

PARTE I
SERVICIOS DE RED

1. TECNOLOGÍAS WEB BÁSICAS

En sus orígenes, la tecnología Web, basada en el protocolo HTTP para la transmisión de hipertexto sobre protocolos de comunicaciones TCP/IP, sirvió para proporcionar una nueva visión de Internet. Si antes, con servicios como FTP, la Red se concebía como un conjunto de nodos servidores cuya topología y estructura física había que conocer, ahora se logra un nivel de abstracción mayor: lo que importa es la información —*los contenidos* en términos del WWW— en lugar de su ubicación, de las características de los servidores en los que está ubicada o de los dispositivos empleados para acceder a la misma: computador personal, PDA, WebTV, teléfono móvil, etc.

Estas características de independencia entre clientes y servidores, unida al auge de la generación de contenidos en Internet y la enorme heterogeneidad reinante en el entorno de las TIC, facilitan que prospere rápidamente este nuevo modelo cliente/servidor.

Sin embargo, debido en parte a su paulatina participación en el mundo de los negocios y a la necesidad, por tanto, de generación de contenidos dinámicos, cada vez son más las tecnologías que en los últimos tiempos han venido a acompañar a los tradicionales *servidores Web* y, porqué no, a *engordar* nuestros *navegadores Web*. Está claro que se necesita una mejor gestión de recursos y una mayor organización de las tecnologías implicadas.

Atrás quedaron los tiempos en que la Web era un mero escaparate de consulta de información estática, dónde las empresas presentaban su catálogo de productos o su oferta de servicios. En la actualidad, se tiende progresivamente a hacer efectivos los diferentes servicios que ya se prestan por otros medios, en esa red universal que es Internet. Para ello, las aplicaciones convencionales cliente/servidor para redes locales o corporativas, escritas pensando en un entorno determinado, deberán ser

adaptadas a las nuevas características e idiosincrasia que imponen las tecnologías de Internet en general y de la Web en particular.

Durante los próximos cuatro capítulos revisaremos su evolución: partiendo en este capítulo del concepto de HTTP básico y de aplicación Web, en los capítulos 2 y 3 analizaremos las tecnologías más importantes que se han ido incorporando tanto en la parte del cliente —DTML, *ActiveX*, *applets java*, *plug-ins*— como en la del servidor —*páginas activas*, componentes en el servidor, contenedores de componentes; finalmente, en el capítulo 4 veremos cómo, de alguna manera, tanto por la complejidad que están adquiriendo las actuales aplicaciones como por la sobrecarga que todas estas tecnologías infligen sobre los servidores Web convencionales, en la actualidad, la tendencia es volver a aligerar a nuestro servidor web, dejándolo que se ocupe de aquello para lo que fue concebido —la gestión del nivel de presentación o interfaz con el usuario final— y organizar un sistema paralelo complementario capaz de dar soporte adecuado a las aplicaciones.

1.1. MODELO CLIENTE-SERVIDOR

El Modelo Cliente-Servidor es uno de los más empleados por un equipo o aplicación (cliente) para acceder a recursos ubicados en otro equipo (servidor) u ofrecido por otra aplicación (servicio).

En la figura 1.1 se muestra un diagrama en el que se destacan los principales elementos que intervienen en un entorno preparado para emplear el modelo cliente-servidor a través de Internet.

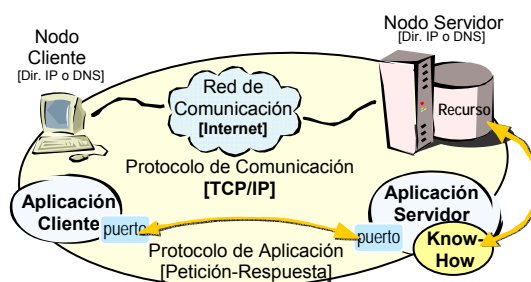


Figura 1.1. Principales elementos del modelo Cliente/Servidor sobre TCP/IP.

El primer elemento a destacar es el *nodo cliente*. El nodo cliente está formado por el equipo computacional —ya sea un PC convencional, un

portátil, una agenda electrónica o un teléfono móvil— con su respectivo sistema operativo y con capacidad para conectarse a una red de computadores. Cada nodo cliente debe disponer de, al menos, una aplicación que denominaremos *aplicación cliente*. Esta aplicación es la responsable de solicitar el recurso o servicio deseado y, en caso de ser interactiva, actuar como interfaz con el usuario final. En la práctica, tanto el nodo cliente como la aplicación cliente se suelen denominar, sencillamente, clientes.

Al otro lado del escenario nos encontramos el *nodo servidor* o *servidor*. En este caso, se trata del equipo —la combinación de hardware más sistema operativo— que posee el *recurso* hardware o software objeto del servicio. El servidor deberá estar conectado de alguna forma a la misma red que el cliente. Sobre él se ejecutará la *aplicación servidora* o *servicio* capaz de atender las solicitudes del cliente y mediante su *know-how* —su *saber cómo*— para acceder y gestionar dicho recurso.

Entre ambos nodos encontramos la *red de comunicación*. Sea cual sea su tecnología física, deberá emplear como protocolo de comunicación (niveles de red y transporte), la *pila TCP/IP*, es decir, el conjunto de protocolos empleado por *Internet*.

En este modelo cliente-servidor resulta imprescindible que ambas partes tengan alguna forma de referenciarse. Un cliente debe poder invocar a un servidor, y este servidor tendrá que ser capaz de devolverle su solicitud. Puesto que la red (lógica) en la que nos basamos es Internet, el mecanismo de identificación que cada nodo conectado a la misma debe poseer —ya sea cliente o servidor—, es lo que denominamos una *dirección IP*. Sin embargo, una dirección IP está compuesta por un número de cuatro bytes que resulta muy incómodo de recordar y manipular por los usuarios humanos. Por esta razón, lo más común es emplear un sistema de traducción de direcciones IP a un sistema de nombres jerárquicos denominando DNS (*Domain Name Server*), en el que el nombre del nodo se expresa mediante una combinación de *nombre* más el *dominio* al que pertenece. La dirección así expresada se denomina *nombre DNS*.

Aunque una dirección IP o un nombre DNS permiten identificar el nodo de red de manera unívoca, todavía falta identificar la aplicación concreta, de entre todas las que se están ejecutando sobre cada nodo, con la que nos queremos comunicar. En este caso, el protocolo TCP/IP proporciona como mecanismo un *número entero* diferente a cada aplicación en ejecución que desee acceder a la red. Este número se denomina *puerto*. Por lo tanto, el par *direcciónIP:puerto* o *nombreDNS:puerto* será el que realmente identifique cliente y servicio.

Finalmente, aunque los nodos de red cliente y servidor se comunican mediante el protocolo de comunicaciones TCP/IP que les permite entenderse a nivel de red y de transporte de datos, las aplicaciones cliente y servidor también deben poseer un mecanismo que les permita hablar entre ellas, es lo que denominamos *protocolo de aplicación*.

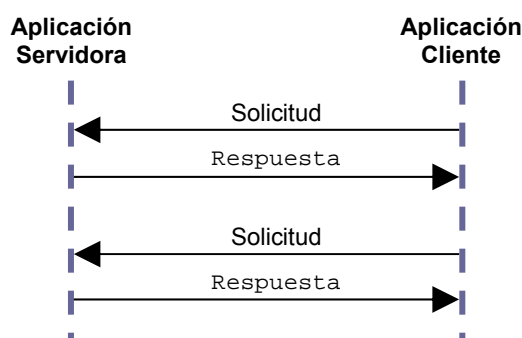


Figura 1.2. Dos secuencias Solicitud-Respuesta genéricas.

En el caso de este tipo de aplicaciones, este protocolo suele ser del tipo *petición-respuesta* (ver figura 1.2). Son protocolos en los que la *respuesta* del servidor a la *solicitud* del cliente se entiende como el justificante de haber sido recibida. Esto permite disminuir el tráfico de red provocado por el envío de justificantes y que, en muchas ocasiones, son el motivo del colapso de los sistemas de comunicación. En la mayor parte de los casos, el protocolo de aplicación es el que proporciona su nombre al servicio: FTP, HTTP, *Telnet*, etc.

1.2. SERVICIO HTTP

El servicio HTTP es un servicio basado en el modelo cliente-servidor sobre Internet, donde el cliente es un *navegador Web* y el servidor es un *servidor Web* (ver figura 1.3), utilizando ambos para su entendimiento el *protocolo de aplicación HTTP* (*HyperText Transfer Protocol*). En este caso el servidor Web se configura por defecto para escuchar solicitudes en el *puerto 80*, de forma que cualquier cliente pueda encontrarlo fácilmente. El recurso que se ofrece lo constituyen documentos de texto denominados *páginas HTML* (*HyperText Markup Language*), por ser éste el lenguaje que se emplea para su codificación (ver figura 1.3).

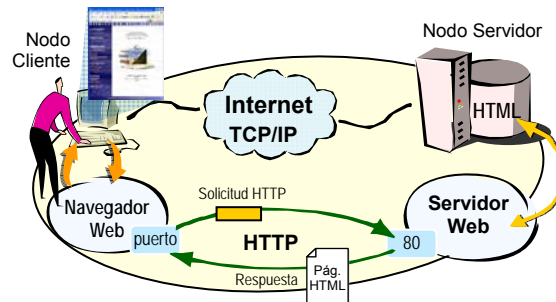


Figura 1.3. Elementos del servicio HTTP.

En la figura 1.4 se puede apreciar un ejemplo de solicitud de página HTML utilizando el protocolo HTTP. La respuesta será la propia página solicitada.

Método	Recurso	Protocolo	Cabecera
GET	/index.html	HTTP/1.1	

Figura 1.4. Ejemplo de una *Solicitud* HTTP.

1.2.1. Protocolo de aplicación HTTP

Se trata de un protocolo de aplicación, basado en texto y del tipo solicitud-respuesta. El formato de una solicitud HTTP responde a la siguiente sintaxis:

```
<Método HTTP> <URI> <Protocolo>
<Cabecera>
<Línea en blanco>
[<Cuerpo>]
```

Donde:

- El *Método HTTP* indica la acción que se solicita al servidor. Alguno de los principales métodos HTTP son los siguientes:
 - HEAD Solicita la cabecera del recurso referenciado.
 - GET Solicita un recurso mediante una URI.
 - POST Solicita un recurso y pasa información adicional en el *cuerpo* de la solicitud.

