos descubrir que es posible tener varias tareas abiertas a la vez y además ejecutándose todas.

Supongamos n tareas abiertas. Windows les da vida gracias al flujo de mensajes, es decir, las tareas están programadas para responder a esos mensajes y en cuanto Windows despacha alguno a una tarea, ésta ejecuta la porción de código correspondiente en respuesta a ese mensaje. Así, si Windows despacha un primer mensaje a la tarea 1, el siguiente a la 2 y así sucesivamente lo que ocurrirá es que se irá conmutando de la tarea 1 a la 2, de la 2 a la 3, etc... Por tanto, el tiempo que se le asigna a cada tarea depende de los mensajes

pendientes en la cola, a quién van dirigidos y el tiempo q tarda la tarea en atenderlos. Como muchos mensajes de la cola son producto de actuaciones del usuario sobre Windows, al final quien de alguna manera conmuta es el usuario pero no debemos olvidar que hay otros mensajes que no son creados o provocados por el usuario.

Si nos centramos ahora en el tema de los sockets, podremos tener varias aplicaciones de red abiertas y ejecutándose todas a la vez, aunque sólo una de ellas esté activa. Esto ocurre de la siguiente forma. Supongamos que tenemos dos aplicaciones abiertas. En cada aplicación disponemos de un socket ya conectado llamado s y s' respectivamente. Ahora, para cada aplicación y sobre s y s' activamos la notificación de eventos mediante *WSAAsyncSelect()*. Cuando me sea posible realizar una lectura sobre s, la aplicación 1 recibirá ese mensaje y leerá. Por otro lado si los datos están disponibles en s', la aplicación 2 recibirá el mensaje y los leerá. Si ambos datos llegan simultáneamente, se conmutará a la aplicación en función de qué mensaje ha ganado la cola primero.

Así, podemos ver que es posible realizar un modelo servidor concurrente en Windows si utilizamos bien la multitarea. Una posible solución sería crear un programa que en vez de crear un proceso hijo, crease una ventana hija oculta. Es decir, por cada nueva conexión que se aceptase, nuestro programa crearía una ventana oculta donde se llevaría a cabo la comunicación con el cliente o el procesamiento de una respuesta simplemente.

En esta solución hay que destacar el problema de limitar el número de ventanas ocultas que creamos para no abusar de los recursos del sistema (Cuantas más ventanas abiertas tengamos en Windows más se relentizará el sistema). Además es necesario crear una asociación entre el manejador de ventana oculta con el descriptor del socket (en otras palabras, realizar un mapeado manejador⇒socket), porque va a ser necesario hacer uso de la función *WSAAsyncSelect()* en la que se deben especificar descriptor, manejador ventana que recibirá las notificaciones y los eventos de red. De esta forma, dentro de una misma tarea tenemos varias minitareas que se van ejecutando según vayan recibiendo notificaciones por parte de *WSAAsyncSelect()* o por parte de la ventana principal.

Dentro de un esquema servidor es necesario implementar la posibilidad de que un "supervisor" o usuario pueda cerrar las ventanas ocultas creadas y por tanto cerrar las conexiones con determinados clientes desde el servidor. Es decir, en nuestro programa servidor sería interesante monitorizar todas las conexiones que hay establecidas en ese instante para que el usuario tome cuenta de ello y tenga la posibilidad de cerrar alguna de las conexiones existentes actualmente.

Respecto a las demás implementaciones comentadas en los capítulos anteriores sobre el modelo Cliente/Servidor no hay nada particular y no es más que trasladar los conceptos allí dados al entorno Windows y su filosofía basada en los mensajes.

Así, el concepto de proceso concurrente en Windows hay que tratarlo con cuidado. Es el flujo de mensajes el que da vida a cualquier aplicación Windows, por tanto, si no queremos que una aplicación muera deberemos asegurarle mensajes.

8.12. Comentario final sobre Windows Sockets

Como análisis final sobre la especificación Windows Sockets decir que en su versión 1.1 está muy limitada respecto a los protocolos que puede utilizar aunque prevé la posibilidad de la utilización de muchos más. Actualmente los fabricantes de pilas de protocolos tan sólo implementan en sus winsock.DLL el protocolo TCP/IP, dejando los restantes que contempla la especificación al margen.

Es necesario destacar que en esta nueva versión aparecen conceptos muy novedosos como poder soportar simultáneamente y a través de una única DLL varios protocolos de transporte; además debido a esta novedad es necesario crear nuevas funciones para poder averiguar de qué protocolos se dispone y obtener información sobre ellos. La estructura WSAData queda ya obsoleta ya que tan sólo reporta información sobre un único protocolo de transporte. También se permite que se compartan sockets entre tareas, que se creen grupos de sockets con ciertos atributos comunes. También se introduce la noción de calidad del servicio, QOS, y una serie de detalles más que si bien lo hacen notoriamente más completo también lo hace más complejo y diferente respecto a la versión 1.1.

