



UNIVERSIDAD
de SEVILLA

ARQUITECTURA DE REDES DE COMPUTADORES II

2ª Práctica. Sockets.

**4º Curso de la Ingeniería en Informática
Departamento de Tecnología Electrónica
Universidad de Sevilla**

Versión 1.0, 26 de marzo de 2.000

TITULACIÓN: **INGENIERO EN INFORMÁTICA**
ASIGNATURA: **ARQUITECTURA DE REDES DE COMPUTADORES II**
(4º Curso)
DOCUMENTO: **Boletín de la práctica 2. Sockets.**
REFERENCIA: **BoletinPractica2.v10.wpd**
FECHA: **26 de marzo de 2.000**

HOJA DE CONTROL DE REVISIONES

Versión/Revisión	Contenido de las actualizaciones
1.0	Primera versión del presente documento (curso 2.000-2.001)

Actuación	Nombre	Firma	Fecha
Elaborado por:	Sergio Díaz Ruiz		26/03/2000

ÍNDICE

1.- INTRODUCCIÓN	5
2.- OBJETIVOS	5
3.- SOCKETS	5
3.1.- Identificación de un socket	6
3.2.- Tipos de sockets	7
4.- MODELO DE COMUNICACIÓN CLIENTE-SERVIDOR	8
4.1.- Proceso de conexión cliente-servidor	8
5.- MODELO DE PROGRAMACIÓN	9
5.1.- Programación de sockets POSIX	11
5.1.1.- <i>Byte order</i>	12
5.1.2.- Tipos de datos para la manipulación de sockets TCP en UNIX	12
5.1.3.- Funciones para la manipulación de sockets TCP en UNIX	13
5.2.- Programación de sockets en Windows	15
5.2.1.- Programación de sockets en Windows usando C++ y MFC	16
5.3.- Programación de sockets mediante Java	17
6.- DESARROLLO DE LA PRÁCTICA	17
6.1.- Nociones básicas sobre servidores web.	17
6.1.1.- Especificando recursos en HTTP	18
6.1.2.- HTML	19
6.1.3.- Contenido dinámico	19
6.1.4.- Seguridad	21
6.1.5.- Ejemplos de servidores web	21
6.2.- Enunciado	21
6.3.- Descripción del subconjunto HTTP a implementar	22
6.3.1.- Solicitudes HTTP	22
6.3.2.- Respuestas HTTP	23
6.3.3.- Ejemplo	25
7.- CRITERIOS DE VALORACIÓN Y PRESENTACIÓN	26
7.1.- Criterios generales	26
7.2.- Criterios específicos de la segunda práctica	27
7.3.- La defensa de la práctica	27

1.- INTRODUCCIÓN

Los *sockets* son interfaces lógicas de entrada/salida que permiten la comunicación entre procesos que pueden residir en máquinas distintas pero que estén conectadas mediante una red. Es el mecanismo utilizado por las aplicaciones en red basadas en tecnología Internet, de ahí su importancia dada la difusión de ésta en la actualidad. Los sockets se describen en el apartado 3.

Las aplicaciones en red suelen responder a un modelo conocido como *cliente-servidor*, que se describe en el apartado 4. En este modelo, el *servidor* es un proceso encargado de la gestión de un determinado recurso de una máquina. El *cliente*, que, en general, se ejecuta en una máquina diferente, interacciona con el recurso *remoto* estableciendo una comunicación con el servidor, siguiendo las reglas impuestas por el protocolo de nivel de aplicación que corresponda.

Tras esta introducción a las comunicaciones en redes tipo Internet, pasamos a describir los interfaces de programación que permiten el acceso a los sockets en distintos lenguajes y sistemas operativos, apartado 5.

Por último, el apartado 6 desarrolla el enunciado de la práctica, mientras que en el apartado 7 se recogen los criterios de evaluación de la misma. Básicamente, la práctica consiste en implementar un servidor *web* sencillo usando sockets.

2.- OBJETIVOS

- Proporcionar una introducción a los conceptos básicos de programación de aplicaciones en red basadas tecnología Internet (TCP/IP):
 - Sockets y sus interfaces de programación en diferentes entornos
 - Modelo cliente-servidor
- Introducir los conceptos fundamentales relativos al protocolo HTTP, servidores web y navegadores
- Llevar a la práctica estos conocimientos implementando un servidor web sencillo capaz de interactuar con un navegador web convencional.

3.- SOCKETS

Los sockets son interfaces lógicas que permiten la comunicación bidireccional entre procesos que se ejecutan en una misma máquina o en máquinas distintas conectadas en red. Por ejemplo, los navegadores web se comunican con los servidores web mediante sockets. Es frecuente encontrar en la bibliografía analogías entre sockets y teléfonos o buzones de correo. En cierto modo, estos últimos permiten la comunicación entre personas de forma similar a como los sockets hacen posible la interacción entre procesos.

La comunicación mediante sockets es el mecanismo usado por las aplicaciones en red que se basan en tecnología Internet, es decir, en la familia de protocolos TCP/IP. De hecho, un socket no es más que la implementación del acceso a los servicios del nivel de transporte de la familia TCP/IP. En las aplicaciones en red es importante distinguir los recursos *locales* de los recursos *remotos*. Siguiendo con

el ejemplo anterior, el navegador web prepara un socket *local* (es decir, en su máquina) para comunicarse con el socket *remoto* dispuesto por el servidor web. Por extensión, suele hablarse también de proceso local/remoto o incluso de máquina local/remota.

3.1.- Identificación de un socket

Desde el punto de vista de un proceso, un socket no es más que un recurso -muy parecido a un fichero- que se solicita localmente al sistema operativo y que, de alguna forma, “abre una puerta hacia el exterior”. Es fundamental que este socket esté identificado convenientemente para que pueda ser referenciado desde el exterior. La forma de identificar el socket está íntimamente relacionada con el mecanismo de direccionamiento de la familia de protocolos subyacente, es decir, la familia TCP/IP (Internet), y se describe a continuación.

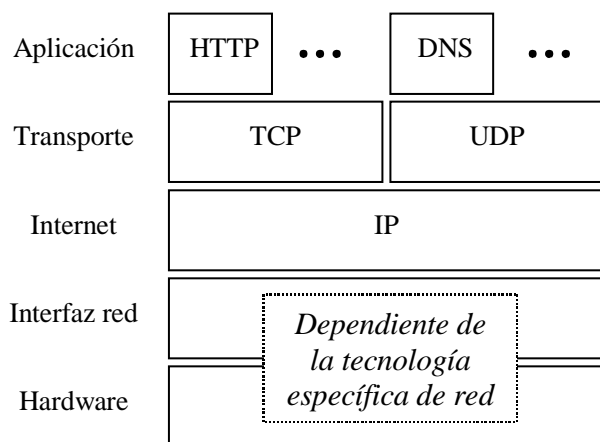


Figura 1. Modelo TCP/IP

En la figura 1 se muestra la pila de protocolos de la familia TCP/IP, que puede asimilarse al modelo de referencia OSI si consideramos vacíos los niveles de sesión y presentación. Sobre una capa dependiente de la tecnología de red usada, encontramos el protocolo de nivel de red *IP* (*Internet Protocol*), responsable básicamente del direccionamiento de las estaciones y del enrutamiento de paquetes. Para nuestros propósitos, lo importante aquí es que, en la red, **una máquina se distingue unívocamente de las demás mediante su dirección de nivel de red, llamada dirección IP**. Sobre el nivel de red,

cubriendo funciones propias del nivel de transporte, encontramos dos protocolos: *TCP* (*Transmission Control Protocol*) y *UDP* (*User Datagram Protocol*). TCP garantiza el transporte fiable de los datos, mientras que UDP no, pero presenta la ventaja de ser mucho más sencillo. Para nosotros, lo importante de estos protocolos de transporte es que incorporan un mecanismo de direccionamiento que permite diferenciar los sockets de una cierta máquina. Es decir, **un socket que reside en una cierta máquina se distingue unívocamente de los demás sockets de la misma máquina mediante su dirección de nivel de transporte, habitualmente denominada *puerto***. Así, hablaremos de “*puertos TCP*” o “*puertos UDP*” dependiendo del protocolo de transporte que corresponda. En definitiva, el puerto nos permite distinguir los procesos relacionados con la red dentro de una misma máquina. Como consecuencia de todo lo anterior, un socket se identifica indicando la máquina en la que reside y el puerto al que está asociado, es decir:

socket: (dirección IP, puerto)

Las *direcciones IP* están formadas por 32 bits, es decir 4 bytes. Habitualmente se representan escribiendo el valor en decimal de cada byte (en el rango 0..255, por tanto), y separando los bytes con un punto, ‘.’. Por ejemplo, *150.214.141.89* es la dirección IP del servidor web del Departamento de Tecnología Electrónica. Estas direcciones son asignadas por el administrador de la red.