PUT Solicita que el servidor almacene la información enviada con el nombre indicado en la URI

- La *URI (Uniform Resource Identifier)* es la forma normalizada de referenciar un recurso. La diferencia con una *URL (Universal Resource Locator)* es que en el último caso, además de la URI, también se incluye el tipo de servicio al que se quiere acceder, la dirección del nodo servidor y el puerto en el que el servicio espera las solicitudes.
- El *Protocolo* indica el nombre y versión del protocolo que espera el cliente.
- La *Cabecera* de la solicitud sirve para indicar diferentes parámetros que modifican la forma en la que se realiza la solicitud o la respuesta. La cabecera finaliza con una línea en blanco. El formato de cada parámetro es el siguiente:
`<etiqueta>: <valor>\r\n`
- El *Cuerpo* de la solicitud es opcional y permite enviar información adicional junto a la solicitud.

En la figura 1.5 se presenta un sencillo ejemplo en el que se solicita al servidor el archivo `miAplicacion.cgi` situado en la carpeta `/cgi`, también se le indica mediante la cabecera algunas condiciones —como que se acepta cualquier tipo de datos, que no cierre la sesión inmediatamente y que nuestro navegador es del tipo genérico—. Además, se emplea el cuerpo de la solicitud para pasar información adicional (que se detallará en los próximos apartados) al servidor Web.

Solicitud	POST /cgi/miAplicacion.cgi HTTP/1.0
Cabecera	Accept: */* Connection: Keep-Alive User-Agent: Generic [línea en blanco]
Cuerpo	Nombre=Paco&eMail=pmacia@dtic.ua.es

Figura 1.5. Ejemplo de solicitud HTTP.

Así mismo, la Respuesta también sigue un formato establecido que se puede sintetizar en la siguiente sintaxis:

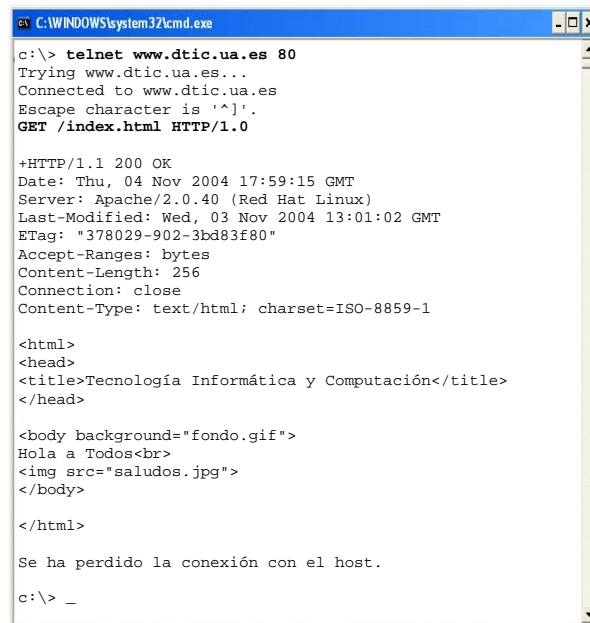
```
<Línea de estado>
<Cabecera>
<Línea en blanco>
[<Cuerpo>]
```

Donde:

- La *Línea de estado* comienza con un símbolo más (+) si todo ha ido bien o con un símbolo menos (-) si se ha producido algún incidente. A continuación se especifica el protocolo que emplea el servidor (por ejemplo: HTTP/1.0), el código de estado informando sobre el desarrollo, negativo o positivo, de la solicitud y, finalmente, una descripción textual del mismo: si ha sido exitosa, si se ha producido un error o, sencillamente, con información de carácter más general.
+|-protocolo código texto
- La *Cabecera* de la respuesta sirve para indicar diferente información sobre el servidor y sobre el objeto solicitado: la fecha, el nombre del servidor, el tipo MIME del objeto, etc. El formato de cada parámetro es el siguiente:
<etiqueta>: <valor>
- El *Cuerpo* es la parte más importante de la respuesta, pues está formado por el contenido del recurso web solicitado, generalmente

una página en formato HTML o un archivo en formato MIME (ambos se describirán más adelante).

En la figura 1.6 se presenta un ejemplo de conexión remota mediante un sencillo cliente *telnet*. En primer lugar se realiza una conexión indicando el nombre DNS del servidor y el puerto en el que se encuentra escuchando (puerto 80). Una vez establecida la conexión, el servidor Web queda a la espera de una solicitud que siga el protocolo HTTP. En el caso del ejemplo se trata de una solicitud para recuperar el archivo `index.html` que se debe encontrar en el directorio raíz del servidor Web. El servidor Web responde indicando con un símbolo más (+) que todo ha ido bien, informa sobre el protocolo con el que trabaja (HTTP/1.1), el código de estado (200) y una breve descripción que indica que todo ha ido bien (OK). A continuación envía la cabecera de la respuesta con algunos datos sobre la conexión y la información a transmitir (fecha, servidor, última modificación del archivo, una marca, el tamaño de los datos y del archivo, el estado de la conexión y el tipo MIME del archivo). Finalmente, tras una línea en blanco que señala el final de la cabecera, utiliza el cuerpo de la respuesta para transmitir el contenido del archivo de texto, en formato HTML, que se le había solicitado. Una vez enviada esta información y puesto que HTTP es un protocolo de tipo petición-respuesta, el servidor da por finalizada la conexión, es decir, si deseamos un nuevo recurso de este servidor, tendremos que realizar una nueva conexión siguiendo nuevamente todos los pasos anteriormente detallados. Por supuesto, un cliente *telnet* no sabe interpretar las etiquetas de formato que ha recibido, por lo que mostrará el contenido del archivo de forma literal.



```

c:\> telnet www.dtic.ua.es 80
Trying www.dtic.ua.es...
Connected to www.dtic.ua.es
Escape character is '^]'.
GET /index.html HTTP/1.0

+HTTP/1.1 200 OK
Date: Thu, 04 Nov 2004 17:59:15 GMT
Server: Apache/2.0.40 (Red Hat Linux)
Last-Modified: Wed, 03 Nov 2004 13:01:02 GMT
ETag: "378029-902-3bd83f80"
Accept-Ranges: bytes
Content-Length: 256
Connection: close
Content-Type: text/html; charset=ISO-8859-1

<html>
<head>
<title>Tecnología Informática y Computación</title>
</head>

<body background="fondo.gif">
Hola a Todos<br>

</body>

</html>

Se ha perdido la conexión con el host.
c:\> _

```

Figura 1.6. Ejemplo de una sesión remota con una solicitud-respuesta HTTP.

En principio, un servidor Web *elemental* sólo está preparado para servir información estática, normalmente en forma de páginas HTML que el navegador Web del cliente se encargará de interpretar y representar, generalmente, de forma gráfica en su monitor. En el ejemplo de la figura 1.6 se había solicitado una sencilla página HTML que implementa el típico caso de ejemplo «Saludos». El contenido del archivo `/index.html` con dicho contenido podría ser el mostrado en la figura 1.7.

```

<HTML>
  <HEAD>
    <TITLE>Saludos</TITLE>
  </HEAD>

  <BODY background="fondo.gif">
    Hola a todos <BR>
    <IMG src="saludos.jpg">
  </BODY>
</HTML>

```

Figura 1.7. Listado del archivo `index.html` con un sencillo documento HTML.

Un documento HTML, tal y como sus siglas indican, está formado por los siguientes elementos:

- Un documento de texto.
- Etiquetas de marcado o de formato:
`<etiqueta [arg1 arg2 ...]> ... [</etiqueta>]`
- Referencias a otros recursos:
` texto descriptivo `

Aunque las páginas o archivos en formato HTML son los recursos más básicos de un servicio Web, pueden requerir otros recursos adicionales para que el navegador Web pueda mostrar todo su contenido (en el ejemplo de la figura 1.7, tras el mensaje «*Hola a todos*», se presentará una imagen en formato *JPEG* (*Joint Photographic Experts Group*), ubicada en el servidor, con el nombre de `saludos.jpg`. Por supuesto, el navegador Web tendrá que recuperar este archivo mediante una nueva solicitud al servidor Web. El problema es que estos recursos suelen ser archivos gráficos, de vídeo o de sonido, que no se encuentran en formato de texto, que es para lo único que está preparado el protocolo de texto HTTP. Para subsanar esta deficiencia, cada archivo binario deberá ser convertido previamente a un archivo de texto mediante un mecanismo de representación externa de datos denominado *MIME* (*Multipurpose Internet Mail Extension*). La figura 1.8 muestra el contenido de un archivo MIME correspondiente a la conversión a texto de una imagen binaria en formato JPEG. Este mecanismo fue originalmente concebido para poder adjuntar archivos en formato binario a los mensajes de correo electrónico mediante el servicio SMTE (que se estudiará en el capítulo 6).