Capítulo 9

WINDOWS SOCKETS ver. 2.0

9.1. Status

Actualmente (finales de 1995) la Versión 2 de Windows Sockets está todavía en desarrollo. Esta versión representa una gran adelanto respecto a la versión 1.1 y deja constancia del interés existente hoy en día por una serie de compañías en que esta nueva versión salga adelante y sea bastante potente, intentando convertirla realmente en una Interfase para la Programación de Red Transparente.

Desde que el Grupo Winsock comenzó la Versión 2 de la especificación en Mayo de 1994, cientos de personas pertenecientes a una serie de compañías y organizaciones han cooperado y contribuido conjuntamente en el diseño y desarrollo de esta especificación. Varias reuniones, correos electrónicos y últimamente conversaciones telefónicas han tenido lugar durante todo este periodo de tiempo con el fin de lograr una nueva especificación muy completa. La existencia de una lista de correo permitió y permite actualmente la participación de un gran número de personas en el diseño, aportando ideas, comentarios o aclaraciones sobre diferentes temas que al final tendrán su reflejo en la especificación final.

Por otro lado, y a parte de la contribución desde la lista de correo, se crearon una serie de grupos de funcionalidad que se reparten el trabajo que acarreaba el diseño de la especificación. Dentro de estos grupos existe un líder o moderador que se encarga de coordinar los esfuerzos de los integrantes del Grupo de Funcionalidad. La siguiente tabla aclara qué compañías han intervenido y en qué grupos de funcionalidad se ha dividido el trabajo, así como los nombre de los líderes de cada grupo de funcionalidad. Es de destacar la participación de compañías tan importantes como Intel, Microsoft, Motorola ó Novell, lo que da una idea del interés actual sobre este software.

Grupo de Funcionalidad	Líderes	Email	Compañía
Generic API	Dave Andersen	David_B_Andersen@ccm.jf.intel.com	Intel

Operating Framework	Keith Moore	keithmo@microsoft.com	Microsoft
Specification Clarifications	Bob Quinn Vikas Garg Paul Brooks	rcq@ftp.com vikas@distinct.com brooks@turbosoft.com	FTP Software Distinct Turbosoft
Name Resolution	Margaret Johnson	margretj@microsoft.com	Microsoft
Connection-Oriented Media	Charlie Tai Sanjay Agrawal	Charlie_Tai@ccm.jf.intel.com sanjaya@microsoft.com	Intel Microsoft
Wireless	Dale Buchholz	drbuchholz@mot.com	Motorola
TCP/IP	Michael Khalandovsky	mlk@ftp.com	FTP Software
IPX/SPX	Tim Delaney	tdelaney@novell.com	Novell
DECnet	Cathy Bence	bence@ranger.enet.dec.com	DEC
OSI	Adrian Dawson	ald@oasis.icl.co.uk	ICL

De la anterior tabla se observa que la principal y única empresa involucrada en el desarrollo de la API es INTEL.

En una reciente conversación con David B. Andersen vía *e-mail* que data del 21-Nov-1995 he podido saber que todavía no se dispone de ninguna Winsock2.DLL para Windows 95 pero que estará disponible pronto. David B. Anderson me comunica por otro lado que actualmente esta en su conocimiento los esfuerzos que se están llevando a cabo para obtener proveedores del servicio tales como IPX/SPX, DecNet, TP4 entre otros. Ya veremos qué son los proveedores del servicio. Es un nuevo concepto necesario que se tuvo que introducir en Winsock 2 para poder soportar varios protocolos de Transporte.

David B. Andersen termina aclarándo que la especificación se encuentra en un estado provisional en el cual se prevé realizar cambios mínimos sobre ella para añadir claridad y solventar algunos problemas que surjan durante la construcción del SDK (Software Developmment Kit). Se estima según él, que la especificación esté lista y funcionando correctamente a mediados de 1996.

9.2. ¿Qué es Winsock 2?

Veamos ahora qué nuevas ideas aporta la versión 2. Esta nueva versión de la especificación es una ampliación de la interface Windows Sockets ver 1.1. que actualmente

está muy extendida. Esta nueva versión, manteniendo la compatibilidad con la ver. 1.1 extiende la interface Winsock en un gran número de áreas de entre las que cabe destacar:

- el acceso a otros protocolos distintos a TCP/IP: Winsock 2 permite que una aplicación pueda utilizar la interfase socket para llevar a cabo accesos simultáneos a cualquier número de protocolos diferentes de transporte que se encuentren debidamente instalados en nuestra máquina y registrados con Winsock 2.

- las facilidades en la resolución de nombres independientemente del protocolo de transporte utilizado: Winsock 2 introduce una serie de nuevas funciones de la API estandarizadas para poder trabajar con los numerosos dominios de resolución de nombres existentes hoy en día como son DNS, SAP, X.500, etc.., de una forma genérica.

- operaciones I/O asíncronas (ó superpuestas; overlapped): siguiendo el modelo establecido en entornos win32, Winsock 2 incorpora el paradigmà "*overlapped*" para operaciones de entrada salida asíncronas con sockets.

- especificación de la calidad del servicio: Winsock 2 establece convenciones para que las aplicaciones negocien los niveles requeridos del servicio con parámetros tales como ancho de banda, latencia, etc..