Existen direcciones IP especiales, como, por ejemplo, la dirección *127.0.0.1*, cuya utilidad se explica a continuación. Toda máquina con capacidad de comunicaciones TCP/IP responde, al menos, a la dirección *127.0.0.1*, aunque ni siquiera esté conectada a la red. Esto nos permite establecer

Sockets

comunicación entre procesos locales mediante sockets sin necesidad de que el equipo posea una dirección IP “real” (o bien, aunque la posea, no necesitamos conocerla para establecer comunicaciones locales). La dirección 127.0.0.1 nos permite comprobar el buen funcionamiento de la pila de protocolos.

Dado que la identificación de los equipos por su dirección IP es poco amigable, se desarrolló el servicio de nombres de dominio (DNS, *Domain Name Service*) que asocia la dirección IP de un host con un nombre de la forma *host.dominio*. Gracias a este sistema, podemos especificar en nuestro navegador la dirección *www.dte.us.es* (host = *www*, dominio = *dte.us.es*) para acceder al servidor web del Departamento de Tecnología Electrónica, en lugar de introducir 150.214.141.89, por seguir con un ejemplo anterior. Análogamente, la dirección 127.0.0.1 equivale al nombre *localhost*.

Los *puertos* son valores de 16 bits, y por tanto cubren el rango 0..65535. Algunos de estos puertos están asignados para las aplicaciones estándar de Internet¹. Por ejemplo, el puerto 80 es el puerto estándar en el que un servidor web recibe las solicitudes del navegador. El resto de los puertos pueden ser usados libremente por las aplicaciones, aunque normalmente los sistemas operativos se reservan un conjunto de puertos para uso interno. Siguiendo con un ejemplo anterior, el servidor web del Departamento de Tecnología Electrónica recibe peticiones desde los clientes a través del socket identificado por (dir. IP = 150.214.141.89, Puerto = 80).

3.2.- Tipos de sockets

Como hemos visto, los sockets están relacionados íntimamente con el protocolo de transporte. En la familia TCP/IP tenemos dos posibles protocolos de transporte, TCP y UDP, por lo que, en consecuencia, existen dos tipos de sockets²:

- sockets TCP, “*stream*” u “*orientados a la conexión*”. La comunicación entre sockets TCP es fiable (sin pérdida de paquetes), ordenada (TCP nos proporciona los paquetes en la secuencia correcta) y sin duplicación de paquetes. TCP, y consecuentemente, este tipo de sockets, es orientado a la conexión. Esto quiere decir que los sockets de los extremos a comunicar deben establecer un *circuito virtual* a través del cual se producirá el intercambio de información. El establecimiento del circuito virtual se suele denominar *conexión*, y es el paso previo al intercambio de datos. Una vez finalizado éste, se procede a la *desconexión*. En la analogía que señalábamos al principio del apartado 3, los sockets TCP se parecen más a los teléfonos, en los cuales la conexión se corresponde con la realización de una llamada. El destino es seleccionado gracias a su número de teléfono y avisado mediante los tonos apropiados. Cuando el destino descuelga, se establece el circuito virtual y es posible la comunicación bidireccional. Cuando ésta termina, los dos extremos cuelgan, cerrando el circuito.
- sockets UDP, “*datagram*” o “*no orientados a la conexión*”. La comunicación entre sockets UDP no garantiza la llegada de todos los paquetes, ni su orden, ni su unicidad. La aplicación que use

¹ Esta asignación de puertos puede encontrarse en el documento RFC1700. Las RFCs (*Request for Comment*) son documentos oficiales del IETF (*Internet Engineering Task Force*), que recogen los estándares, borradores de estándar y recomendaciones sobre todos los aspectos de Internet.

² A veces es posible usar, además de los tipos de sockets descritos, un tipo especial denominado “*raw*”, a través del cual se tiene acceso directamente al protocolo IP, dejando vacío el nivel de transporte. Los *sockets raw* quedan fuera del ámbito de esta práctica.

sockets de este tipo tendrá que encargarse de resolver estos problemas potenciales. Para la comunicación entre sockets UDP no se establece un circuito virtual, sino que en su lugar se realiza un intercambio de *datagramas*, es decir, de paquetes sueltos. Siguiendo con la analogía anterior, los sockets UDP se parecen más a los buzones de correo. El emisor sólo se encarga de introducir la información en el buzón. El servicio de correos, que representa a la red de comunicaciones, transporta la información hasta el destino, introduciéndola en su buzón. En ningún momento ha existido un contacto directo entre los extremos. De hecho, el emisor ni siquiera tiene la certeza de que el destino exista (en cuyo caso la información se pierde; en la comunicación entre sockets UDP no hay el equivalente postal a la devolución al remitente).

A partir de este punto, nos ocuparemos únicamente de los sockets TCP.

4.- MODELO DE COMUNICACIÓN CLIENTE-SERVIDOR

Las aplicaciones en red basadas en sockets TCP suelen presentar la arquitectura cliente-servidor. En este modelo, existen los procesos a comunicar pueden presentar los siguientes roles:

- **Servidor.** Se trata de un proceso que gestiona el acceso a algún tipo de recurso, por ejemplo, páginas web o una base de datos. El servidor es, por tanto, un proceso ininterrumpido, ya que sin él, el recurso no estaría disponible para las aplicaciones en red que lo necesiten.
- **Cliente.** Es un proceso que necesita acceder al recurso remoto gestionado por el servidor, de forma que tendrá que comunicarse con éste para llevar a cabo su tarea.

Hay que tener en cuenta que un mismo proceso puede presentar un rol de servidor respecto de otro proceso, pero al mismo tiempo también rol de cliente desde el punto de vista de un tercer proceso.

4.1.- Proceso de conexión cliente-servidor

El proceso de conexión entre cliente y servidor se ha representado en la figura 2, y se describe a continuación.

- (1) El proceso servidor, en su arranque, reserva un socket, *s*, denominado *socket servidor*, en su máquina local, y se queda en estado de espera, figura 2a.
- (2) El socket servidor (dirección IP y número de puerto) es conocido por el cliente, de forma que, cuando éste necesita acceder al recurso gestionado por el servidor, crea un socket, *c*, denominado *socket cliente*, y envía una solicitud de conexión al socket servidor, figura 2b.
- (3) El proceso servidor, que recibe la solicitud del cliente a través del socket servidor, crea un nuevo socket, *s'*, que se encargará de la comunicación con el cliente que ha solicitado la conexión, y lo conecta al socket cliente, figura 2c. El nuevo socket hereda la dirección del socket servidor (dirección IP y puerto)³. El socket servidor vuelve a su estado de escucha, lo que permite que se puedan aceptar nuevas solicitudes desde otros clientes de forma concurrente.

³ El socket que se ha creado en la máquina servidora para llevar a cabo la comunicación con el cliente tiene características similares al socket cliente, esto es, no está en estado de escucha, sino preparado para enviar y recibir información.

Normalmente, los servidores son *multihilo*⁴, para poder soportar esta forma de trabajo; así, cuando se crea un socket en el servidor para interactuar con un determinado cliente, se crea también un nuevo proceso servidor, que se encargará de atender las solicitudes de dicho cliente. Una vez establecida la comunicación, es posible el intercambio de información. Dado que son sockets de tipo TCP, se tiene garantía de que todos los datos llegarán, y además, en el orden correcto, y sin duplicados.

(4) Cuando el intercambio de información termina, el proceso cliente y el proceso que atiende a éste en el servidor cierran sus respectivos sockets, finalizando así la conexión. Por tanto, volvemos al estado inicial, figura 2d.