1.2.2. Arquitectura HTTP

Podemos considerar que la arquitectura técnica, física, de un sistema distribuido para la prestación del servicio HTTP tiene como fundamento una red de comunicaciones. Sobre ésta, (a la derecha en la figura 1.9), encontramos la estructura del cliente. Subiendo por las diferentes capas de dicha estructura hallamos el hardware del nodo cliente en el que destaca la tarjeta adaptadora de red (NIC) que le permite acceder al canal de comunicación. A continuación encontramos el sistema operativo que gestiona el nodo y en el que se ubica la pila TCP/IP que implementa los protocolos de comunicaciones. Finalmente llegamos a la aplicación cliente que, en este caso, es un navegador Web. Como ya se ha podido estudiar en el apartado anterior, el navegador Web tendrá capacidad para comunicarse con el servidor mediante el protocolo HTTP y para interpretar los archivos

```
Content-Type: image/jpeg; name="menu.jpg"  
Content-Transfer-Encoding: base64  
  
/9j/4AAQSkZJRgABAQEAYABGAAD/2wBDAAAgBGbcGBqGHBwcJCQCgKDBQNDA  
LDBkSEW8UHRofHh0aHBwgJc4nICIsIwxCKDcpLDaxNDQ0Hyc5PTgyPC4zNDL/  
2wBDAQKjCQwLDBgNDRgyIRwhmJlYmJlYmJlYmJlYmJlYmJlYmJlYmJlYmJlY  
MjYmJlYmJlYmJlYmJlYmJlYmJlYmJlYmJlYmJlYmJlYmJlYmJlYmJlYmJlY  
MjYmJlYmJlYmJlYmJlYmJlYmJlYmJlYmJlYmJlYmJlYmJlYmJlYmJlYmJlY  
MjYmJlYmJlYmJlYmJlYmJlYmJlYmJlYmJlYmJlYmJlYmJlYmJlYmJlYmJlY  
HwAAAQUBAQEBAQEAAAAAAAAAAAECAwQFBgcICQoL/8QAtrRAAAGEDAwIEAwUF  
BAQAAAF9AQIDAAQRBRIhMUEGE1FhByJxFDKBkaEII0KxwRVS0fAkM2JyggkK  
FhcYGRolJicoKSo0NTY3ODk6Q0RFRkdISUpTVFVFWVhlZWmnKZWZnaGlqc3Rl  
dnd4eXkgZGIWgh4iUjPktIJWW15iZmqGjpKWmp6ipqrKztLW2t7i5usLDxMXG  
x8jJytLTlNXWl9jZ2uHi4+Tl5uf06erx8vP09fb3+Pn6/8QAHWEEAAWEBAQEB  
AQEBAQAAAAAAAECAwQFBgcICQoL/8QAtrREAAgECBAQBACFBAQAAQJ3AAEC  
AxEEBSExBhJBUDqhchrMiMoEIFEKRobHBcSMzUvAVYNLRChYkNOEL8rcYGrom  
JygpkJU2NZg5OkNERUZHSelKULRVVLjYWVpjZGVmZ2hpand0dxZ3eh16goOE  
hyahiImKkpOUlZaXmJmaoqOKpaanqKmqsro0tba3uLm6wsPExcBHyMnK0tPU  
ldbX2Nna4uPk5ebn6Onq8vP09fb3+Pn6/9oADAMBAAIRAxEAPwDk2t7crM/m  
+QhjL28e4SFvnC7WIxggbjkgdOonzCoWokeIlQquCRnlHJOtk9eccYHA75Jd  
btY2dlfcwPtmicSRtgHawOrwfCVFge/514D9m3yyk7dnO9+l/Tdvr5H7RCml  
r+vy7H//2Q==  
  
---8CFDA75A284D5A8033E016C87CBCE897---
```

Listado 1.8. Contenido de una imagen en formato binario JPEG recodificada en formato de texto MIME dentro de un documento HTML.

Si seguimos analizando esta arquitectura técnica podemos observar la estructura del servidor Web que, desde el punto de vista de los elementos que la conforman y de su organización, no difiere en gran medida de la estructura del navegador web. Como se puede apreciar en la figura 1.9, el servidor está compuesto por un hardware conectado a la red mediante su NIC, un sistema operativo con su propia pila TCP/IP —quizá todo ello mucho más optimizado para soportar mayor carga en las transacciones de red y en el acceso al sistema de almacenamiento masivo— y una aplicación servidora responsable de gestionar el servicio Web. Dicha aplicación también debe entender HTTP, ser capaz de codificar los archivos binarios en formato MIME y, lo que realmente la diferencia de la aplicación cliente, el núcleo principal está compuesto por el gestor de transacciones y por el acceso al sistema de archivos en el que se encuentran las páginas HTML y demás recursos asociados.

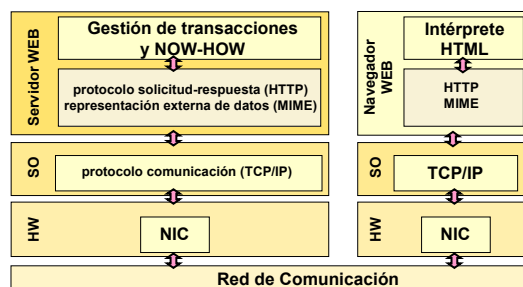


Figura 1.9. Arquitectura técnica física de una aplicación Web sobre Internet.

1.3. INTERFAZ CGI Y APLICACIONES WEB

Básicamente, una *aplicación Web* es cualquier aplicación basada en la arquitectura cliente-servidor, donde el cliente es un *navegador Web* y el servidor es un *servidor Web* (ver figura 1.3), utilizando ambos para su entendimiento el *protocolo de aplicación HTTP (HyperText Transfer Protocol)*. Sin embargo, con las características del servidor Web que hemos estudiado hasta el momento, el servicio HTTP tan solo es capaz de proporcionar una serie de páginas estáticas, previamente definidas, que difícilmente servirían para alcanzar un objetivo tan ambicioso.

Un requisito mínimo para poder construir una aplicación a partir del servicio HTTP es contar con la capacidad de generar páginas HTML de forma dinámica en función de determinadas condiciones establecidas por el cliente. Por suerte, el servicio HTTP básico cuenta con una característica adicional que permite hacer esto: un servidor Web puede invocar una aplicación externa a petición del cliente, pasarle una serie de parámetros que éste haya indicado y, tras la ejecución de dicha aplicación, recoger su salida para, seguidamente, retransmitírsela al cliente.

Estas aplicaciones se ejecutan en el nodo servidor y pueden estar desarrolladas en cualquier lenguaje de programación, compilado o interpretado, soportado por dicho servidor. La única restricción que se les impone es que implementen un mecanismo que les permita comunicarse con el servicio Web; concretamente: acceder a los parámetros de ejecución —en caso de que éstos existan— determinados por el cliente y que su salida —en formato textual y preferiblemente en formato HTML— pueda ser recuperada por el servidor Web para pasársela al cliente que invocó la aplicación. Este mecanismo está perfectamente definido para que ambas partes la conozcan de antemano y se denomina *CGI (Common Gateway*

Interface); razón por la cual, este tipo de aplicaciones se denominan también *aplicaciones CGI*.

En la figura 1.10 se presenta un sencillo ejemplo de solicitud HTTP GET en la que un cliente solicitaría al servidor la ejecución de la aplicación `busca.php` ubicada en la carpeta `/cgi-bin` y le pasa como argumento un parámetro identificado como `clave`, cuyo valor será `pdi`.

Método	URI	Cabecera
GET	/cgi-bin/busca.php?clave=pdi	

Figura 1.10. Formato de invocación de una aplicación CGI.

Otro ejemplo de solicitud de ejecución de una aplicación CGI, pero esta vez empleando el método POST, es el mostrado en la figura 1.11. En él puede observarse cómo los parámetros y sus valores, se transfieren ahora en el cuerpo de la solicitud.

Solicitud	POST /cgi/miAplicacion.cgi HTTP/1.0
Cabecera	Accept: /*/* Connection: Keep-Alive User-Agent: Generic [línea en blanco]
Cuerpo	Nombre=Paco&eMail=pmacia@dtic.ua.es

Figura 1.11. Ejemplo de solicitud HTTP utilizando el método POST.

En la figura 1.12 se muestra un sencillo escenario en el que un cliente invoca la ejecución de una aplicación CGI. Su funcionamiento, siguiendo el procedimiento que define la interfaz CGI es el siguiente:

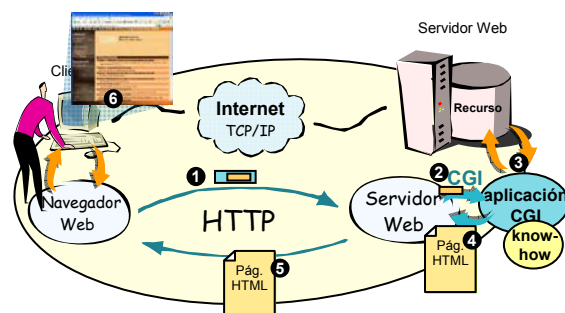


Figura 1.12. Escenario de desarrollo de un servicio HTTP mediante CGI.

- 1 Cliente o navegador Web a Servidor Web. El usuario, a través de su navegador Web solicita la ejecución de una aplicación CGI al servidor. Esta solicitud puede realizarse mediante el método GET o el método POST. En el caso de una solicitud GET, los parámetros y sus valores se incluyen en la propia URI (ver figura 1.10). Con el método POST, estos mismos datos se incluyen en el cuerpo de la solicitud (ver figura 1.11).
- 2 Servidor Web a Aplicación invocada. El Servidor Web copia determinados parámetros en variables de entorno (ver figura 1.13) —en Win-CGI éstas se guardan en ficheros `.INI`— y ejecuta la aplicación. Si el método de invocación era POST, además, escribe los datos del cuerpo de la solicitud (es decir, los parámetros y sus valores) en el descriptor de archivo que se utiliza como entrada estándar de la aplicación.
- 3 La aplicación puede desempeñar cualquier función —como por ejemplo: realizar un cálculo o acceder a una base de datos— y tomar decisiones antes de formatear la información en HTML, para dejársela al servidor Web, el cual la enviará al cliente o navegador. Podemos considerar que el *know-how* ha pasado del servidor Web a la aplicación CGI.
- 4 La aplicación, una vez realizada su labor (o a medida que la realiza), escribe en su salida estándar el resultado, procurando hacerlo en un formato válido para el cliente Web: texto en formato HTML, MIME o XML, por poner algunos ejemplos. Precisamente, el servidor Web tendrá

redirigida esta salida a un descriptor propio, a través del cuál leerá este resultado.

5 El servidor Web enviará la información al cliente Web que realizó la invocación a modo de respuesta HTTP, conformando el cuerpo de la misma a partir de los datos obtenidos en el paso anterior.

6 Finalmente, el navegador Web, una vez recibida la respuesta, la interpretará y la mostrará (posiblemente tras recuperar mediante otras solicitudes otros recursos: gráficos, audio, etc.) al usuario de forma gráfica en su pantalla.

Variable de Entorno	Valor
REQUEST_METHOD	GET
QUERY_STRING	/cgi-bin/busca.php?clave=pdi
CONTENT_LENGTH	29
SCRIPT_NAME	/cgi-bin/busca.php
SERVER_PORT	80

Figura 1.13. Contenido de algunas variables de entorno utilizadas por el protocolo CGI.

La propuesta CGI aporta una capacidad de respuesta dinámica al servidor Web, de modo que las respuestas ya no van a ser meras páginas HTML estáticas, sino que pueden estar en «*función de*». Proporciona, además, una razonable libertad de elección del lenguaje de desarrollo de la aplicación CGI, que puede realizarse mediante programación en lenguajes nativos de la plataforma y a los que probablemente estarán acostumbrados los programadores de la compañía, con lo que la curva de aprendizaje es mínima.