- posibilidad de realizar multicast y multipoint mediante funciones generales de la API independientes del protocolo de transporte utilizado, es decir, la existencia de un método para que una aplicación pueda saber de las capacidades de la capa de transporte sobre la que trabaja para realizar un multicast o multipoint, y en caso afirmativo, realizar éstos de una forma genérica, independientemente del protocolo de transporte.

- y finalmente otras extensiones frecuentemente requeridas desde la lista de correo como: compartición de sockets, aceptación de conexiones condicionadas, intercambio de datos en el establecimiento de una conexión o en la desconexión, posibilidad de agrupar sockets con unas características iguales, etc..

Todas estas innovaciones serán vistas en el presente capítulo para dar un idea más clara del futuro de Winsock 2. Lo que si se evitará es realizar una descripción detallada de las nuevas funciones de la API y tan sólo comentar las líneas generales en las que se mueve esta nueva especificación.

9.3. ¿A quién va dirigido la especificación Winsock 2?

Dentro de esta especificación hay que distinguir tres documentos de los cuales al programador tan sólo le interesa uno:

1. **Windows Sockets 2 Application Programming Interface**
2. **Windows Sockets 2 Protocol-Specific Annex**
3. **Windows Sockets 2 Service Provider Interface**

Las personas que están interesadas en el diseño de aplicaciones que hagan uso de las capacidades que ofrece Winsock 2 deberán por tanto familiarizarse con la especificación de la API.

Sin embargo, todas aquellas personas que están interesadas en la realización de un protocolo de transporte particular que sea accesible vía Winsock 2, deberán familiarizarse primero con la especificación SPI (*Service Provider Interface*). El documento de la SPI especifica la interfase que los proveedores de servicio deben cumplir para poder ser accesibles vía Windows Sockets ver. 2. Estos documentos, se encuentran a diferentes niveles.

Y por último, el documento *Windows Sockets 2 Protocol-Specific Annex* contiene información específica de un número de protocolos de transporte que son accesibles vía Windows Sockets 2. En principio este documento no nos interesa al igual que el de SPI.

9.4. Nuevos conceptos, adiciones y cambios

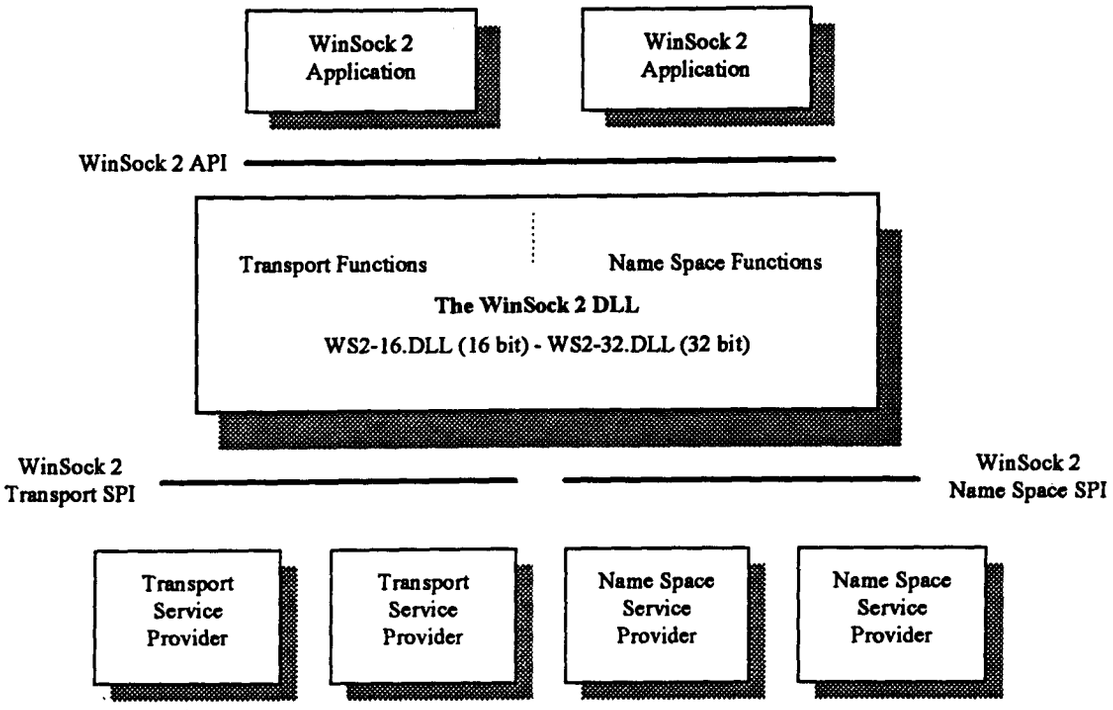
A partir de aquí se intenta resumir los principales cambios y adiciones que están por venir en Winsock ver 2. Para adquirir una información detallada sobre cómo utilizar una determinada función, se recomienda siempre acudir a la descripción provisional de la API.