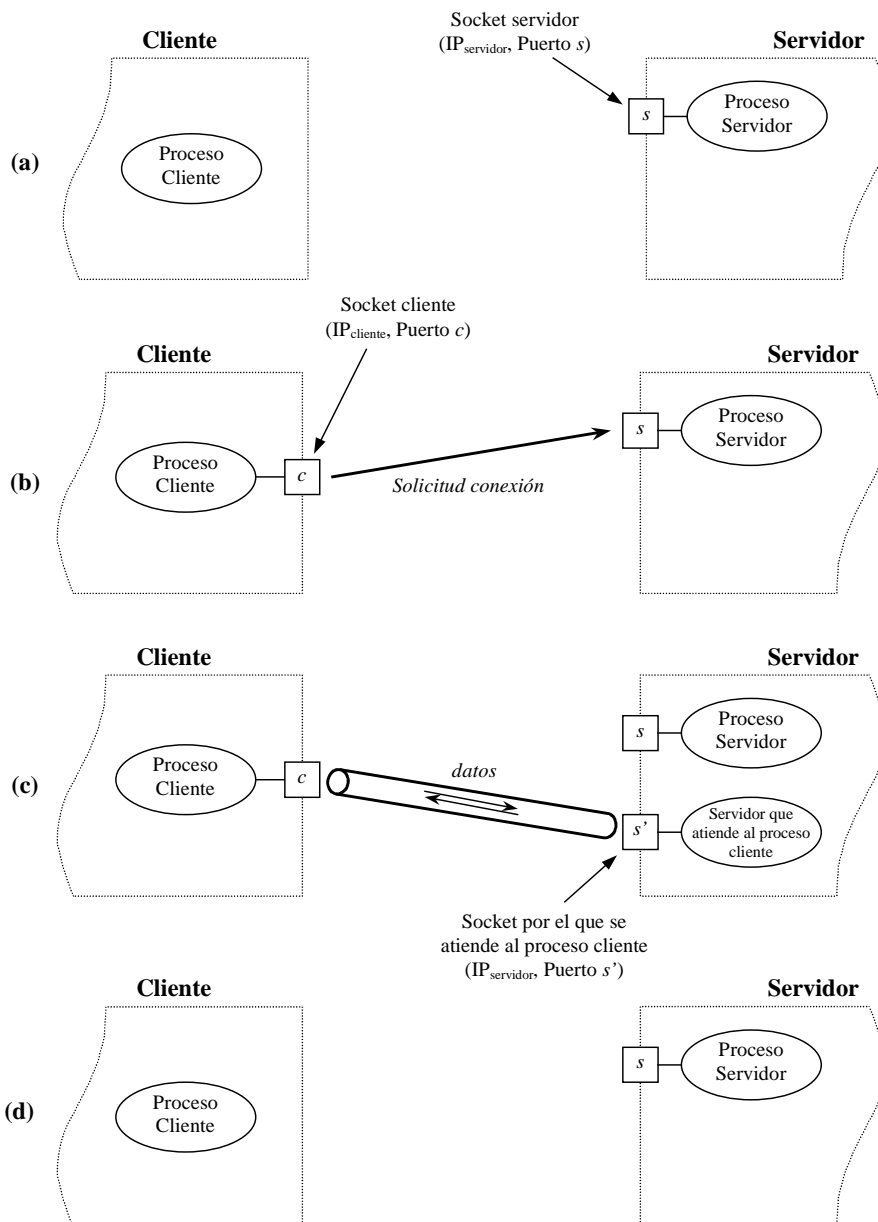


Figura 2. Proceso de conexión entre cliente y servidor.

⁴ Esto es, que tienen varios procesos ejecutándose concurrentemente en la máquina.

5.- MODELO DE PROGRAMACIÓN

El proceso de conexión descrito en el apartado anterior, desde el punto de vista del programador de procesos servidores o clientes, se corresponde con el diagrama mostrado en la figura 3.

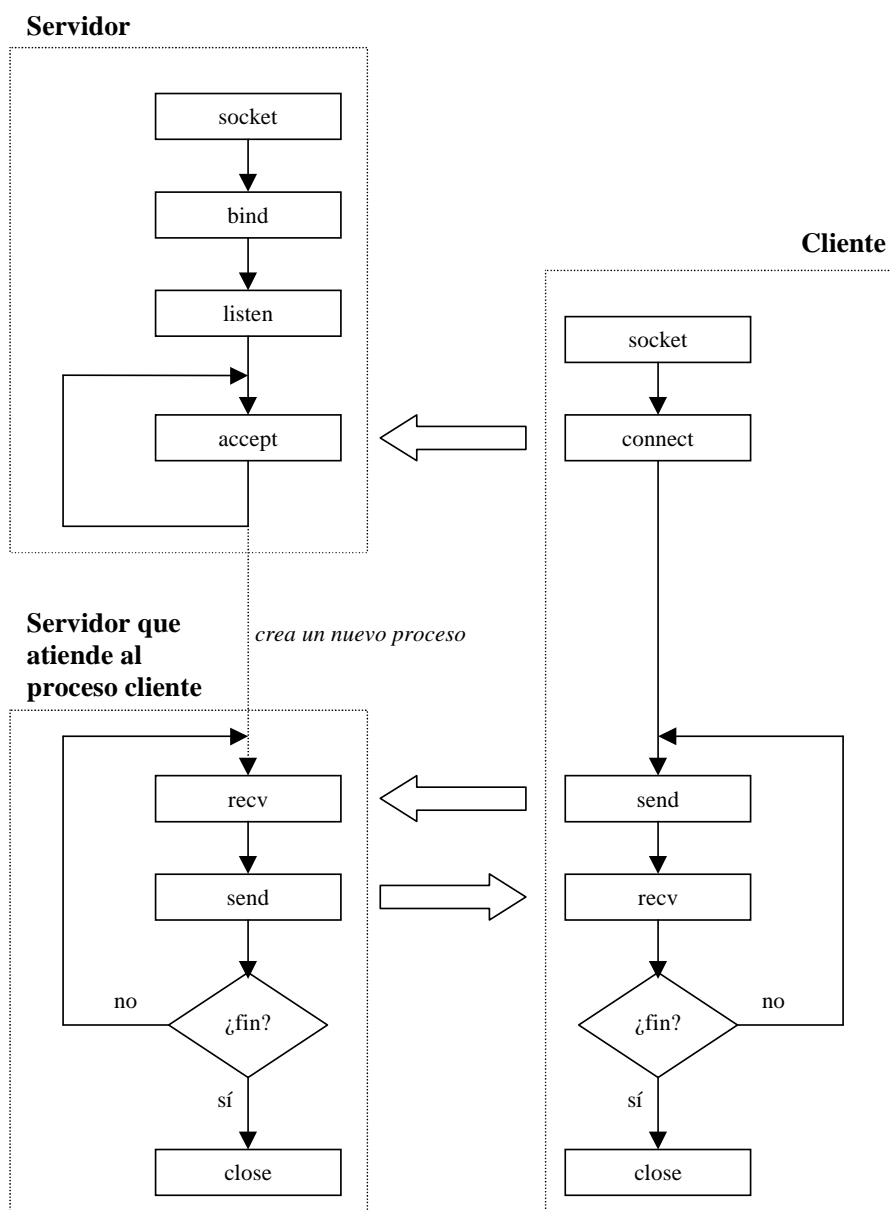


Figura 3. Proceso de conexión desde el punto de vista de programación.

Las acciones que aparecen en el diagrama se explican a continuación.

El proceso servidor crea (“abre”) el socket servidor(*socket*), le asigna un puerto (*bind*) en la máquina servidora y lo prepara para la escucha (*listen*), asignándole una cola de conexiones pendientes de

procesar. A partir de este momento, el socket pasa a estado de espera de conexión por parte del cliente (*accept*).

El cliente, por su parte, crea un socket de usuario (*socket*) y trata de conectarlo al socket servidor(*connect*), para establecer la comunicación.

Cuando el proceso servidor recibe tal solicitud, éste crea un proceso hijo para que se encargue de la comunicación con el cliente. También crea un socket (*accept*) y se lo asigna al nuevo proceso. El proceso servidor vuelve a su estado de espera de conexión (*accept*).

El proceso cliente y el proceso hijo del servidor intercambian información (*send, recv*). Normalmente, aunque no es obligatoriamente así, el cliente es el primero que envía información (tal como aparece en la figura 3). Cuando la comunicación termina, ambos “cierran” sus respectivos sockets (*close*).

Las acciones *socket*, *bind*, *listen*, *accept*, *connect*, *send*, *recv* y *close* se corresponden uno a uno con las funciones del API en lenguaje C del sistema UNIX para el manejo de sockets. Independientemente de la variante de UNIX que consideremos, el API es el mismo, ya que todas ellas siguen las especificaciones del estándar *POSIX*. Windows también sigue este enfoque y proporciona un API muy similar al de UNIX aunque con algunas modificaciones y extensiones.

El apartado 5.1 trata con cierto detalle el API de UNIX (según el estándar *POSIX*) para la manipulación de sockets; dada la semejanza con su equivalente en Windows, el apartado 5.2 se limita a mostrar las diferencias entre ellos. Éste incluye también una breve referencia acerca de las clases disponibles para la programación de sockets bajo C++ usando las MFC (*Microsoft Foundation Classes*). Por último, el apartado 5.3 hace una breve introducción a la programación de sockets en Java, lenguaje muy extendido actualmente con soporte multiplataforma, lo cual lo hace muy interesante ya que un programa escrito en Java funcionará en Windows, Linux, Solaris, HP-UX, etc. sin necesidad de recompilación.

5.1.- Programación de sockets POSIX

Los sockets en UNIX pueden distinguirse por su dominio de comunicación, esto es, atendiendo a dónde residen los procesos que se van a comunicar a través de ellos:

- Dominio UNIX (*AF_UNIX*): ambos procesos residen en la misma máquina.
- Dominio Internet (*AF_INET*): los procesos residen en máquinas posiblemente distintas conectadas mediante una red tipo TCP/IP.