Como desventajas, hay que mencionar que con esta tecnología no hay relación entre el programa CGI y el servidor Web, pues ambos se ejecutan en procesos separados. Por tanto, no podemos controlar al programa CGI, es decir, no sabemos si terminó con éxito, si «*se colgó*» o si devolvió un resultado inesperado. Asimismo, como se instancia un programa CGI con cada petición que se realiza al servidor web, incluso aunque sea el mismo cliente el que la efectúe, el rendimiento no es el más óptimo y, además, se sobrecargan los recursos del nodo servidor. Finalmente, con CGI existe poca o nula portabilidad entre plataformas distintas, al escribirse el programa en un lenguaje nativo. Si el CGI se

realiza en lenguaje interpretado, por ejemplo tipo Perl o PHP, aumenta la portabilidad —sólo habría que tener el intérprete correspondiente en la otra máquina— a costa de reducir el rendimiento con respecto a un lenguaje compilado.

Analicemos algunos sencillos ejemplos que nos ayuden a comprender mejor el funcionamiento de los CGIs.

En primer lugar, el listado que se muestra en la figura 1.14 corresponde a una aplicación mínima creada mediante un guión de comandos (*script*) que podrá ejecutar casi cualquier intérprete de comandos (*shell*) de Unix. En este ejemplo, el proceso escribirá en su salida estándar el código correspondiente a una página HTML que es equivalente al ejemplo de *saludo* mostrado con anterioridad en la figura 1.7. Si un cliente invoca este script mediante una solicitud HTTP del tipo: GET /cgi/saludos.sh HTTP/1.1, el servidor Web se encargará de que dicho guión se ejecute y recogería el resultado de su salida estándar para devolvérselo al usuario. El resultado final para este usuario coincidirá con el del ejemplo 1.7, sólo que en esta ocasión, la página no reside físicamente en un sistema de archivos, en realidad se ha generado «*al vuelo*» (o de forma dinámica).

```
#!/usr/local/sh
# Comenzamos indicando quién debe interpretar el script.

# Sencillo script de shell que genera por la salida
# estándar una página HTML de saludo.

# Generamos la cabecera de la respuesta HTTP
print "Content-type: text/html"
print

# Generamos el cuerpo de la respuesta HTTP
print "<HTML>"
print "<HEAD>"
print "<TITLE>SALUDOS</TITLE>"
print "</HEAD>"
print "<BODY>"
print "<H1>¡Hola a Todos!</H1>"
print "</BODY>"
print "</HTML>"
```

Figura 1.14. Guión de comandos (*script de shell*) denominado `saludos.sh`, compatible con la mayoría de *shells* UNIX (`ksh`, `bsh`, `bash`), pensado para actuar como un programa CGI.

Otro ejemplo equivalente al anterior es el presentado en el listado de la figura 1.15. En este caso, en lugar de un *script*, se ha optado por una versión escrita en *lenguaje C* que se deberá compilar para generar el correspondiente binario ejecutable. El resultado de invocar este binario por parte de un cliente será, nuevamente, análogo al de ejecutar el anterior *script* (ejemplo 1.14) o invocar la página HTML del ejemplo 1.7.

```
/* Programa C utilizado como programa CGI *
 * generando como salida una página html *
 * con el mensaje "Hola a todos" */

#include <stdio.h>

main(int argc, char *argv[]) {

    printf("Content-type: text/html\n\n");
    printf("<HTML>\n");
    printf("<HEAD>\n");
    printf("<TITLE>SALUDOS</TITLE>\n");
    printf("</HEAD>\n");
    printf("<BODY>\n");
    printf("<H1>;Hola a Todos!</H1>\n");
    printf("</BODY>\n");
    printf("</HTML>\n");

} /* de main */
```

Listado 1.15. Contenido del archivo `saludos.c` con la versión en lenguaje C de la aplicación CGI "Hola a todos".

Un ejemplo un poco más «*interactivo*» es el que se presenta en la figura 1.16. En este caso, vamos a determinar qué método de invocación se ha empleado leyendo la variable de entorno `REQUEST_METHOD`. En caso de haber empleado el método GET, los argumentos se encontrarán en la variable de entorno `QUERY_STRING`. Si, por el contrario, se ha empleado el método POST, el servidor Web nos habrá dejado estos argumentos en la entrada estándar. En cualquier caso, el tamaño de la cadena con los argumentos lo tendremos reflejado en la variable de entorno `CONTENT_LENGTH`.

Como resultado, este programa generará de forma dinámica el código HTML de una página en la que se mostrará cuales fueron los argumentos de la invocación.

```

/* Aplicación C escrita para ser ejecutada como      *
 * programa CGI, pensada para obtener los argumentos *
 * pasados por el navegador Web, ya sea a través de *
 * una invocación mediante el método GET, como el POST */

int main (void) {
    char metodo[100];
    char argumentos[1000];
    int tam;

    /* Obtenemos el método */
    strcpy(metodo, getenv("REQUEST_METHOD"));
    if (strcmp(metodo, "GET") = 0) {
        /* Argumentos en variable de entorno QUERY_STRING */
        strcpy(argumentos, getenv("QUERY_STRING"));
    }
    else {
        if (strcmp(metodo, "POST") = 0) {
            /* Argumentos en la entrada estándar */
            tam = atoi(getenv("CONTENT_LENGTH"));
            fgets(argumentos, tam + 1, stdin);
        }
        else
            printf("Error. Metodo no soportado");
    }

    /* Cabecera */
    printf("Content-type: text/html\n");
    printf("\n");

    /* Cuerpo */
    printf("<HTML>\n");
    printf("<HEAD>\n");
    printf("<TITLE>Argumentos de entrada</TITLE>\n");
    printf("<BODY>\n");
    printf("Argumentos del CGI: %s\n", argumentos);
    printf("</BODY>\n");
    printf("</HTML>\n");
}

```

Listado 1.19. Ejemplo en lenguaje C del tratamiento de las peticiones HTTP GET y POST desde una aplicación CGI.

1.3.1. Formularios Web

Tal y como ya se ha explicado, un cliente puede invocar una aplicación CGI mediante una solicitud HTML a través del método GET o POST. Sin embargo, si cada vez que un usuario desea invocar una función, debe

construir a mano una de estas solicitudes, todo el sistema iría en contra de su filosofía global de transparencia y facilidad de uso de cara al usuario. Además, de forma directa, un usuario sólo podría generar solicitudes del tipo GET en la que los parámetros y valores se pasan directamente en la línea de solicitud.

Una primera alternativa consistiría en generar URIs de forma manual e incluirlas mediante etiquetas HREF. El problema es que mediante este mecanismo no se pueden modificar los valores de los parámetros de las llamadas y, por lo tanto, se convierte en una solución parcial, adecuada sólo para casos muy concretos.

Por suerte, HTML prevé esta necesidad y proporciona una serie de etiquetas que permiten definir diferentes campos para la introducción de datos dentro de nuestra página, recoger la información introducida por el usuario en los mismos y generar automáticamente la solicitud HTTP adecuada para invocar una aplicación CGI con los valores introducidos por el usuario. Este tipo de páginas HTML que incluyen esta posibilidad se denominan *formularios HTML* y las etiquetas que lo hacen posible son: `<FORM>`, `</FORM>` e `<INPUT>`.

La etiqueta FORM es la que determina el alcance del formulario, permite definir el método de invocación a utilizar y la aplicación CGI a la que se hará referencia.

Las etiquetas INPUT definen los diferentes campos y botones que presentará el formulario para la introducción de información. Se pueden definir campos de tipo texto, desplegables o de selección. También se pueden definir botones genéricos y, sobre todo, se puede definir un tipo de botón especial denominado `submit`. Este botón es uno de los más importantes del formulario pues, cuando el usuario pulsa sobre él, desencadena que el navegador Web dé por concluida la introducción de datos, los recoja y genere con ellos la solicitud GET o POST adecuada para el servidor Web.

Según esto, podríamos definir un *formulario Web* como una página Web que incluye una serie de botones y campos para la recogida de datos junto con un botón para realizar la invocación.

La figura 1.20 muestra el código HTML de una página Web que incluye un sencillo formulario de búsqueda con un campo de introducción de texto y un botón para comenzar con la misma. En la figura 1.21 se puede observar cómo presentaría un navegador Web convencional este formulario. En este ejemplo, al pulsar sobre el botón «buscar» que es del tipo especial «`submit`», el navegador generaría una solicitud similar a:

```
GET /cgi-bin/busca.cgi?Nombre=Paco&Maciá
```

```
<HTML>
<HEAD>
<TITLE>Búsqueda...</TITLE>
</HEAD>

<BODY>
<H1>Formulario de búsqueda</H1>
<HR>

<FORM method="GET" action="buscar.cgi">
  Introduzca un nombre:<br>
  <INPUT NAME="Nombre"><P>
  <INPUT type="submit" value="buscar"><P>
</FORM>

<HR>
</BODY>
</HTML>
```

Listado 1.20. Formulario HTML con invocación a CGI.

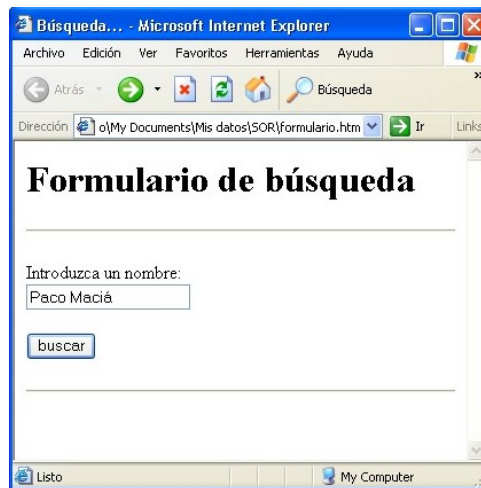


Figura 1.21. Representación gráfica por un navegador de la página HTML de la figura anterior y que incluye un formulario.

2. AMPLIACIONES EN EL CLIENTE

El navegador Web ha sido adoptado por la comunidad de usuarios de Internet como la aplicación cliente por excelencia para este entorno. No obstante, la Web se ha convertido en un medio complejo donde las aplicaciones requieren de más potencia, sin exceder los requisitos deseados por los usuarios finales (aplicaciones ligeras, mínimos requisitos de instalación y configuración, lógica de negocio, separación entre estilo y datos). Son numerosas las tecnologías que han permitido extender el navegador Web inicial para acercarlo a un contenedor en la parte del cliente que cumpla los requisitos exigidos en estas aplicaciones.