9.4.1. Acceso simultáneo a múltiples protocolos de transporte

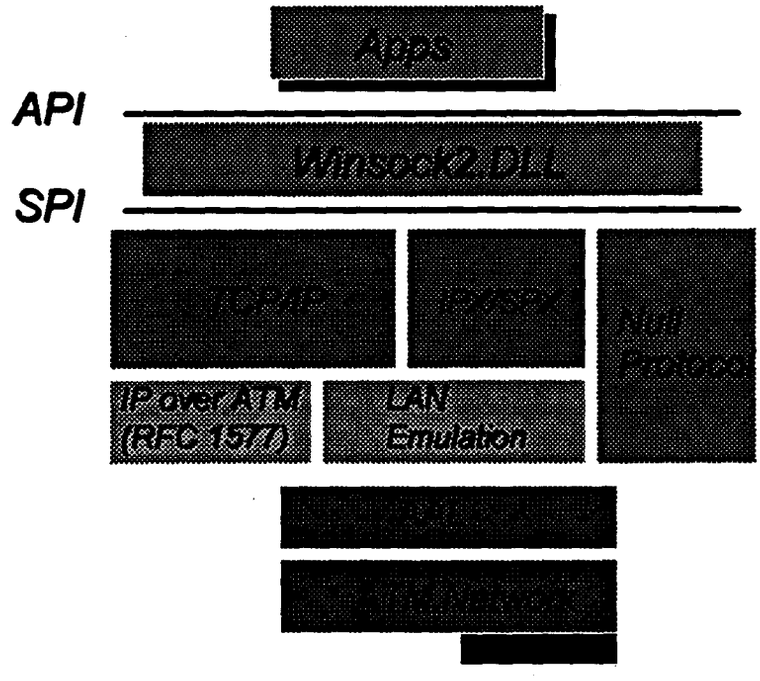
Winsock 2 proporciona un acceso simultáneo a múltiples protocolos de transporte lo que provoca que la arquitectura de Windows Sockets 2 se vea modificada substancialmente. Con Windows Sockets 1.1, la DLL que implementaba la interfase, era suministrada por el vendedor de la pila de protocolos TCP/IP. La interfase, por tanto, entre Windows Sockets y la pila de protocolos era única y característica del propietario.

Winsock 2 cambia este modelo definiendo una interfase estándar entre la DLL y las pilas de protocolos existentes. Gracias a esto es posible que puedan ser accedidas de forma simultánea múltiples pilas de protocolos de diferentes vendedores desde una única DLL. Es más, el soporte que proporciona Windows Sockets ahora no se verá limitado a TCP/IP como lo era en el caso de la versión 1.1. sino que podrá abarcar cualquier pila de protocolos que se adapte a las especificaciones de la SPI.

Por otro lado, no sólo existirán proveedores del servicio de transporte como pueden ser TCP/IP, DecNet, IPX/SPX, etc.. sino que aparecerán también proveedores del servicio de dominio de nombres, en los cuales debemos citar como más importantes el DNS, X.500, SAP, etc.. Así, la arquitectura de Winsock 2 queda como se muestra en la siguiente figura:



Con la arquitectura arriba mostrada se puede observar que no es necesario que cada vendedor suministre su propia DLL, dado que una única DLL deberá trabajar a través de todas las pilas de protocolos instaladas. Así, Winsock.DLL deberá ser vista de la misma forma que un componente del sistema operativo. En el siguiente gráfico podemos ver una posible configuración sobre una red ATM.



Desde el punto de vista de desarrollo, Microsoft se ha comprometido a crear la librería de enlace dinámico Winsock2 tanto para Windows 95 como para Windows NT, que será disponible gratuitamente. Por otro lado, la corporación Intel ha expresado su buena voluntad para producir una Winsock2.DLL disponible bajo los entornos Windows 3.1 y Windows 3.11 debido a la evidente demanda que existe todavía bajo estos sistemas operativos.

Resumiendo, el concepto más importante que debemos tener claro es la aparición de los denominados proveedores del servicio, que serán las pilas de protocolos instaladas en la máquina. La necesidad de una interfase que estandarice el acceso entre estas pilas de protocolos y Winsock2.DLL, llamada SPI, y otra interfase que estandarice el acceso de una aplicación a Winsock2.DLL, llamada API.

Con este esquema se ha conseguido un mayor nivel de abstracción en Winsock 2 y se permite tener varios protocolos de transporte disponibles a través de una única DLL.

Por otro lado, dado el gran número de protocolos de transporte y las familias de direcciones que soportará cada uno, será necesario establecer también una serie de proveedores del servicio de domino de nombres para poder localizar una dirección dentro de un dominio de nombres específico. Si bien en Winsock 1.1 sólo nos enfrentábamos al DNS por el que se regía TCP/IP ahora debemos tener en cuenta otros tales como SAP, X.500, etc.. por los que se rigen otros protocolos diferentes a TCP/IP. Así, la API deberá proporcionar un conjunto de funciones al programador lo más generales posibles que permitan resolver nombres en direcciones independientemente del proveedor de servicio de dominio de nombres utilizado.