Realmente, el dominio *AF_UNIX* no es necesario, pues con sockets de dominio *AF_INET* podemos comunicar dos procesos locales. Sin embargo, si restringimos el dominio a *AF_UNIX*, la comunicación será más eficiente al emplear menos recursos⁵.

Los sockets *AF_INET* admiten la clasificación que ya conocemos, en función del protocolo de transporte usado: sockets TCP (*SOCK_STREAM*) y sockets UDP (*SOCK_DGRAM*).

En esta práctica nos ceñiremos al uso de sockets de dominio *AF_INET* y de tipo *SOCK_STREAM* (TCP).

⁵ Adicionalmente se obtiene mayor seguridad, ya que procesos remotos no podrán acceder a los sockets *AF_UNIX*.

Sockets

Todos los tipos de datos y funciones que aparecen a continuación están incluidas de forma estándar en la librería *libc* que posee cualquier sistema tipo UNIX. El objetivo de los siguientes apartados es proporcionar una referencia básica; para más información, consultar las páginas de manual mediante el comando `man`.

5.1.1.- Byte order

El flujo de datos entre sockets TCP está orientado al intercambio de bytes (`char` en lenguaje C). Supongamos que queremos una secuencia de *words* (`short` en C) o *double words* (`long` en C), compuestos por 2 y 4 bytes respectivamente. ¿En qué orden debemos enviar los bytes individuales que componen cada uno de estos datos? Sistemas basados en microprocesadores distintos usan diferentes convenciones a la hora de almacenar internamente estos tipos de datos, como es el caso de los Intel x86 (primero el byte menos significativo, convención *little endian*) y Motorola 680xx (primero el byte más significativo, convención *big endian*).

Para solucionar el problema, se distinguen dos tipos de ordenamiento: el *host order* (el orden con el que se almacena en el equipo, que será *little* o *big endian* dependiendo del sistema) y el *network order* (el orden establecido por convención para las redes tipo Internet, concretamente *big endian*), y se establecen las siguientes reglas:

- todos los datos que se envíen hacia la red deben convertirse al *network order*.
- todos los datos que se reciban de la red deben convertirse al *host order*.

De esta forma, si el extremo A quiere transmitir un dato (de longitud mayor a un byte) al extremo B, lo primero que hace es convertir el dato desde el *host order* de A al *network order*. El dato, una vez recibido en B en *network order*, se transforma al *host order* de B. De esta forma, independientemente de cuáles sean los *host order* de A y B, la comunicación se realiza correctamente. La clave está en que se ha definido un *network order* universal, que siempre se respeta.

Insistimos en que el problema de orden de envío es relevante sólo cuando la unidad de información tiene una longitud superior a un byte.

Las funciones `short htons(short)` y `long htonl(long)` realizan la conversión de *host order* a *network order* de un entero tipo `short` o `long`, respectivamente, mientras que `short ntohs(short)` y `long ntohl(long)` realizan las conversiones opuestas, es decir, de *network order* a *host order*. Estas funciones están definidas en el fichero de cabecera `<netinet/in.h>`.

5.1.2.- Tipos de datos para la manipulación de sockets TCP en UNIX

```
struct in_addr
```

Estructura que representa una dirección IP.

`unsigned long s_addr`: entero de 32 bits que almacena la dirección IP. Debe estar en *network order*. La constante `INADDR_ANY` puede usarse para especificar la dirección local.

```
struct sockaddr_in
```

Estructura que representa la dirección de un socket de dominio Internet.

struct sockaddr_in

`short int sin_family`: dominio del socket, que, en este tipo de datos, debe especificar el dominio Internet, mediante la constante `AF_INET`.

`unsigned short sin_port`: número de puerto asociado al socket. Debe estar en network order.

`struct in_addr sin_addr`: dirección IP de la máquina correspondiente al socket.

`unsigned char sin_zero[8]`: debe estar inicializado a 0s (por ejemplo, usando `memset()` o `bzero()`).

5.1.3.- Funciones para la manipulación de sockets TCP en UNIX**int socket(int domain, int type, int protocol)**

Función que crea o “abre” un socket.

`int domain`: dominio del socket a crear. En nuestro caso, usar la constante `AF_INET`.

`int type`: tipo de socket. En nuestro caso, `SOCK_STREAM`.

`int protocol`: protocolo asociado al socket. Se recomienda el valor 0 para que la función escoja automáticamente el protocolo más adecuado según el tipo de socket especificado (que en nuestro caso, será TCP).

Devuelve (`int`): descriptor del nuevo socket. El valor -1 indica que no se pudo crear el socket.

int bind(int s, struct sockaddr *name, int namelen)

Asigna una dirección (dirección IP y puerto) al socket especificado. Esto hace que sea posible referenciar al socket desde un proceso remoto.

`int s`: descriptor del socket al que se le va a asignar una dirección.

`struct sockaddr *name`: dirección a asignar al socket. Para nosotros, `name` debe ser una variable de tipo `struct sockaddr_in`, por lo que es necesario realizar un *cast*, de la forma `(struct sockaddr *) name`, al hacer la llamada a `bind()`.

La función `bind()` no es exclusiva de sockets TCP, lo cual explica que este parámetro sea de tipo `struct sockaddr`, más general que `struct sockaddr_in`.

`int namelen`: tamaño de la estructura `struct sockaddr`, calculado con el operador `sizeof()`.

Devuelve (`int`): 0 si la operación tuvo éxito; -1 indica un error.

int listen(int s, int backlog)

Prepara un socket para ser utilizado en modo servidor, asignándole una cola de conexiones pendientes de procesar.

`int s`: descriptor del socket sobre el que se va a actuar.

`int backlog`: tamaño máximo de la cola de conexión.

Devuelve (`int`): 0 si la operación tuvo éxito; -1 indica un error.

int accept(int s, struct sockaddr *addr, int *addrlen)

Pone a un socket servidor en estado de escucha, monitorizando la cola de solicitudes de conexión. Cuando recibe una solicitud desde un cliente, y ésta es aceptada, crea un socket (con el mismo número de puerto que el socket servidor) que es conectado al socket cliente.

Mientras que no se reciban solicitudes de conexión, la función `accept ()` no devuelve el control (esto es, presenta bloqueo).

`int s`: descriptor del socket sobre el que se va a actuar.

`struct sockaddr *addr` (parám. salida): dirección del socket cliente que ha solicitado la conexión. En nuestro caso, que tratamos con sockets TCP, `addr` debe ser una variable de tipo `struct sockaddr_in`, por lo que habrá que hacer un cast al llamar a esta función (análogamente a `bind ()`).

`int *addrlen` (parám. salida): tamaño de la estructura devuelta en `addr`.

Devuelve (`int`): descriptor del socket creado. Este socket sólo es válido para la conexión recién iniciada. El valor `-1` indica que no se pudo crear el socket.

int connect(int s, struct sockaddr *name, int namelen)

Trata de establecer una conexión entre el socket local (cliente) `s` y el socket remoto (servidor) identificado por `name`.

`int s`: descriptor del socket local (cliente).

`struct sockaddr *name`: dirección del socket servidor con el que se desea conectar. En nuestro caso, que tratamos con sockets TCP, `name` debe ser una variable de tipo `struct sockaddr_in`, por lo que habrá que hacer un cast al llamar a esta función (análogamente a `bind ()`).

`int namelen`: tamaño de la estructura dada en `name`.

Devuelve (`int`): 0 si la operación tuvo éxito; `-1` indica un error.

int send(int s, const char *msg, int len, int flags)

Envía una secuencia de bytes a través del socket `s`.

`int s`: descriptor del socket a través del cual se quieren enviar datos.

`const char *msg`: secuencia de bytes (`char`) a enviar.

`int len`: número de bytes a enviar de `name`.

`int flags`: en nuestro caso, no se utilizarán flags, de forma que este parámetro debe ser 0.

Devuelve (`int`): el número de bytes realmente enviados; `-1` indica un error.

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

Recibe una secuencia de bytes a través del socket `s`. La función posee bloqueo, esto es, no se devuelve el control hasta que se recibe algo.

`int s`: descriptor del socket a través del cual se quieren recibir datos.

`char *buf`: buffer en el que se almacenan los datos recibidos.

`int len`: tamaño del buffer `buf`.

`int flags`: en nuestro caso, no se utilizarán flags, de forma que este parámetro debe ser 0.

```
int recv(int s, char *buf, int len, int flags)
```

Devuelve (int): el número de bytes recibidos; -1 indica un error.

```
int close(int s)
```

Cierra el socket s.

int s: descriptor del socket a cerrar.

Devuelve (int): 0 si la operación tuvo éxito; -1 indica un error.