En un principio el navegador Web solamente ofrecía al cliente la posibilidad de visualizar información de tipo estática (HTML). Con la evolución de las .COM la competencia por presentar páginas Web más atractivas para el usuario y con una mayor funcionalidad han surgido diferentes tecnologías que permiten incorporar características multimedia en las páginas e interactuar con el usuario. De esta forma ha sido necesario ampliar las funcionalidades del navegador Web para que soporte todas estas características.

2.1. SCRIPTS EN EL CLIENTE

Una de las técnicas utilizadas para dotar de dinamismo las páginas Web en la parte del cliente ha sido el uso de código incrustado dentro del código HTML estático. Estos fragmentos de código son interpretados en tiempo de ejecución en el cliente por el navegador Web y se denominan *SCRIPT*. De esta forma permite dotar a la página Web de efectos y funcionalidades. Las páginas de este tipo dependen de la plataforma y más concretamente del navegador sobre el cual se ejecuten. La ventaja que pueden ofrecer es la rápida respuesta ante acciones del usuario permitiendo la descarga del servidor. No obstante, se debe contemplar el enorme número de

navegadores Web existentes y la compatibilidad con los diferentes lenguaje de tipo *Scripts*.

2.1.1. El lenguaje JavaScript

JavaScript es un lenguaje interpretado desarrollado por Netscape bajo el nombre de *LiveScript* y que tras formar la alianza SUN y Netscape pasó a tomar el nombre con el que hoy en día es conocido. El objetivo era crear un lenguaje con características similares, en lo que a sintaxis se refiere, a los lenguajes de alto nivel pero de uso sencillo, sin necesidad de utilizar compiladores. Simplemente, se introducía código JavaScript dentro de un documento HTML y el navegador se encargaba de interpretar este código.

Como se ha comentado, el código de lenguaje JavaScript se introduce dentro del código HTML. Existen cuatro formas diferentes de incrustar código Javascript.

- Entre las etiquetas `<script>` y `</script>`
- Especificando un archivo JavaScript en el código HTML (*.js).
- Especificando una expresión Javascript como valor de un atributo HTML.
- Especificando código Javascript en un evento de los controles HTML que los contemplen (ej. botón).

2.1.1.1. Etiqueta de marcado y archivos JavaScript

Para diferenciar el código HTML del código JavaScript se utiliza la etiqueta de marcado `<SCRIPT>` para indicar que comienza un fragmento de código JavaScript y `</SCRIPT>` para indicar el final.

A lo largo de un documento HTML se pueden utilizar varias veces dichas etiquetas, pero siempre que se introduzca una etiqueta de apertura se deberá introducir una de cierre.


```

<SCRIPT>
  window.alert("Hola Mundo JavaScript!!!")
</SCRIPT>
<HTML>
  <HEAD>
    <TITLE>Ejemplo JavaScript</TITLE>
  </HEAD>
  <BODY>
    Hola Mundo HTML!!!
  </BODY>
</HTML>

```

Listado 2.1. Ejemplo sencillo de introducción de javascript en un documento HTML.

En principio no existe ninguna norma que indique dónde se debe introducir las etiquetas JavaScript, aunque es importante conocer cómo el navegador interpreta el código para no obtener resultados no deseados.

Cuando un cliente realiza una petición a un servidor Web el servidor responde con un documento que puede contener HTML y código JavaScript. El navegador Web comienza a analizar el código del documento HTML y va interpretando el código, bien sea JavaScript o HTML, y procediendo a su ejecución. En el ejemplo anterior primero se mostraría en pantalla una ventana con el mensaje `Hola Mundo JavaScript!!!`. Posteriormente se procedería a mostrar el documento HTML.

```

<HTML>
  <HEAD>
    <TITLE>Ejemplo JavaScript</TITLE>
  </HEAD>
  <BODY>
    Hola Mundo HTML!!!
  </BODY>
</HTML>
<SCRIPT>
  window.alert("Hola Mundo JavaScript!!!")
</SCRIPT>

```

Listado 2.2. Ejemplo sencillo de introducción de JavaScript en un documento HTML.

En este último caso primero, se mostraría en pantalla el mensaje `Hola Mundo HTML!!!`. Posteriormente se mostraría una ventana con el mensaje `Hola Mundo JavaScript!!!`.

Para separar el código HTML del de JavaScript y poder gestionarlo y mantenerlo de una forma óptima podemos utilizar los archivos de extensión `*.js`. De esta forma tendremos archivos con el contenido JavaScript que

anteriormente se encontraba entre las etiquetas `<script>` y `</script>`. Esta forma de incrustar JavaScript nos permite, además, reutilizar funcionalidades comunes a varias páginas. Es posible combinarlo con la forma de introducir JavaScript anteriormente mencionada.

```
<HTML>
  <HEAD>
    <TITLE>Ejemplo JavaScript</TITLE>
  </HEAD>
  <BODY>
    Hola Mundo HTML!!!
  </BODY>
</HTML>
<SCRIPT SRC="saludo.js">
  ...
</SCRIPT>
```

Listado 2.3. Ejemplo sencillo de introducción de JavaScript en un documento HTML mediante archivos *.js.

```
window.alert("Hola Mundo JavaScript!!!")
```

Listado 2.4. Contenido del archivo `saludo.js`.

El atributo `SRC` puede contener una *URL* relativa o absoluta.

En estas dos formas de incrustar código JavaScript, junto con la tercera manera, el navegador en cuanto recibe el documento desde el servidor analiza el código y directamente va interpretándolo, mostrando el resultado al usuario.

2.1.1.2. Valores de atributos y eventos

Otra de las posibles maneras de incrustar código JavaScript es introduciéndola dentro de las etiquetas del código HTML, bien como valores de atributos, bien como valores de eventos de los controles.

En el primer caso el valor del atributo HTML es creado de manera dinámica de tal forma que cuando el navegador vaya a interpretar el código HTML interpretará la expresión JavaScript asociada.

```

<HTML>
  <HEAD>
    <TITLE>Ejemplo JavaScript</TITLE>
  </HEAD>
  <BODY>
    <HR WIDTH="{barWidth}%" ALIGN="LEFT" />
    Hola Mundo HTML!!!
  </BODY>
</HTML>

```

Listado 2.5. Ejemplo de código HTML dinámico mediante JavaScript.

En el segundo caso se produce como respuesta a una acción realizada por el usuario, un evento. Mediante JavaScript es posible controlar las acciones que realiza un usuario al interactuar con la página Web y de esta forma realizar una funcionalidad ante el comportamiento del usuario. Para conseguir esta segunda forma, el código JavaScript debe ser introducido en una serie de atributos del código HTML.

```

<SCRIPT>
  function saludar()
  {
    Document.alert("Hola Mundo Javascript");
  }
</SCRIPT>
<HTML>
  <HEAD>
    <TITLE>Ejemplo JavaScript</TITLE>
  </HEAD>
  <BODY>
    <input type=button value=enviar
      onclick="saludar()" />
    Hola Mundo HTML!!!
  </BODY>
</HTML>

```

Listado 2.6. Ejemplo de evento en JavaScript.

Cuando se pulsa el control de tipo botón con la leyenda `enviar` se llama al evento `onclick` del control que realiza una llamada a la función JavaScript `saludar()`.

2.2. HTML DINÁMICO: DHTML

DHTML (acrónimo de *dynamic HTML*) no es un lenguaje de *Script* como Javascript. Se trata de la unión de varias tecnologías para crear sitios Web dinámicos e interactivos. DHTML utiliza HTML estático, un lenguaje de tipo *Script* de lado del cliente (como JavaScript), un lenguaje de definición de estilos para la presentación (por ejemplo *CSS*) y el modelo de objetos de documentos (*DOM*). DHTML no es una tecnología por sí misma si no que es la unión de un conjunto de éstas.

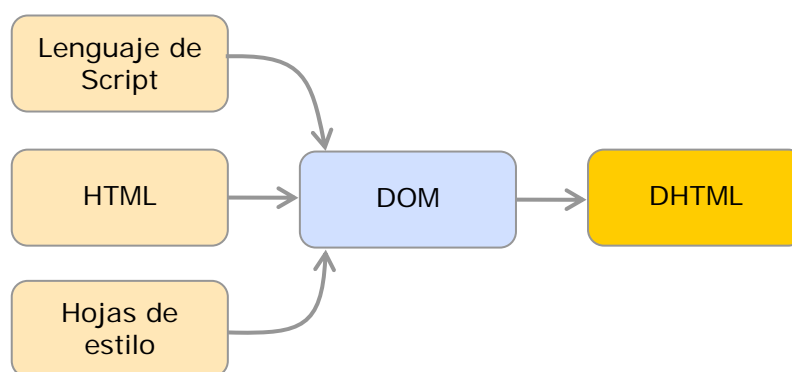


Figura 1.2. Ejemplo de solicitud HTTP

La problemática que puede surgir del uso de DHTML es que al ser una agrupación de tecnologías dependientes del navegador, se pueden encontrar problemas de interpretación en función del navegador utilizado.

2.2.1. Hojas de estilo en cascada: CSS

CSS es el acrónimo de *Cascading Style Sheets* u hojas de estilo en cascada. Se trata de un lenguaje de definición de estilo para la presentación de documentos escritos con lenguajes de marcado (tipo XML). La aparición de esta tecnología ha permitido separar los estilos de presentación de un documento Web permitiendo dar un mayor dinamismo a los sitios Web. Con este lenguaje podemos definir los colores, fuentes y otros aspectos de la presentación.

2.2.1.1. Introducción de hojas de estilo en documentos HTML

Existen tres formas de dar estilo mediante el lenguaje CSS a un documento HTML.

- Mediante un archivo externo (*.css) que se enlace al documento HTML

```
h1 {color: blue; text-align: center}
```

Listado 2.7. Ejemplo de archivo CSS.

```
<HTML>
<HEAD>
  <TITLE>Ejemplo JavaScript</TITLE>
  <LINK rel="stylesheet" type="text/css"
    href="http://www.dtic.ua.es/grupoM/presentacion.css"/>
</HEAD>
<BODY>
  <h1>Hola mundo azul</h1>
</BODY>
</HTML>
```

Listado 2.8. Ejemplo de referencia de archivo CSS en HTML.