9.4.2. Compatibilidades con aplicaciones Winsock 1.1

Cuando se crea una nueva versión de cualquier software siempre aparece la duda de la compatibilidad con las versiones anteriores. Esto plantea problemas tales como heredar malas conductas de versiones antiguas para poder hacerlas compatibles con las nuevas versiones. Windows Sockets en este respecto, amplia notablemente la API con nuevas funciones pero no elimina las antiguas por razones de compatibilidad. Esto no provoca que

se hereden malas conductas, ya que los nuevos programas actuarán con las nuevas funciones de la API y actuarán eficazmente, mientras que los programas diseñados bajo la versión 1.1. seguirán funcionando igual que antes; el único problema será el tamaño de la DLL que se podría reducir eliminando las funciones obsoletas de la versión 1.1 pero esto tampoco es una gran problema. La modificación en la arquitectura de Windows Sockets ver. 2 es evidente que provoca grandes cambios. Cambios que serán invisibles de cara a las aplicaciones creadas con Windows Sockets ver 1.1.

La versión 2 de la DLL será totalmente compatible con la versión 1.1 tanto a nivel de código fuente como a nivel binario, suponiendo, claro está, que exista al menos una pila de protocolos TCP/IP correctamente instalada y registrada vía Winsock 2.

9.4.2.1. Compatibilidad del código fuente

Esto significa que todas las funciones de la API de Winsock ver. 1.1 se mantienen en la versión 2 como ya se comentó anteriormente. Así que el código fuente de una aplicación Winsock ver. 1.1 puede ser fácilmente transportado al sistema Winsock 2 sin más que añadir o incluir el fichero de cabecera de Winsock2 y "*linkar*" de nuevo la aplicación con las nuevas librerías.

Los diseñadores de software deben entender esta compatibilidad como el primer paso en una transición completa hacia Winsock 2.

9.4.3. Protocolos de Trasnporte Disponibles

Para que un protocolo de transporte sea accesible vía Winsock éste debe estar debidamente instalado en el sistema y registrado con Winsock 2. La versión 2 de Winsock DLL exportará una serie de API's[#] las cuales llevarán a cabo el proceso de registración. Estas incluyen la creación de una nueva registración o la eliminación de una ya existente.

[#] el término API's no es más que una forma de hablar ampliamente utilizada aunque no muy correcta. La API es un conjunto de funciones y cuando decimos API's nos referimos a una serie de funciones de esta API y no a un conjunto de API diferentes.

Cuando se registra un protocolo de transporte, el que invoca estas funciones (presumiblemente el fichero *script* de instalación de la pila de protocolos) suministra una o más estructuras PROTOCOL_INFO[##] conteniendo una información detallada respecto al protocolo. Esta estructura será la utilizada por la aplicación para conocer las características de la pila de protocolos respectiva.

Para obtener más detalles de cómo es llevada a cabo esta operación se recomienda acudir al documento referente a la SPI (Service Provider Interface).

9.4.4. Utilización de varios protocolos diferentes

Desde el punto de vista de una aplicación, para averiguar a qué protocolos de transporte puede acceder debe llamar a la función *WSAEnumProtocols()*, obteniendo además las estructuras PROTOCOL_INFO con las que se registrarón y en las cuales existe información sobre cada protocolo.

La idea de que se obtendrá una estructura por cada protocolo de transporte disponible es incompleta. Puede ocurrir, aunque no es lo normal, que un mismo protocolo de transporte tenga múltiples conductas o comportamientos diferentes por lo que para cada comportamiento dispondremos de una estructura de este tipo.

En la versión 1.1 de Winsock existía una única familia de direcciones (AF_INET) en la cual se incluían una serie de tipos de sockets bien conocidos e identificadores de protocolos. Con Winsock 2 este concepto sufre un gran avance debido a la posibilidad de disponer de un gran número de familias de direcciones, tipos de sockets, y de protocolos diferentes.

Por motivos de compatibilidad entre ambas versiones se han mantenido los identificadores de la versión 1.1 sobre AF_INET, pero se espera que aparezcan muchas

[##] es una estructura destinada a guardar las características de los protocolos instalados en la máquina y registrados con Winsock 2. Por ejemplo, la función WSAEnumProtocols() devuelve una estructura de este tipo por cada protocolo instalado para proporcionar a la aplicación un método de averiguar las caracterísitcas de los protocolos.

nuevas familias de direcciones y protocolos, que si bien tendrán identificadores únicos no serán bien conocidos.

Si en Winsock 1.1 existía una estructura WSDATA que nos informaba de la pila de protocolos TCP/IP instalada, ahora este modelo ya no sirve y queda obsoleto dado que deberíamos tener varias estructuras WSDATA y con campos más generales, independientes del protocolo. Debido a esto nace la estructura PROTOCOL_INFO antes nombrada.

Por tanto, el concepto aquí va más allá del que se vio en la versión 1.1. Si en la versión 1.1. cuando el diseñador creaba un programa éste analizaba primero sobre el papel qué protocolos de los disponibles le venía mejor (si TCP o UDP) ahora no se tendrá que molestar en saber que TCP/IP implementa TCP y UDP, que uno es orientado a conexión y el otro no, etc.. sino que en su aplicación enumerará los protocolos de transporte disponibles en su máquina y según los atributos de la comunicación que desee (ej. orientado a conexión o no, seguro o no seguro, etc..) buscará entre las estructuras PROTOCOL_INFO aquel protocolo de los disponibles que le venga mejor.