Para poder utilizar estas funciones hay que incluir los ficheros de cabecera `<sys/types.h>` y `<sys/socket.h>`.

Es destacable que muchas de estas funciones utilizan la convención de devolver el valor -1 ante cualquier posible error. Cuando esto se produce, el sistema modifica el valor de una variable global, denominada `errno`, asignándole un valor que indica el tipo de error que se ha producido⁶. La función `perror()` emite un mensaje de error en función del valor de `errno`. Estas características están disponibles al incluir el fichero de cabecera `<stdio.h>`. Para más información, consultar las páginas de manual `man`.

Otras funciones interesantes, aunque no fundamentales, relacionadas con sockets son: `inet_addr()` e `inet_ntoa()`, para el tratamiento de direcciones IP; `gethostname()`, `gethostbyname()` y `getpeername()` para el uso del servicio de nombres de dominio (DNS), traduciendo direcciones IP en nombres/dominios de hosts y viceversa; `getprotoent()` y `getprotobyname()` para la selección de protocolos a la hora de crear sockets; `select()`, para la monitorización de un conjunto de sockets; `fcntl()`, para modificar las propiedades de un socket (como por ejemplo, hacer no bloqueantes las llamadas a `accept()` o `recv()`), etc. Si se desea realizar programación orientada a eventos, hay que usar el mecanismo de las señales de UNIX (funciones `signal()`, `sigaction()`, etc.). Para más información sobre ellas, consultar las páginas de manual `man`.

5.2.- Programación de sockets en Windows

Bajo Windows, las operaciones de sockets son competencia de la librería dinámica `winsock2` (*Windows Sockets Version 2*). Básicamente, el API `winsock2` consta de las funciones estilo POSIX descritas en el apartado anterior, aunque incluyendo algunas modificaciones, y de un conjunto de funciones propias, que, por convención, están precedidas por el prefijo `WSA-`. Tan sólo trataremos con las más fundamentales de estas funciones añadidas tipo `WSA-`. A continuación se enumeran los aspectos propios de Windows en cuanto a la programación de sockets:

- Para poder usar las funciones de sockets en Windows, hay que incluir el fichero de cabecera `<winsock2.h>` en lugar de `<sys/socket.h>`, `<sys/types.h>` y `<netinet/in.h>`.
- Es necesario añadir al enlazador (linker) la librería estática `ws2_32.lib`. En Microsoft Visual C++ 6.0, esto se hace incluyéndola en *Project > Settings > Link > Object/Library Modules*.

⁶ Los posibles valores que puede tomar `errno` se han definido como constantes en el fichero de cabecera `<errno.h>`.

- Los descriptors de sockets, que en POSIX son tipo `int`, aquí son de tipo `SOCKET`. Por ejemplo, el prototipo de la función `listen()` pasa a ser `int listen(SOCKET s, int backlog)`; análogamente con el resto de funciones.
- Las funciones que devuelven un descriptor de socket utilizan ahora la constante `INVALID_SOCKET` para indicar un error (en lugar de devolver `-1`).
- El resto de funciones que manejan sockets (que devuelven `int`) indican una situación de error mediante la constante `SOCKET_ERROR` (en lugar de devolver `-1`). Es el caso, por ejemplo, de `send()` y `recv()`.
- La función `close()` en winsock2 se denomina `closesocket()`, aunque manteniendo la misma signatura.
- En winsock2, la variable `errno` no es actualizada con el error producido. En su lugar, hay que invocar a `WSAGetLastError()` para obtener el código de error.
- Antes de realizar cualquier llamada a winsock2 es necesario invocar la función `WSAStartup()`, que se encarga de iniciar esta librería. Análogamente, cuando ya no sea necesario seguir usando winsock2, debe llamarse a `WSACleanup()`. Típicamente, el código necesario para llamar a `WSAStartup()` es de la forma:

```
WORD wVersionRequested;  
WSADATA wsaData;  
wVersionRequested = MAKEWORD(2, 2);  
if(WSAStartup(wVersionRequested, &wsaData) != 0) {  
    printf("Error: WSAStartup\n");  
    exit(1);  
}
```

Mientras que para `WSACleanup()`:

```
if(WSACleanup() == SOCKET_ERROR) {  
    printf("Error: WSACleanup\n");  
    exit(1);  
}
```

Para más información sobre la programación de sockets en Windows, puede consultarse la *MSDN Library Visual Studio*, en *Platform SDK > Networking and Distributed Services > Windows Sockets Version 2 API*.

5.2.1.- Programación de sockets en Windows usando C++ y MFC

Microsoft ha desarrollado un conjunto de clases C++ que encapsulan prácticamente todo en API de Windows, incluyendo las funciones de manejo de sockets; son las *clases base de Microsoft* o MFC (*Microsoft Foundation Classes*). Por lo tanto, es una alternativa a considerar especialmente si estamos desarrollando una aplicación con un enfoque orientado a objetos para Windows. En MFC, se definen dos clases relacionadas con sockets:

- *CAsyncSocket*. Encapsula las funciones descritas en el apartado anterior, facilitando además la posibilidad de realizar programación orientada a eventos, de forma que se pueden definir funciones de callback que son notificadas cuando se ha recibido información, cuando el socket está libre para seguir enviando, etc.
- *CSocket*. Deriva de la anterior. Proporciona un interfaz con un mayor nivel de abstracción, evitando detalles que sí aparecen en *CAsyncSocket*. En particular, las funciones de *CSocket* incorporan bloqueo.

Para más información, consultar la *MSDN Library Visual Studio*.

5.3.- Programación de sockets mediante Java

Independientemente de si trabajamos en Windows o en alguna variante de UNIX, tenemos la opción de programar en Java. Por ello incluimos este apartado fuera de los dos anteriores. Una de las ventajas de Java es que el mismo programa puede ser ejecutado prácticamente en cualquier plataforma sin necesidad de adaptar el código fuente, y sin ni siquiera recompilarlo. Además de esta característica, Java tiene la ventaja de que posee un conjunto muy extenso de clases, agrupadas en paquetes, que facilitan la tarea de programación. En particular, el paquete `java.net` proporciona un pequeño conjunto de clases que permiten trabajar con sockets de manera muy cómoda.

- En particular, las clases `Socket` y `ServerSocket` encapsulan toda la funcionalidad de los sockets (TCP) cliente y servidor, respectivamente (los sockets que, en el servidor, se encargan del intercambio de información con los clientes, son conceptualmente idénticos a los sockets cliente, y por tanto, se representan también mediante `Socket`).
- Los sockets tipo UDP están disponibles mediante `DatagramSocket`, que intercambian datagramas, a su vez representados por la clase `DatagramPacket`.
- La clase `InetAddress` permite tratar direcciones IP y acceder al DNS.

Para más información, consultar la documentación del API del SDK de Java.

6.- DESARROLLO DE LA PRÁCTICA

El objetivo es, básicamente, implementar un servidor web sencillo, para llevar a la práctica todo lo aprendido sobre sockets. En primer lugar vamos a comentar algunos aspectos básicos que debemos conocer sobre servidores web. Con ello, estaremos en posición de dar el enunciado de la práctica, que se complementa con una descripción del protocolo de aplicación que debe implementar nuestro servidor.

6.1.- Nociones básicas sobre servidores web.

La web utiliza el protocolo de aplicación HTTP⁷ (*HyperText Transfer Protocol*) que se apoya sobre TCP (ver figura 1). Este protocolo trabaja en una configuración cliente-servidor. Se trata de un protocolo sin estados, cuyos mensajes pueden ser de dos tipos: solicitudes (`requests`) o respuestas (`responses`). El escenario siguiente representa una sesión HTTP típica entre un cliente y un servidor:

- El cliente envía una solicitud HTTP al servidor, la cual incluye obligatoriamente el identificador del recurso al que se desea acceder y, opcionalmente, información sobre el cliente, sobre la versión utilizada de HTTP, etc. Es frecuente que el identificador de recurso especifique un fichero que reside en el servidor, y que el cliente desea obtener a través de la red. Es la situación que se produce cuando con el navegador web queremos visualizar una página, que normalmente no es más que un fichero contenido en un servidor.
- El servidor analiza la solicitud, y en base a ella accede al recurso indicado, realizando sobre el la operación que se haya pedido. Frecuentemente, el recurso es un fichero y la operación solicitada, la descarga del mismo a través de la red. El servidor construye una respuesta HTTP

⁷ HTTP está definido en RFC2616.

y la envía al cliente. Esta respuesta contiene información de estado que indica el resultado de la operación realizada en el servidor, y, en su caso, la información resultante de procesar la solicitud; en el supuesto anterior, el propio fichero solicitado.