- Mediante el uso del elemento <style> dentro del documento HTML

```
<HTML>
<HEAD>
  <TITLE>Ejemplo JavaScript</TITLE>
  <STYLE type="text/css">
    h1 {color: blue; text-align: center}
  </STYLE>
</HEAD>
<BODY>
  <h1>Hola mundo azul</h1>
</BODY>
</HTML>
```

Listado 2.9. Ejemplo de estilo mediante el uso de <style>.

- Mediante el uso del atributo style dentro de los elementos HTML

```

<HTML>
  <HEAD>
    <TITLE>Ejemplo JavaScript</TITLE>
  </HEAD>
  <BODY>
    <h1 style="color: blue">Hola mundo azul</h1>
  </BODY>
</HTML>

```

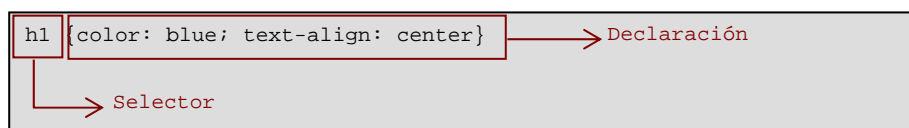
Listado 2.10. Ejemplo de estilo mediante el atributo style.

Este último caso es poco recomendable puesto que no permite una separación entre contenido y presentación.

2.2.1.2. Funcionamiento de las hojas de estilo CSS

CSS funciona a través de la declaración de reglas que establecen los estilos sobre determinados elementos. Un documento de estilos puede contener una o más reglas que se aplicarán a determinados documentos HTML o XML. Las reglas se componen de dos partes:

- Selector que identifica la regla. Normalmente se refiere a la etiqueta o elemento HTML al que queremos dar el estilo. En otras ocasiones, es un nombre lógico único el cual es especificado en el atributo class de un elemento o etiqueta HTML al que queremos darle un estilo determinado. Esta última forma de uso es utilizada cuando diferentes elementos HTML tienen el mismo estilo. Además, en el primer caso podemos establecer varios estilos para un mismo elemento a través del atributo class.
- Declaración que especifica los estilos determinados para un elemento HTML.



Listado 2.11. Declaración de estilos.

En el primer ejemplo todos los elementos del documento HTML, `h1`, establecerían el texto entre dichas etiquetas a color azul y centrado.

Si yo quiero establecer varios estilos para el elemento `h1` se debería utilizar el nombre del elemento `h1` seguido por un `'.'` y un nombre lógico

para cada regla. En el documento HTML, en cada elemento `h1` se debería utilizar el atributo `class` indicando el identificador del estilo a aplicar.

```
h1.primeros {color: blue; text-align: center}  
h1.segundo {color: red; text-align: center}
```

Listado 2.12. Ejemplo de archivo CSS.

```
<HTML>  
  <HEAD>  
    <TITLE>Ejemplo JavaScript</TITLE>  
  </HEAD>  
  <BODY>  
    <h1 class="primero">Hola mundo azul</h1>  
    <h1 class="segundo">Hola mundo rojo</h1>  
  </BODY>  
</HTML>
```

Listado 2.13. Uso de estilos en HTML.

2.2.2. El modelo de objetos de documento: DOM

El Modelo de Objetos de Documento (*Document Object Model*) define una manera de representar los elementos de un documento estructurado, tal como HTML o XML, como objetos con sus métodos y propiedades y como se accede y manipula el documento a través de sus objetos. Esta tecnología presenta una manera de acceder a los elementos de un documento de este tipo, tanto a sus elementos como a sus atributos, permitiendo añadir, eliminar o modificar estos elementos. Por lo tanto se puede definir como una interfaz de programación de aplicaciones (API) para documentos HTML o XML.

Con el Modelo de Objetos de Documento los programadores pueden construir documentos, navegar por su estructura, y añadir, modificar o eliminar elementos y contenido. Se puede acceder a cualquier cosa que se encuentre en un documento HTML o XML, y se puede modificar, eliminar o añadir usando el Modelo de Objetos de Documento, salvo algunas excepciones. En particular, aún no se ha especificado las interfaces DOM para los subconjuntos internos y externos de XML.

```
<HTML>
  <HEAD>
    <TITLE>Ejemplo</TITLE>
  </HEAD>
  <BODY>
    <TABLE>
      <TR>
        <TD>Nombre</TD>
        <TD>Apellidos</TD>
      </TR>
    </TABLE>
  </BODY>
</HTML>
```

Listado 2.14. Documento XML.

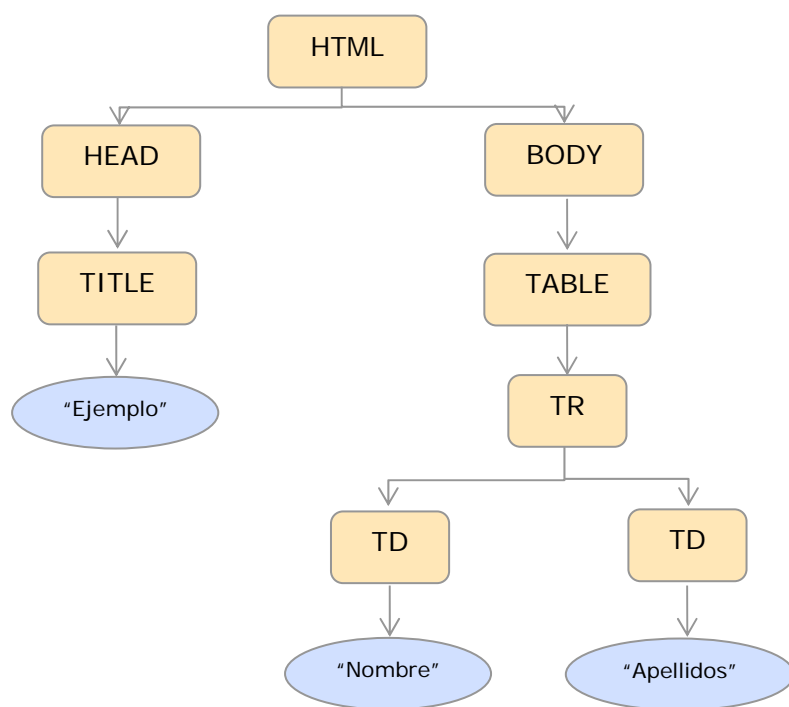


Figura 2.3. Representación estructurada del documento XML.

Los documentos DOM tienen una estructura jerárquica similar a una estructura de árbol pero no tiene porque mantener necesariamente esta organización. Puede además estar compuesto de varios documentos con este tipo de estructura.

2.3. JAVASCRIPT Y XML ASÍNCRONO: AJAX

AJAX es el acrónimo de *Asynchronous JavaScript And XML*. Al igual que DHTML no se trata de ningún lenguaje de programación. AJAX es una técnica de desarrollo de aplicaciones Web que combina un conjunto de tecnologías con el objetivo de proveer al cliente una navegación ágil y rápida y convirtiendo el navegador en un entorno muy dinámico. Entonces, ¿Cuál es la diferencia con DHTML? Básicamente, se puede decir que AJAX es una combinación de DHTML junto con otra tecnología que es la que marca la principal diferencia, el *objeto XMLHttpRequest* (este elemento tampoco es nuevo y fue propuesto por Microsoft en 1998). La verdadera diferencia de esta técnica radica en su filosofía de funcionamiento, dónde de forma transparente al usuario, el cliente mantiene una comunicación asíncrona con el servidor en un segundo plano. Realiza peticiones GET y POST obteniendo un documento XML como resultado y utilizando DOM le permite leer los datos y modificar la página. A efectos del usuario se verá que la página cambia de aspecto sin la necesidad de recargarla.

Las tecnologías básicas usadas para el desarrollo de aplicaciones Web con AJAX son:

- *XHTML* y *CSS* como estándares de presentación.
- *DOM* para mostrar e interactuar con la información.
- *XML* y *XSLT* manipulación e intercambio de datos (en la parte del servidor).
- *XMLHttpRequest* para el envío y la recepción de información de forma asíncrona.
- *JavaScript* como enlace para gestionar todas las tecnologías anteriores.

2.3.1. Modelo de aplicaciones Web con AJAX

Tradicionalmente, en las aplicaciones Web el usuario interactúa con el servidor a través de un formulario, que una vez completado, se le envía por medio del navegador como una petición Web. Este tipo de aplicaciones tienen ciertos límites y no es posible crearlas con las mismas características que las aplicaciones de escritorio (versatilidad, interactivas, etc.). Cada vez

que se requieran datos del servidor se debe recargar elementos comunes (información de presentación). Esto hace, por un lado, que la transferencia de información sea mayor y por otro, que el usuario tenga que estar a la espera de cada nueva recarga.

Fundamentalmente, AJAX establece una capa entre la interfaz del cliente y la aplicación de servidor, que hace que las operaciones de comunicación y trasiego de información junto con el refresco de datos de cara al usuario se hagan de manera transparente para éste. De esta forma se consigue parte de la potencia de las aplicaciones de escritorio y además se mejora la comunicación, puesto que la nueva capa introducida (*motor AJAX*) solamente obtiene los datos necesarios en cada momento sin necesidad de descargar la estructura del documento en cada operación.

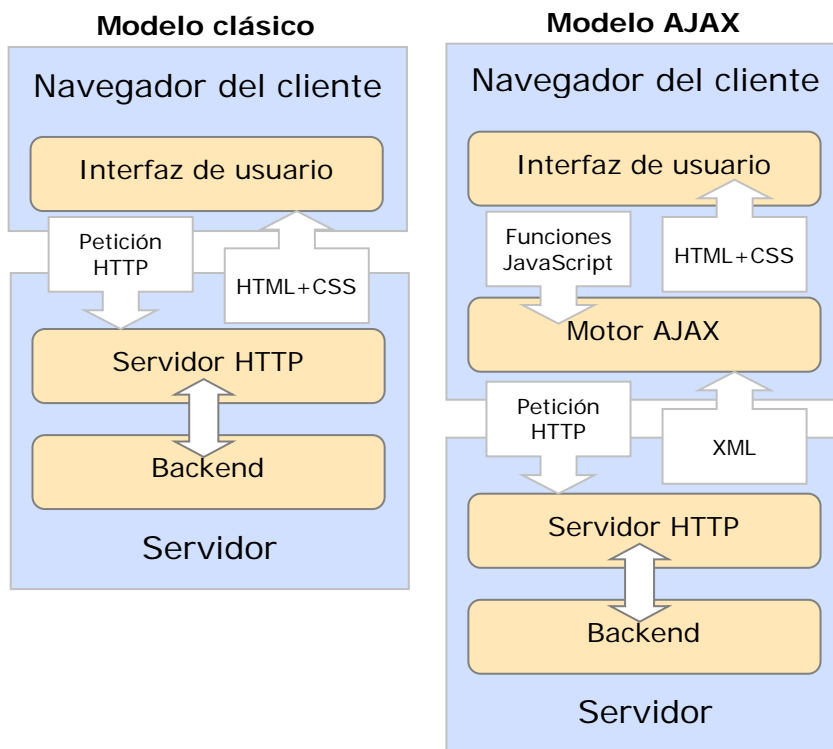


Figura 2.4. Modelo tradicional Web y modelo AJAX.