Así, la selección de los protocolos según los atributos o conducta que desee la aplicación en la comunicación frente a la elección dentro de un número de protocolos bien conocidos hará más potente a la versión 2 en el sentido de que podrá aprovechar las ventajas de la implantación de nuevos protocolos de transporte y sus conductas según vayan apareciendo.

Si antes decíamos que para una comunicación segura y orientada a conexión debemos elegir TCP y para una no orientada a conexión y no segura UDP, ahora preguntaremos dentro de nuestra aplicación por una serie de atributos de la comunicación de forma general, esto nos reportará uno o varios protocolos de transporte disponibles que cumplen esos requisitos. Es decir, no debemos saber nada de los protocolos de transporte, Winsock nos reportará las conductas de cada uno y la que mejor nos venga en ese caso será la que utilizaremos.

9.4.5. Resolución de Nombres independientemente del protocolo

Como ya se comentó, la aparición de nuevos protocolos provocará la aparición de nuevos dominios de nombres, y de nuevos métodos para resolver un nombre dentro de un dominio en su dirección correspondiente. Así, para cada tipo de dominio de nombres tendríamos una serie de funciones diferentes y características que nos resolverían el problema. Pero Winsock 2 vuelve a ir más allá en su intento por suministrar una interfase de comunicación transparente e intenta estandarizar estas funciones independientemente del dominio en el que nos encontremos. Para ello deja recaer todo el peso sobre el Proveedor del Dominio de Nombres, que será el que resuelva de forma particular el problema; pero a nivel de aplicación serán funciones estándar de la API.

9.4.6. Entradas/Salidas Asíncronas y Objetos Evento

Winsock 2 introduce el concepto de I/O superpuestas "*overlapped*" y requiere que todos los proveedores de transporte soporten esta conducta. El modelo seguido para realizar operaciones asíncronas en sockets es idéntico al utilizado por Win32. Para profundizar en lo que son los objetos eventos recomiendo la lectura de cualquier bibliografía sobre la programación en Windows 95 y Windows NT.

9.4.7. Introducción del concepto de Calidad del Servicio QOS

Los mecanismos básicos de QOS (Quality Of Service) en Winsock 2 provienen de la especificación de flujo descrita por Craig Partridge en la Requets For Comment número 1363 (RFC_1363) que data de Septiembre de 1992.

La especificación de flujo describe un conjunto de características sobre un flujo de datos unidireccional propuesto a través de una red. Una aplicación debe asociar un par de especificaciones de flujo por cada socket (una especificación de flujo para cada sentido) en el momento del establecimiento de la conexión utilizando *WSAConnect*(). Esto es un mecanismo que proporciona una indicación sobre el tipo de calidad del servicio que se requiere y proporciona un mecanismo de realimentación para las aplicaciones de forma que van adaptando el uso de la red según las capacidades de ésta.

Ahora hay que distinguir en los métodos en que se establecerá la QOS dependiendo si la comunicación es orientada a conexión o es en modo datagrama.

Si se trata de una comunicación orientada a conexión es muy normal que la aplicación establezca la QOS en el momento del establecimiento de la conexión con su pareja. Sin embargo también es posible establecer una QOS antes de la conexión o simplemente variarla una vez conectados utilizando la función *WSAIoctl*(). Si nos conectamos haciendo uso de *WSAConnect*() e indicamos una nueva QOS, en caso de haber especificado otra QOS anteriormente mediante *WSAIoctl*() la especificada anteriormente mediante *WSAIoctl*() no tendrá efecto. Para que no se vea afectada y se tenga en cuenta la QOS especificada con *WSAIoctl*() se deberá anular los parámetros asociados con la QOS en la función *WSAConnect*().

Si la función *WSAConnect*() da un resultado correcto, es decir que no reporta errores, es que tanto la red como la pareja a la que nos conectamos han aceptado nuestra QOS. Si por el contrario da un error, es que la red no admite esa QOS requerida por nuestra aplicación y debemos bajar la calidad y volver a renegociar la QOS o bien terminar la aplicación si se considera que es inaceptable la nueva QOS para llevar a cabo el servicio.

Como hemos visto, la QOS establece una mecanismo de negociación entre las posibilidades de la red y los requerimientos de la aplicación.

Después de cualquier intento de conexión, los proveedores de las capas de transporte actualizan las estructuras de la especificación de flujo asociada para indicar, en la medida de lo posible, las condiciones existentes de la red. Si los proveedores actualizan esas estructuras con los valores por defecto mostrados en la especificación nos están indicando que no disponen de ninguna información sobre las condiciones actuales de la red. Es necesario advertir que los valores retornados por el proveedor sobre la QOS de la red no es del todo fiable dado que no nos suministra una idea extremo a extremo y tan sólo es válida a nivel local. Además la QOS suministrada es una mera aproximación y no es tan real como la que devuelve la red cuando se establece la conexión. Digo esto porque es un punto que deben tener muy en cuenta las aplicaciones. En un modelo no orientado a conexión es

posible utilizar la función *WSAConnect*() tal y como se discutió en el capítulo anterior; es decir, un *connect*() en modo datagrama tan sólo fija una dirección destino por defecto, a donde serán enviados los datagramas en caso de no indicar la dirección destino en ellos. No enviará ningún paquete a la red para confirmar la conexión con el otro extremo.