6.1.1.- Especificando recursos en HTTP

El identificador de recurso viene dado normalmente como un localizador o URL (*Uniform Resource Locator*), que es el término con el que nos referimos a las direcciones con las que estamos acostumbrados a tratar en los navegadores web. Su formato (simplificado) es el siguiente:

http://host[:puerto][/path]

que en realidad es un caso particular, para el protocolo HTTP, del formato general:

protocolo://host[:puerto][/path]

Un URL nos indica mediante qué *protocolo* (en nuestro caso, nos limitaremos a HTTP), en qué socket (indicado por *host*⁸ y *puerto*) y bajo qué *path* podemos encontrar un cierto recurso. Por ejemplo, `http://www.dte.us.es/ing_inf/arc2/index.htm` especifica la página inicial de la asignatura, que se corresponde con el fichero `/ing_inf/arc2/index.htm` almacenado en el servidor `www.dte.us.es` y accesible a través del puerto 80 (que es el puerto que se toma por defecto para el protocolo HTTP, cuando no se explicita ninguno).

El path indicado en el URL no es necesariamente el path local bajo el que reside el fichero en el servidor. Normalmente, los servidores disponen de un **directorio de publicación**, por ejemplo, `C:\InetPub\wwwroot\` en Windows (con el servidor *Microsoft Personal Web Server*, por ejemplo, disponible en Windows 98). Bajo este directorio de publicación se colocan todos los ficheros que queremos que sean accesibles desde el exterior mediante web. El servidor web, cuando recibe en la solicitud HTTP un URL, considera que el path indicado realmente es relativo respecto del directorio de publicación. Así, si se nos solicita un URL cuyo path es `/ing_inf/arc2/index.htm`, el servidor lo convertirá en el path local especificado por `C:\InetPub\wwwroot\ing_inf\arc2\index.htm`. El directorio de publicación es configurable en el servidor. Habitualmente, el servidor viene acompañado de alguna herramienta que nos permite configurar este tipo de opciones.

Hay dos cuestiones importantes a señalar en cuanto al mecanismo de conversión entre paths URL y paths locales que acabamos de describir.

- A veces, el path especifica un directorio en lugar de un fichero. Realmente esto es muy común; por ejemplo, cuando escribimos `http://www.dte.us.es/` realmente estamos especificando el path `/`, y no indicamos ningún fichero en concreto. En este caso, el servidor automáticamente busca en ese directorio el **documento predeterminado**, habitualmente `index.html`. De esta forma, al recibir el URL `http://www.dte.us.es`, el servidor lo convertirá automáticamente a `http://www.dte.us.es/index.html`. El nombre del documento predeterminado es configurable en el servidor, siendo comunes también los nombres `index.htm` y `default.htm`.

⁸ El host puede identificarse por su dirección IP o por su nombre, registrado en el servicio DNS (ver apartado 3.1 para más información).

- Normalmente, los servidores web permiten la definición de **alias**. Por ejemplo, en Windows, un directorio tal como `C:\InetPub\Scripts` estaría fuera del directorio de publicación, y por ello, no sería accesible desde el exterior. Sin embargo, el servidor puede permitir asociar un path URL virtual, por ejemplo, `/cgi-bin`, a un path local real, `C:\InetPub\Scripts`, de forma que un URL tal como `http://www.dte.us.es/cgi-bin/script1` se convierte a `C:\InetPub\Scripts\script1`. Se dice entonces que `/cgi-bin` es un *alias* del path local `C:\InetPub\Scripts`. Con este enfoque, `/` no es más que un alias para el directorio de publicación (`C:\InetPub\wwwroot` en nuestro ejemplo).

6.1.2.- HTML

HTML (*HyperText Transfer Language*) es un lenguaje de *markups* o etiquetas que permite dar formato al texto y a las imágenes de una página web: especificar tamaño y tipos de letra, tamaño y posición de las imágenes, definición de tablas, formularios sencillos, etc. aunque quizá su característica fundamental sea la capacidad para definir *hiperenlaces*, esto es, referencias a otras páginas, lo cual proporciona la posibilidad de navegación entre ellas.

Las etiquetas HTML son palabras reservadas encerradas entre corchetes angulares, `<` y `>`. Habitualmente van por pares, es decir, existe una etiqueta que abre un entorno y una similar que lo cierra. En ese entorno, se aplica el formato asociado a la etiqueta. Por ejemplo, para poner un texto en negrita, lo especificamos del siguiente modo:

```
<B>Este texto va en negrita.</B>
```

donde `` (*Bold*) es la etiqueta de apertura del entorno “negrita” y ``, la etiqueta de cierre del mismo. Tras interpretar el código anterior, un visor HTML nos mostraría lo siguiente:

Este texto va en negrita.

Las etiquetas de cierre son iguales que las etiquetas de apertura, pero anteponiendo el símbolo `'/'` al nombre de la etiqueta. Existen etiquetas de apertura de las cuales no existe su correspondiente etiqueta de cierre.

En resumen, HTML nos permite especificar el *contenido estático* de una página, básicamente texto e imágenes. La interacción con el usuario es posible a través de formularios. Sin embargo, HTML ha previsto la inclusión de extensiones que permiten aumentar las posibilidades de presentación e interacción con el usuario de las páginas web. Estas extensiones permiten incorporar *contenido dinámico* a las páginas.

6.1.3.- Contenido dinámico

Hablamos de *contenido dinámico* cuando nos referimos a que la página que visualizamos nos permite un grado de *interactividad* superior al soportado por HTML, o bien cuando la página ha sido *generada* dinámicamente por el servidor para nosotros, en lugar de existir previamente en el servidor. Estos dos enfoques dan como resultado dos formas de lograr el comportamiento dinámico:

- Ejecución en el lado cliente (*client-side*). Junto con la página se descarga hacia el navegador algún tipo de código. Este código se ejecuta en el cliente, y de alguna forma permite una

mayor interactividad del usuario con la página. En esta categoría encontramos:

- *Scripts*: a veces embebidos en el propio código HTML, los lenguajes de script permiten añadir funcionalidad a una página. Está extendido el uso de JavaScript y VBScript (Visual Basic Script de Microsoft). Su uso es frecuente, por ejemplo, a la hora de comprobar si los datos introducidos en un formulario son erróneos (si un número de teléfono demasiado corto, si un NIF no incluye la letra, etc.) antes de enviar los datos hacia el servidor. También es frecuente su uso para proporcionar efectos visuales a la página: fundidos en negro, barridos, etc. Muy potentes, teniendo la posibilidad incluso de interactuar con controles ActiveX o JavaBeans.
 - *Applets Java*: el lenguaje Java permite crear aplicaciones con interfaz gráfico de usuario que se muestran en una ventana del navegador, formateadas como un elemento más de la página. Los applets suelen utilizarse para tareas que van desde generar animaciones relativamente complejas en la página, hasta interaccionar con bases de datos y aplicaciones web⁹. Gracias a que están escritos en Java, son ejecutables en cualquier plataforma. El uso de un lenguaje potente como es Java proporciona una gran flexibilidad a la hora de crear applets, existiendo en la red aplicaciones tan dispares como la generación de coeficientes de filtros digitales, simuladores sencillos de circuitos electrónicos o cálculo de una efemérides astronómica.
 - *Animaciones Macromedia / Shockwave Flash*: la herramienta visual Macromedia Flash nos permite crear animaciones con gráficos y texto, con capacidad de interacción con el usuario en base a botones y otros componentes gráficos. La animación se almacena en un fichero accesible mediante web y entendible por el navegador (en realidad, gracias a una extensión o *plug-in* del navegador, en este caso específica para Flash). Es muy usado para la creación de las páginas principales de los sitios web, dada su calidad gráfica y flexibilidad de diseño. Sirva como ejemplo la página principal de la web del Departamento de Tecnología Electrónica, <http://www.dte.us.es>, que ha sido desarrollada en Flash.
- Ejecución en el lado servidor (*server-side*). Normalmente relacionado con el tratamiento de formularios enviados por los usuarios mediante un navegador. Este tratamiento supone una ejecución de código por parte del servidor, que con frecuencia consiste en una serie de consultas y/o actualizaciones de una base de datos accesible desde el servidor. Existen varias alternativas:
- *CGI (Common Gateway Interface)*: es la aproximación que podríamos denominar “clásica”. Un programa CGI no es más que un ejecutable normal que reside en el servidor (habitualmente bajo el path virtual `/cgi-bin` o `/Scripts`), que es invocado por éste cuando el usuario envía el formulario (pulsando el botón *submit/enviar* del mismo). Se suelen programar en lenguaje Perl, que proporciona librerías específicas para esta labor, aunque dado que se trata de un ejecutable normal, puede ser escrito en C, C++, etc.
 - *Servlets*: la alternativa a los CGI usando Java.
 - *Scripts de lado servidor*: últimamente existe una cierta proliferación de lenguajes de script que permiten añadir código ejecutable a las páginas HTML, como es el caso de ASP, JSP y PHP. Este código es ejecutado por el servidor antes de servir la página;

⁹ aplicaciones cuyo interfaz de usuario es accesible mediante web

el resultado de la ejecución no es más que código HTML que completa dinámicamente la página web solicitada, lista ahora para ser entregada al navegador. Supongamos, por ejemplo, que rellenamos un formulario para hacer una búsqueda en una base de datos. El navegador envía los datos hacia la página de búsqueda, que incorpora un esqueleto HTML y scripts de lado servidor. El servidor ejecuta los scripts, que acceden a la base de datos y formatean el resultado de la búsqueda en HTML. Este código resultante se inserta en la página de resultados, sustituyendo a los scripts, y se entrega al navegador. Desde el punto de vista de éste, la página de resultados está escrita en HTML convencional, aunque en realidad ha sido generada, en parte, dinámicamente por el servidor.