El *motor AJAX* o, conceptualmente, la nueva capa que se introduce, no es más que un conjunto de archivos *js* utilizados como librerías, que se encargan de *renderizar* la interfaz del usuario y realizar la comunicación con el servidor de manera transparente al usuario. Este motor permite establecer una interacción entre el usuario y la aplicación de forma asíncrona (figura 2.5), es decir, independientemente de la comunicación con el servidor.

Cualquier acción realizada por el usuario, en lugar de generar una petición HTTP, genera una llamada a una función JavaScript al *motor AJAX*. Si para llevarse a cabo la petición del usuario no es necesaria información del servidor o no requiere realizar una acción en el servidor, se resolverá en el propio *motor AJAX* ubicado en el cliente (validación de datos, visualización de información en memoria, etc.). Si por lo contrario, la petición requiere de la ejecución de un proceso en el servidor para generar la respuesta, el *motor AJAX* realiza las peticiones necesarias de manera asíncrona, mediante algún objeto como el *XMLHttpRequest*, sin parar la interacción con el usuario.

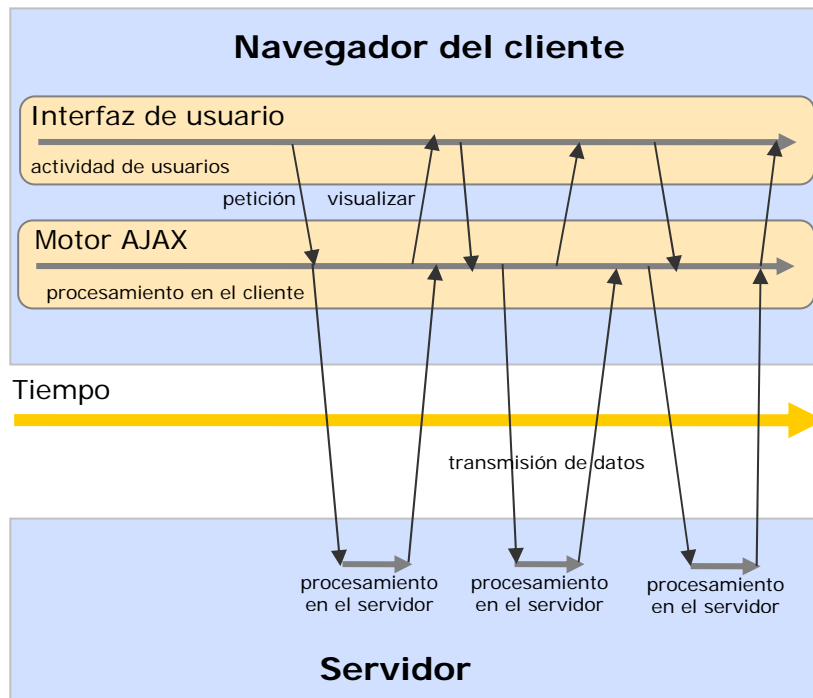


Figura 2.5. Modelo asíncrono de AJAX.

De una manera más concreta podríamos describir el funcionamiento de AJAX con los siguientes pasos:

- El usuario interactúa con la interfaz Web provocando un evento.
- El motor AJAX crea e inicializa un objeto *XMLHttpRequest*.
- El objeto *XMLHttpRequest* realiza una petición HTTP al servidor.
- El servidor procesa la petición y devuelve como resultado la información estructurada en XML.
- El objeto *XMLHttpRequest* obtiene la información como XML y se llama a la función encargada de procesarla.
- Se actualiza el DOM de la página asociada a la petición a partir del resultado devuelto por el servidor.

2.3.2. El objeto *XMLHttpRequest*

El objeto *XMLHttpRequest* fue desarrollado originalmente por Microsoft como un objeto ActiveX y está presente desde la versión 5 del Internet

Explorer. Este objeto ha sido implementado en otros navegadores como en Mozilla desde la versión 1.0 y en Safari desde la 1.2. Por este motivo su uso se ha extendido a navegadores ampliamente utilizados. Este objeto ofrece un API que permite realizar conexiones HTTP con el servidor el cual puede devolver la respuesta usando XML, texto plano, JavaScript o JSON. El objeto ha sido utilizado por los diferentes *lenguajes de Script* y utiliza para la comunicación un canal de conexión independiente. Este objeto permite a lenguajes como JavaScript realizar peticiones HTTP a un servidor remoto sin la necesidad de volver a cargar la página que se este visualizando. Como se comentó anteriormente con este objeto se pueden estar haciendo peticiones y recibiendo respuestas del servidor en segundo plano sin que el usuario final sea consciente de este proceso.

Un pequeño problema es, que, mientras que para utilizar el objeto desde JavaScript en un navegador IE se instancia un objeto ActiveX como se muestra en la listado 2.15, en Mozilla o Safari se trata de un objeto nativo (listado 2.16), y su forma de instanciarlo es completamente diferente. Se debe tener en cuenta este aspecto para hacer la aplicación compatible con los navegadores.

```
Var req = new ActiveXObject("Microsoft.XMLHTTP");
```

Listado 2.15. Instancia del objeto como ActiveX.

```
Var req = new XMLHttpRequest();
```

Listado 2.16. Instancia del objeto nativo de Mozilla o Safari.

Con motivo de esta inconsistencia en el objeto es posibles establecer la funcionalidad en JavaScript que permita indicar el objeto que se debe usar, como se muestra en el listado 2.17.

```

var req;

function loadXMLDoc(url)
{
    // Opción para objeto XMLHttpRequest nativo
    if (window.XMLHttpRequest) {
        req = new XMLHttpRequest();
        req.onreadystatechange = processReqChange;
        req.open("GET", url, true);
        req.send(null);

        // Opción para objeto ActiveX de IE/Windows
    } else if (window.ActiveXObject) {
        req = new ActiveXObject("Microsoft.XMLHTTP");
        if (req) {
            req.onreadystatechange = processReqChange;
            req.open("GET", url, true);
            req.send();
        }
    }
}

```

Listado 2.17. Instancia del objeto nativo de Mozilla o Safari.

Los métodos que ofrece el API *XMLHttpRequest* son:

- *abort()*: Detiene la petición actual.
- *getAllResponseHeaders()*: Devuelve todas las cabeceras de la respuesta como pares de etiqueta y valores en una cadena.
- *getResponseHeader("headerLabel")*: Devuelve el valor de una cabecera determinada.
- *open("method", "URL", asyncFlag, "userName", "password")*: Asigna la *URL* de destino, el método y otros parámetros opcionales de una petición pendiente.
- *send(content)*: Envía la petición, opcionalmente se puede enviar una cadena de texto o un objeto DOM.
- *setRequestHeader("label", "value")*: Asigna un valor al par *label/value* para la cabecera enviada.

De la misma forma las propiedades que ofrece son:

- *onreadystatechange*: El manejador del evento llamado en cada cambio de estado del objeto. Permite preparar la recepción de la respuesta del servidor asociando el manejador a una función que tratará la respuesta.

- *readyState*: Entero que indica el estado del objeto (0 = sin inicializar, 1 = cargando, 2 = fin de la carga, 3 = actualizando la información recibida, 4 = Operación completada).
- *responseText*: Cadena de texto con los datos devueltos por el servidor.
- *responseXML*: Objeto DOM devuelto por el servidor.
- *status*: Código numérico devuelto por el servidor, ejemplos: 404 si la página no se encuentra, 200 si todo ha ido bien, etc.
- *statusText*: Mensaje que acompaña al código de estado.

A continuación se muestra un pequeño ejemplo básico de uso del AJAX con la aplicación "Hola Mundo".

Lo primero que se debe hacer es crear una variable (*httpRequest*) que será la que contenga la instancia al *objeto XMLHttpRequest*. Después, como se comentó anteriormente se debe crear la parte de código que nos permita instanciar el *objeto XMLHttpRequest* en función del navegador desde el cual se solicite la página.

Una vez instanciado se debe asociar al manejador la función que realizará la lectura de los datos enviados por el servidor. Este proceso como se muestra en el listado 2.19 se realiza con el atributo o propiedad `onreadystatechange` y la función se ejecutará cada vez que la propiedad `readyState` del *objeto XMLHttpRequest* cambie de estado. Para leer estos datos se deberá comprobar que la petición se encuentre en estado 4, como se realiza en la función asociada `alertContent`. En el ejemplo simplemente se mostraría un mensaje de alerta con el contenido de la respuesta del servidor, pero en función del objetivo de la aplicación se podría manejar el resultado mediante DOM para presentar al usuario los datos finales.


```

<script type="text/javascript" language="javascript">
function makeRequest(url) {
    var httpRequest;

    if (window.XMLHttpRequest) { // Mozilla, Safari, ...
        httpRequest = new XMLHttpRequest();
        if (httpRequest.overrideMimeType) {
            httpRequest.overrideMimeType('text/xml');
        }
    }
    else if (window.ActiveXObject) { // IE
        try {
            httpRequest = new ActiveXObject("Msxml2.XMLHTTP");
        }
        catch (e) {
            try {
                httpRequest = new ActiveXObject("Microsoft.XMLHTTP");
            }
            catch (e) {}
        }
    }
    if (!httpRequest) {
        alert('Giving up :( Cannot create an XMLHTTP instance');
        return false;
    }
}

```

Listado 2.18. “Hola Mundo” con AJAX, parte I.

Una vez asociada la función al manejador se debe abrir la conexión con el servidor mediante los métodos GET, POST o HEAD. Para ello se debe indicar la *URL* de la que se desea obtener los datos, el tipo de modo a usar (en AJAX siempre se usará el modo asíncrono indicando como valor del parámetro `true`). Para realizar la conexión se debe llamar al método `send` del objeto *XMLHttpRequest* después de `open`.