9.4.7.1. Renegociaciones de la QOS establecida la conexión

Después de haber establecido la conexión, dado el carácter variable de una red, sus condiciones pueden variar. Además es posible que uno de los extremos desee cambiar la QOS. En estos casos es necesario diseñar un mecanismo de notificación, que alerte a la aplicación sobre el cambio de la QOS por parte del otro extremo o que las condiciones de la red han variado y por tanto es necesario una renegociación de la QOS.

Si las características de la red han variado nuestra aplicación recibirá un mensaje FD_QOS ó FD_GROUP_QOS (según si afecta a un socket o un grupo de sockets). Entonces nuestra aplicación deberá analizar qué parámetros han cambiado en la QOS y si son aceptables. Para ello se hace uso una vez más de la función *WSAIoctl*() pasándole como parámetro SIO_GET_QOS ó SIO_GROUP_QOS.

9.4.7.2. Características de la QOS implementada en Winsock 2

La especificación de flujo propuesta para Winsock 2 divide las características de la QOS en una serie de áreas generales tales como:

✓ Descripción de la fuente de tráfico.

Es la forma en la que el tráfico generado por nuestra aplicación será inyectado en la red. Aquí se detallarán los valores de velocidad del flujo de datos (token rate), el tamaño máximo del buffer (token size) y el pico de velocidad que puede ser alcanzado en una ráfaga (peek bandwidth).

✓ Latencia.

Indica o fija los límites superiores de la cantidad de retardo que es aceptable y su variación. Es decir, podremos especificar que un retardo de menos de x msg es aceptable y que ese retardo no varíe de un momento a otro en no más de y msg.

✓ Nivel de garantía del servicio.

Aquí indicamos el nivel de garantía que exijimos a la hora de establecer la QOS fijada. De nada sirve dar unos datos de QOS muy buenos si después no se cumplen. Con el nivel de garantía del servicio especificamos hasta que punto deben cumplirse nuestras exijencias en la QOS especificada.

✓ Coste.

Este campo actualmente no se utiliza y sólo se ha tenido en cuenta para un futuro cuando se haya establecido una métrica significativa sobre el coste de cada enlace.

✓ Parámetros específicos del proveedor.

La especificación de flujo puede ser extendida en otros aspectos particulares de cada proveedor. En este campo se contemplan todos esos detalles particulares de cada proveedor.

9.4.7.3. Estructuras asociadas a la QOS en Windows Sockets

Es hora de ver un poco más en detalle las estructuras utilizadas por Windows Sockets para especificar la QOS según el modelo descrito antes.

La estrucutra más general es la siguiente en la que se especifica una QOS por cada sentido de la comunicación y además se encuentra el campo donde se especifican las particularidades de la QOS implementadas por cierto proveedor.

```
typedef struct _QualityOfService
{
        FLOWSPEC  SendingFlowspec;    /* QOS en el sentido de la transmisión */
        FLOWSPEC  ReceivingFlowspec;  /* QOS en el sentido de la recepción */
```

```
        WSABUF      ProviderSpecific;      /* Buffer donde se almacenan los
parámetros */
                                           /* específicos de cada proveedor. */
    } QOS, FAR *LPQOS;
```

Veamos ahora cómo es el tipo de dato FLOWSPEC.

```
    typedef struct _flowspec
    {
        int32   TokenRate;          /* Expresado en bytes/sg  */
        int32   TokenBucketSize;    /* Expresado en bytes.  */
        int32   PeakBandwidth;      /* Expresado en bytes/sg */
        int32   Latency;            /* Expresada en microsegundos */
        int32   DelayVariation;     /* En microsegundos también  */
        GUARANTEED LevelOfGuarantee; /* Nivel de garantía */
        int32   CostOfCall;         /* Valor reservado para el futuro. Poner a 0
*/
        int32   NetworkAvailability; /* Disponibilidad de la red: 1 si es accesible
*/
                                    /*                          0 si no es accesible
*/
    } FLOWSPEC, FAR *LPFLOWSPEC;
```

Dentro del campo LevelOfGuaranty los valores que se aceptan son:

```
    typedef enum
    {
        BestEffortService,
        PredictiveService,
        GuaranteedService
    } GUARANTEE;
```

9.4.8. Grupos de Sockets

Winsock 2 introduce la noción de grupo de sockets como una forma para una aplicación de indicar a los proveedores de servicio que un conjunto particular de sockets están relacionados y que este grupo así formado tienen ciertas propiedades o atributos comunes. Los atributos de un grupo incluyen tanto prioridades de los sockets entre sí dentro del grupo y de la calidad de servicio de éstos.