6.1.4.- Seguridad

Actualmente, y dada la expansión del comercio electrónico (*e-commerce*), es una característica fundamental a tener en cuenta en un servidor web.

La seguridad, a través de la web se consigue aplicando técnicas de criptografía al intercambio de información entre cliente y servidor. Para este intercambio se utiliza el protocolo HTTPS, que es una extensión de HTTP que soporta conexiones seguras mediante SSL (*Secure Socket Layer*, ahora denominadas también TLS, *Transport Level Security*), mecanismo que añade encriptación a las conexiones.

6.1.5.- Ejemplos de servidores web

A continuación se enumeran algunos de los servidores web¹⁰ comerciales más conocidos.

- *Apache*: es el servidor con mayor penetración de mercado: 63%. Es un servidor muy potente, con soporte php, servlets, etc. y con la particularidad de que es distribuido bajo licencia freeware. Existen versiones para las distintas variantes de UNIX (incluido Linux) y Win32.
- *iPlanet Enterprise*: desarrollado conjuntamente por Netscape y Sun (bajo el sello iPlanet). Existen versiones para las distintas variantes de UNIX, y para Windows NT.
- *Microsoft Internet Information Server (IIS)*: incorporado en los sistemas operativos Windows NT Server, Windows 2000 Server y Advanced Server.
- *Microsoft Personal Web Server (PWS)*: servidor básico, para Win32, incluido en Windows 98 SE. Se distribuye también bajo licencia freeware.

6.2.- Enunciado

El objetivo de la práctica es implementar un servidor web sencillo, pero capaz de interactuar con los navegadores convencionales tales como *Netscape Navigator* y *Microsoft Internet Explorer*. El servidor a implementar deberá poseer las siguientes características:

- La implementación deberá basarse en funciones de manejo de sockets (TCP). Se permite la entrega de la práctica en C bajo UNIX o Windows, C++ (usando MFC) bajo Windows, o Java (cualquier plataforma). Se prohíbe el uso de librerías no incluidas de forma estándar en estos entornos, a menos que sean desarrolladas por el propio alumno, en cuyo caso formarán parte del código a entregar. Los entornos de desarrollo serán los siguientes: compilador *cc* o *gcc* en

¹⁰ Consultar <http://serverwatch.internet.com/webserver.html> para más información sobre servidores web.

- UNIX, *Microsoft Visual C++* en Windows, y *Java 2 SDK* (JDK) para trabajar en lenguaje Java (cualquier plataforma).
- El servidor deberá responder a solicitudes HTTP siguiendo las especificaciones del apartado 6.3, que define el subconjunto a implementar de dicho protocolo.
 - El servidor será *monohilo* con el objeto de simplificar la implementación. Esto es, el proceso servidor no creará un nuevo proceso hijo al aceptar una conexión, sino que él mismo se encargará de intercambiar información con el cliente. No obstante, se valorarán positivamente las prácticas que implementen adecuadamente la versión multihilo del servidor web que aquí se especifica.
 - El servidor es un proceso de segundo plano y por tanto, no debe presentar interfaz de usuario. Toda la salida que se estime oportuno deba generar el servidor deberá ser dirigida a un fichero de *log* (histórico).
 - El servidor dispondrá de un directorio de publicación¹¹ y definirá un documento por defecto. No es obligatorio ofrecer la posibilidad de configurar el directorio de publicación y el documento predeterminado, pero se valorarán positivamente aquellas implementaciones que lo permitan. Tampoco es necesario implementar el mecanismo de alias, aunque se valorará positivamente si se hace.
 - El servidor web usará el puerto que se le especifique en la línea de comandos. Esto nos permitirá usar otros puertos diferentes del puerto por defecto para HTTP (80) durante las pruebas, evitando así interferir con el servidor web existente en caso de que hubiese uno instalado. Normalmente trabajaremos con números de puerto altos, como, por ejemplo, 8000, para evitar usar puertos ya asignados a aplicaciones estándar de Internet.

6.3.- Descripción del subconjunto HTTP a implementar

Tal como se mencionó anteriormente, HTTP¹² presenta dos tipos de mensajes: solicitudes (*requests*) y respuestas (*responses*). Los mensajes tienen una cabecera de texto ASCII, compuesta de varios campos, y un cuerpo de mensaje que puede estar en cualquier formato. En las solicitudes, el cuerpo del mensaje suele estar vacío, o bien contener los datos de un formulario. En las respuestas, el cuerpo del mensaje suele contener páginas HTML, imágenes, ficheros comprimidos, ficheros ejecutables, etc.

6.3.1.- Solicitudes HTTP

El formato general de una solicitud HTTP es el siguiente (CRLF = caracteres retorno de carro, avance de línea):

```
Request = Request-Line
        *(( general-header
          | request-header
          | entity-header ) CRLF)
        CRLF
        [ message-body ]
```

De todos estos campos, los más interesantes para nosotros son Request-Line y message-body.

¹¹ El directorio de publicación deberá estar bajo la cuenta de un usuario normal si se decide trabajar en UNIX para resolver la práctica (ya que normalmente no tendremos acceso a los permisos del superusuario).

¹² Consultar RFC2616 para una descripción detallada de HTTP/1.1.

Sockets

Las cabeceras `general-header`, `request-header` y `entity-header` son opcionales pero frecuentemente implementadas, ya que proporcionan mucha información al servidor. En nuestro caso, serán ignoradas por simplicidad.

La sintaxis de `Request-Line` es la siguiente (SP = carácter espacio en blanco):

```
Request-Line = Method SP Request-URI SP HTTP-Version CRLF
```

Con esta línea, el cliente HTTP le solicita al servidor que se efectúe la operación indicada por `Method` al recurso indicado por `Request-URI` (**en nuestro caso, un path URL**), usando la versión del protocolo indicada con `HTTP-Version`. Ésta se indica mediante la cadena:

```
HTTP-Version = "HTTP/" version
```

donde `version` será 0.9, 1.0 ó 1.1.

Aunque existen varias opciones para `Method`, las fundamentales son GET y POST, ambas usadas por el cliente para solicitar al servidor que le envíe el recurso indicado por `Request-URI` en un mensaje de respuesta. La diferencia entre GET y POST está relacionada con el envío de datos (por ejemplo, procedentes de un formulario) hacia el servidor.

- GET: los datos a enviar al servidor se incluyen en el propio URL (lo cual repercute en `Request-URI`), de la siguiente forma:

```
http://host[:puerto][ /path[?query]]
```

donde *query* es una sucesión de pares *nombre=valor*, separados por el carácter '&'. Por ejemplo, un formulario para consultar las notas de una asignatura podría originar una solicitud HTTP requiriendo el URL

```
http://host.dominio/cgi-bin/notas.cgi?dni=44555666&asig=arc2
```

lo cual, en la solicitud HTTP, implicaría una `Request-Line` como ésta:

```
GET /cgi-bin/notas.cgi?dni=44555666&asig=arc2 HTTP/0.9
```

- POST: los datos a enviar al servidor se incluyen en el campo `message-body`. Este método es preferible al anterior cuando hay muchos datos que enviar.

Nuestro servidor, por simplicidad, sólo atenderá a solicitudes tipo GET, suponiendo además que el cliente no envía datos, es decir, que los URL requeridos no incorporan *query*. Por tanto, los recursos solicitados siempre serán ficheros locales al servidor.