Por ejemplo, las aplicaciones que intercambien flujos de datos multimedia a través de una red se ven beneficiadas dado que se permitirá establecer una relación entre todos los sockets utilizados. Como mínimo esto deberá dar una pista al proveedor del servicio sobre las prioridades relativas de cada flujo de datos que deberá transportar. Por ejemplo, una aplicación de videoconferencia deseará que el socket utilizado para la transmisión del audio tenga mayor prioridad que el utilizado para el vídeo. Es más, existen proveedores de servicio (ej: telefonía digital y ATM) que pueden utilizar la especificación de la calidad de servicio de un grupo para determinar las características apropiadas para la llamada subyacente o el circuito de conexión. Los sockets dentro de un grupo son entonces multiplexados de la forma usual a través de esa llamada. Permitir que la aplicación identifique los sockets que componen un grupo y que se especifique sus atributos hace que tales proveedores de servicio puedan operar de una forma más eficiente.

9.4.9. Compartir sockets

Esta nueva posibilidad introducida en Winsock 2 permite que un mismo socket pueda ser utilizado a través de varias tareas. La función *WSADuplicateSocket*() es introducida en la especificación para este fin. Esta función tiene como entradas el descriptor del socket local y un manejador de la tarea destino, sobre la que se creará un nuevo descriptor que hará referencia al mismo socket local. Esta función retorna un nuevo descriptor para el socket que es únicamente válido en el contexto de la tarea destino.

9.5. Comentario final sobre Windows Sockets 2.0.

Esta nueva versión de Windows Sockets pretende conseguir obtener una interfaz de programación en red donde se abstraen los detalles de las capas inferiores lo máximo posible. Además esta nueva especificación intenta conseguir que el programador tenga un acceso mediante una única DLL a varias capas de transporte diferentes, para lo cual ha tenido que variar su arquitectura. Uno de los grandes pasos ha sido este dentro de la nueva especificación.

Para finalizar hacer incapié en el interés de muchas empresas en que Windows Sockets se convierta en un estandar para la programación en red bajo Windows. El salto que se ha dado de la versión 1.1. a la 2.0. es todo un reto por parte de estas empresas.

En un futuro no muy lejano la mayoría de las aplicaciones que deseen interactuar con la red lo harán por medio de la interfaz Windows Sockets.

Bibliografía

🕮 Andrew S. Tanenbaum, "Redes de Ordenadores", De. Prentice Hall

🕮 Uyless Black, "Redes de Ordenadores, Protocolos, normas e interfaces", De. Prentice Hall

🕮 Douglas E. Corner - David L. Stevens, "Internetworking with TCP/IP, VOLUME III", Ed. Prentice Hall

🕮 Jaime Peña Tresancos, "Fundamentos y desarrollo de programas en Windows 3.X", Ed. Anaya Multimedia

🕮 Helbert Schildt, "Aplique Turbo C++ para Windows", Ed. McGraw Hill

🕮 Ángel Franco García, "Programación de aplicaciones Windows con Borland C++ y ObjectWindows", Ed. McGraw Hill

🕮 David D. Clark, RFC 814 "Name, Adresses, Ports and Routes", Mit Laboratory of Computer Science - Computer Systems and Communications Group, Julio 1982

🕮 J. Postel, RFC 1591 "Domain Name System Structure and Delegation", Marzo 1994

🕮 Jean-Marie Rifflet, "Comunicaciones en UNIX", Ed. McGraw Hill, 1992

🕮 Bill Rieken - Lyle Weiman, "Adventures in UNIX Network Applications Programming", Ed. John Wiley & Sons, New York 1992

🕮 Lee Atkinson - Mark Atkinson, "Programación en Borland C++ 3.X", Ed. Anaya Multimedia, 1993

🕮 Reg Quinton (reggers@julian.uwo.ca), "The Domain Name Service", Computing and Comunications Services, The University of Western Ontario

🕮 Martin Hall- Mark Towfig - Geoff Arnold - David Treadwell - Henry Sanders, "Windows Sockets; An Open Interface for Network Programing under Microsoft Windows" Version 1.1.,

🕮 Martin Hall (martin@jsbus.com), "A guide to Windows Sockets", 1 Junio de 1993

🕮 Reg Quinton, "An Introduction to Socket Programming", Computing and Comunicatios Services, The University of Western Ontario

🕮 Peter Norton - Paul Yao, "Guia Peter Norton: Windows 3.1; Técnicas de Programación", Ed. Ariel S.A.

📖 Rafael García Oliva, "Programa de Simulación de los niveles bajos de una arquitectura OSI bajo el entorno Microsoft Windows 3.1.", Proyecto Fin de Carrera de la especialidad de Transmisión y Datos de la Escuela Universitaria de Ingenieros Técnicos de Telecomunicación de Las Palmas de G.C., Mayo 1994

📖 Namir Clement Shammas, "Libreria ObjectWindows", Ed. Anaya Multimedia, 1993

📖 Uday O. Pabrai, "UNIX Internetworking", Ed. Artech House, Boston 1993