6.3.2.- Respuestas HTTP

El formato general de un mensaje de respuesta HTTP es el siguiente:

```
Response = Status-Line  
          *(( general-header  
             | response-header
```

```
| entity-header ) CRLF)
CRLF
[ message-body ]
```

Para nosotros, sólo son relevantes los campos `Status-Line`, `message-body` y una cabecera de tipo `entity-header`, concretamente `Content-Type`. Por tanto, el formato de `Response` se simplifica a:

```
Response = Status-Line
          entity-header CRLF
          CRLF
          message-body
```

```
entity-header = "Content-Type" ":" media-type
```

El campo `Status-Line` indica el resultado de la operación solicitada por el cliente. Tiene el siguiente formato (SP = carácter espacio en blanco; CRLF = caracteres retorno de carro, avance de línea):

```
Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

Nuestro servidor siempre responderá con `HTTP/0.9` en `HTTP-Version`. El resultado de la operación se indica mediante `Status-Code` y `Reason-Phrase`. El valor de `Status-Code` es un entero de 3 dígitos, donde el primero indica la categoría de la respuesta HTTP, de este modo:

```
1xx: Informational - Request received, continuing process
2xx: Success - The action was successfully received, understood, and accepted
3xx: Redirection - Further action must be taken in order to complete the request
4xx: Client Error - The request contains bad syntax or cannot be fulfilled
5xx: Server Error - The server failed to fulfill an apparently valid request
```

Nuestro servidor sólo contemplará los siguientes posibles valores para `Status-Code`:

```
200: La solicitud ha sido tratada con éxito. En nuestro caso, esto supone que el recurso
solicitado existe y se devuelve en el campo message-body. La Reason-Phrase asociada
es "OK".
404: El recurso solicitado no existe. La Reason-Phrase asociada es "Not Found".
501: El servidor no soporta la funcionalidad requerida para resolver la solicitud. Es lo que
nuestro servidor debe responder ante cualquier otra solicitud que no sea la estipulada en el
apartado anterior. La Reason-Phrase asociada es "Not Implemented".
```

En los dos últimos casos, que corresponden a situaciones de error, debe devolverse, en el campo `message-body`, un fragmento de código HTML que permita visualizar en el navegador un mensaje de error. Por ejemplo, para el error 404, debemos devolver en `message-body`, lo siguiente:

```
<html>
<head><title>404: No encontrado</title></head>
<body><font face="Arial">
<h1>No encontrado</h1>
Error 404: el recurso solicitado no ha sido encontrado.
</font></body>
```


</html>

lo cual se muestra en el navegador como:

No encontrado

Error 404: el recurso solicitado no ha sido encontrado.

Análogamente para el error 501.

Tal como se ha indicado previamente, para nuestra implementación simplificada, la única cabecera a devolver es la siguiente:

```
entity-header = "Content-Type" ":" media-type
```

donde `media-type` indica al cliente HTTP el tipo de recurso (fichero si no hubo error) devuelto en el campo `message-body`. El campo `media-type` espera la especificación del tipo en formato MIME (Multipurpose Internet Mail Extension), que no es más que una forma textual de describir el tipo de un recurso. En esta práctica, los tipos MIME que deben ser reconocidos por nuestro servidor son:

- `text/html`: página HTML, asociada a las extensiones `.html` y `.htm`
- `text/plain`: texto ASCII sin formato, asociado a la extensión `.txt`
- `image/gif`: imagen en formato GIF, asociada a la extensión `.gif`
- `image/jpeg`: imagen en formato JPEG, asociada a las extensión `.jpg`

Si el servidor debe devolver cualquier otro tipo de fichero (por ejemplo, `.zip`), indicará que su tipo MIME es `application/octet-stream`, que es el más general posible.

6.3.3.- Ejemplo

Consideremos, a modo de ejemplo, una interacción típica entre cliente y servidor.

Supongamos que el usuario del navegador escribe en la barra de dirección, o bien selecciona un hipervínculo de tal forma que el navegador ahora deba mostrar la página correspondiente al URL `http://www.dte.us.es/index.htm`. El navegador envía una solicitud HTTP al servidor `www.dte.us.es`, de esta forma:

```
GET /index.htm HTTP/1.1
Accept: image/gif, image/jpeg, application/msword, */*
Accept-Language: es
... (otras cabeceras)
(retorno de carro)
```

A lo que nuestro servidor contestará, suponiendo que exista el fichero `index.htm`:

```
HTTP/0.9 200 OK
Content-Type: text/html
(retorno de carro)
... (el contenido del fichero index.htm)
```

El navegador interpreta el código HTML escrito en `index.htm` y detecta que para visualizar la página,

Sockets

necesita de una imagen, concretamente la dada por <http://www.dte.us.es/img/fondo.jpg>. Entonces, la solicita al servidor, análogamente a como lo hizo antes con *index.htm*:

```
GET /img/fondo.jpg HTTP/1.1
Accept: image/gif, image/jpeg, application/msword, */*
Accept-Language: es
... (otras cabeceras)
(retorno de carro)
```

El servidor ahora contesta con:

```
HTTP/0.9 200 OK
Content-Type: image/jpeg
(retorno de carro)
... (el contenido del fichero fondo.jpg)
```

Supongamos por un momento que, por error del administrador del servidor web, la imagen no se localiza bajo el path indicado. Entonces nuestro servidor informa de la situación e incluye el mensaje de error que se mostraba en el apartado anterior:

```
HTTP/0.9 404 Not Found
Content-Type: text/html
(retorno de carro)
<html>
<head><title>404: No encontrado</title></head>
<body><font face="Arial">
<h1>No encontrado</h1>
Error 404: el recurso solicitado no ha sido encontrado.
</font></body>
</html>
```

7.- CRITERIOS DE VALORACIÓN Y PRESENTACIÓN

7.1.- Criterios generales

En el curso 2000/2001 se realizarán dos prácticas de laboratorio obligatorias, que se valorarán de 0 a 10 puntos. La nota media de las prácticas supondrá un 10% de la nota final de la asignatura, correspondiendo el 90% restante a la nota del examen final. El peso de cada práctica se especificará en la misma.

A no ser que se indique lo contrario, y con carácter general, será de aplicación lo siguiente:

- Los estudios teóricos se realizarán de manera **individual**. Toda la documentación se entregará perfectamente identificada con el nombre del alumno y grupo y puesto al que pertenece, indicando alguna forma de contacto (teléfono o dirección de correo electrónico).
- Cuando haya que presentar un programa, éste deberá encontrarse libre de errores de compilación y se entregará en un (o varios si procede) disquete(s) en los que se incluya todo y sólo lo necesario para obtener el ejecutable. Dicho disco deberá identificarse conforme a lo especificado más arriba.
- Para acceder a las sesión de laboratorio será necesario presentar el estudio teórico.
- El estudio práctico se completará durante la sesión de prácticas y se entregará a la finalización de ésta.
- No es necesario presentar el trabajo a mano.
- Se valorará la presentación en general, así como la corrección, versatilidad y estilo de

- programación.
- Está expresamente prohibida la realización de la práctica en grupos.
- La documentación que se aporte podrá recogerse en el lugar que se designe una vez publicadas las actas finales de la asignatura.
- Las prácticas se realizarán en el laboratorio L4 situado en la segunda planta del Edificio Rojo.

NO SE CONVALIDAN LAS PRÁCTICAS DE OTROS AÑOS

Los alumnos que suspendan o no hayan realizado alguna de las prácticas (o partes de ellas) deberán superar un examen final de prácticas que coincidirá con la fecha del examen final de teoría. Por motivos de organización aquellos alumnos que deseen realizar este examen deberán comunicárselo a los profesores al menos 7 días antes.

El examen de la práctica se valorará como APTO o NO APTO, **no puntuando en la nota final de la asignatura**. La calificación final coincidirá con la del examen de teoría.

7.2.- Criterios específicos de la segunda práctica

La segunda práctica de ARC2 se regirá, aparte de lo apuntado en el epígrafe anterior, por lo siguiente:

- Se puntuará entre 0 y 5 puntos.
- Se permite la entrega individual o en grupos de dos alumnos como máximo.
- No será presencial; es decir, no habrá una sesión especial de laboratorio para realizar los trabajos, sino que el laboratorio permanecerá abierto con acceso libre en los horarios que designe el personal de éste. En el laboratorio de la asignatura se indicará debidamente qué ordenadores pueden usarse.
- La práctica deberá entregarse en una convocatoria oficial de la asignatura, coincidiendo con el examen teórico.
- Las consultas sobre la práctica serán atendidas en horario de tutorías por el prof. Sergio Díaz, en el despacho 3.26 del Departamento de Tecnología Electrónica (Edificio Rojo, 3ª planta).
- Los alumnos que entreguen la práctica en el plazo señalado anteriormente tendrán que presentarse a una sesión de defensa de la práctica para poder aprobarla. Los horarios para la defensa de la práctica se comunicarán debidamente a los interesados.
- El código fuente deberá entregarse impreso en papel y en formato electrónico, en un disco debidamente etiquetado con el nombre del alumno. Éste será el disco que se utilizará para las pruebas de funcionamiento en el momento de la defensa de la práctica.

7.3.- La defensa de la práctica

La defensa de la práctica consistirá en:

- Compilación del programa
- Prueba de funcionamiento
- Cuestiones sobre la teoría incluida en el boletín o sobre la implementación