El método `send` envía la petición con los datos pasados como parámetros como cuerpo de la petición.

En la última parte del código de ejemplo del listado 2.19 se mostraría al usuario un enlace que al pulsarlo invocaría a la función que inicia todo el proceso (`makeRequest`).

```

    httpRequest.onreadystatechange = function() {
        alertContents(httpRequest);
    };
    httpRequest.open('GET', url, true);
    httpRequest.send(null);
}
function alertContents(httpRequest) {
    if (httpRequest.readyState == 4) {
        if (httpRequest.status == 200) {
            alert(httpRequest.responseText);
        }
        else {
            alert('There was a problem with the request.');
```

Listado 2.19. “Hola Mundo” con AJAX, parte II.

En el listado 2.20 se muestra el documento XML que devuelve el servidor como respuesta a la petición realizada por el cliente.

```

<?xml version="1.0" ?>
<root>
    Hola mundo!!!
</root>
```

Listado 2.20. Archivo XML devuelto por el servidor (holamundo.xml).

2.4. APLICACIONES LIGERAS

Existe la posibilidad de extender el comportamiento y las funcionalidades de los navegadores Web. Por un lado, se pueden introducir componentes software para que se ejecuten o corran en el contexto de terceras aplicaciones, en este caso en los navegadores Web. En este primer caso el navegador debe soportar este tipo de aplicaciones de forma nativa. Estas aplicaciones ligeras no tienen sentido de manera independiente. La

programación de estos componentes es más compleja y requieren más conocimientos que el uso de los *lenguajes de Script*. Otra gran diferencia, es que estos componentes han sido compilados previamente a su utilización y lo que recibe el navegador Web es código binario. En el caso de los *Script*, es el propio navegador el que cada vez que se carga la página o se inicia una acción interpreta el código. Una de las ventajas de estos componentes es que suelen ser más potentes que los *lenguajes de Script* interpretados en el cliente puesto que se basan en lenguajes de programación de alto nivel con las características de dichos lenguajes. Pero no todos los navegadores soportan este tipo de aplicaciones de manera nativa, debido a la cantidad de aplicaciones ligeras que aparecen cada día. Por este motivo, la mayoría de los navegadores implementan una arquitectura de plug-ins o conectores, que permite añadir módulos para incorporar aplicaciones ligeras.

2.4.1. Applets

Un *Applet* es un programa escrito en lenguaje Java que se puede incluir en una página HTML de la misma manera que incluimos una imagen. Cuando se solicita desde un navegador una página Web, esta puede contener una referencia a un Applet. Cuando el navegador esté interpretando la página HTML llegará hasta la etiqueta que incluye el Applet y posteriormente le solicita al servidor la transmisión del código binario del Applet. Una vez el código es transferido al cliente, es posible ejecutarlo en el contexto de la máquina virtual de java asociada al navegador, pero en el lado del cliente (generalmente con la ayuda de un plug-in). Generalmente, los principales inconvenientes del uso de los Applets en las aplicaciones Web son: por un lado, el tiempo que se requiere para la transmisión del Applet y que dependerá de la línea que se posea y por otro lado, que el navegador se haya habilitado para soportar aplicaciones java, es decir que se incluya la máquina virtual de java para poder ejecutar el Applet.

Como desventajas, en relación con JavaScript, cabe señalar que los Applets son más lentos de procesar y que tienen espacio muy delimitado en la página donde se ejecutan, es decir, no se mezclan con todos los componentes de la página ni tienen acceso a ellos. Por este motivo, con los Applets de Java no se puede, directamente, hacer cosas como abrir ventanas secundarias, controlar Frames, formularios, capas, etc, pero si ampliar la funcionalidad de nuestro navegador e incluso acceder a componentes del lado del servidor.

La principal ventaja de utilizar Applets consiste en que son mucho menos dependientes del navegador que los *Scripts* en JavaScript, incluso independientes del sistema operativo del ordenador donde se ejecutan. Además, Java es más potente que JavaScript, por lo que las aplicaciones de los Applets podrán ser mayores.

En la figura 2.21 se puede ver el código para crear un sencillo Applet. Este tipo de Applet que extiende de la clase `java.applet.Applet` se utiliza cuando no vamos hacer uso de componentes *GUI* de tipo *Swing* y utiliza únicamente los componentes *AWT*. Pero esta opción comienza a estar en desuso y se utiliza la clase `javax.swing.JApplet` la cual permite introducir en el Applet componentes *GUI de tipo Swing*. El único problema con este último tipo de applets es que el plug-in instalado debe soportar *Swing*.

```
import java.applet.*;
import java.awt.*;
public class HolaMundo extends Applet {
    public void paint (Graphics g) {
        g.drawString ("¡Hola Mundo!!",10,80);
    }
}
```

Listado 2.21. Ejemplo de un Applet sencillo.

```
<html>
  <head>
    <title> Ejemplo simple de applet </title>
  </head>
  <body>
    <p>A continuación está la salida del programa</p>
    <applet code="Clock.class" width=170 height=150>
      <param name=bgcolor value="000000">
      <param name=fgcolor1 value="ff0000">
      <param name=fgcolor2 value="ff00ff">
    </applet>
    No hay disponible un intérprete de Java
  </body>
</html>
```

Listado 2.22. Inserción de un Applet en un documento HTML.

2.4.1.1. Ciclo de vida de un Applet

Mediante la invocación de ciertos métodos, un navegador gestiona el ciclo de vida de un Applet, si el applet es cargado en una página Web.

El ciclo de vida de un Applet básicamente se compone de cuatro pasos relacionados con un método cada uno.

- El método `init` es utilizado para realizar los procesos de inicialización del Applet, si es necesario. Este método es invocado después de que el navegador lea el atributo `param` de la etiqueta `<applet>`.
- El método `start` es automáticamente invocado por el navegador después del método `init`. También se invoca a este método siempre que el usuario regrese a la página que lo contiene después de haber navegado por otras páginas.
- El método `stop` es invocado de forma automática siempre y cuando el usuario abandone la página que contiene el Applet.
- El método `destroy` es invocado cuando el navegador es cerrado siguiendo el cauce habitual.

De esta forma, el Applet puede ser inicializado una única vez, arrancado y parado una o más veces en su ciclo de vida y destruido, también, una única vez.

La etiqueta `<APPLET>` de arriba especifica que el navegador debería cargar la clase cuyo código compilado está en el fichero llamado `HelloWorld.class`. El navegador busca este fichero en el mismo directorio que contiene el documento HTML que contiene la etiqueta.

Cuando el navegador encuentra el fichero de la clase, la carga a través de la red, si es necesario, en el ordenador donde se está ejecutando el navegador. Entonces el navegador crea un ejemplar de la clase. Si incluimos un applet dos veces en la misma página, el navegador sólo cargará el Applet una vez y creará dos ejemplares de esa clase.

Cuando un navegador que soporte Java encuentra una etiqueta `<APPLET>`, reserva un área de pantalla de la anchura y altura especificadas, carga los *bytecodes* de la subclase `Applet` especificada, crea un ejemplar de la subclase y luego llama a los métodos `init` y `start` del ejemplar.

Incluimos los Applets en páginas HTML usando la etiqueta `<APPLET>`. Cuando un navegador visita una página que contiene un Applet suceden los siguientes pasos:

- El navegador busca el archivo `.class` de la subclase del Applet.

- La localización del archivo *.class* (que contiene los *bytecodes* Java) se especifica con los atributos `CODE` y `CODEBASE` de la etiqueta `<APPLET>`.
- El navegador descarga los *bytecodes* a través de la red hasta el ordenador del usuario.
- El navegador crea un ejemplar de la subclase `Applet` (cuando nos referimos a un `Applet`, generalmente nos referimos a este ejemplar).
- El navegador llama al método `init` del `Applet` (este método realiza cualquier inicialización que sea necesaria).
- El navegador llama al método `start` del `Applet` (este método normalmente arranca un *thread* que realiza las tareas del `Applet`).

Una subclase `Applet` es la clase principal, la clase controladora, pero los `Applets` también pueden usar otras clases. Estas otras clases pueden ser locales del navegador, proporcionadas como parte del entorno Java, o ser clases personalizadas que el usuario suministra. Cuando el `Applet` intenta utilizar una clase por primera vez, el navegador intenta encontrarla en el *host* en el que se está ejecutando. Si no puede encontrarla allí, busca la clase en el mismo lugar de donde vino la subclase de `Applet`. Cuando el navegador encuentra la clase, carga sus *bytecodes* (a través de la red, si es necesario) y continúa ejecutando el `Applet`.

Cargar código ejecutable a través de la red es un riesgo de seguridad clásico. Para los `Applets` Java, este riesgo se reduce porque el lenguaje Java está diseñado para ser seguro (por ejemplo, no permite punteros a memoria). Además, los navegadores compatibles Java mejoran la seguridad imponiendo sus propias restricciones. Estas restricciones incluyen no permitir a los `Applets` que carguen código escrito en otros lenguajes distintos de Java, y no permitiendo que los `Applets` lean o escriban ficheros en el *host* del navegador.

2.4.1.2. Instalación de Java plug-in en navegadores

Existen pequeños navegadores, como el *appletviewer* proporcionado por SUN en la máquina virtual de java JDK 1.2 que ya incorpora la compatibilidad con los `Applets`.

En el resto de navegadores independientes de Java, para poder visualizar un `Applet` en un navegador tipo IE o Mozilla Firefox es necesario instalar el plug-in de Java. El *Java plug-in* es incluido como parte del entorno de ejecución de Java, JRE, y permite establecer una conexión entre los navegadores más populares y la plataforma Java. Este permite a los `Applets` ejecutarse en el contexto del navegador. Cargar el *Java plug-in* es un proceso muy sencillo que únicamente requiere instalar la máquina

virtual de java JRE. Automáticamente se podrán visualizar los Applets en nuestro navegador.

Una vez instalado tenemos que asegurarnos que está habilitada en el navegador la opción para permitir usar el JRE y visualizar Applets.

En el navegador IE debemos ir al menú de *herramientas* → *opciones de Internet* → pestaña de *opciones avanzadas* y asegurarnos que en el *check box Java (Sun)* este marcado, de lo contrario no podremos visualizar el Applet.

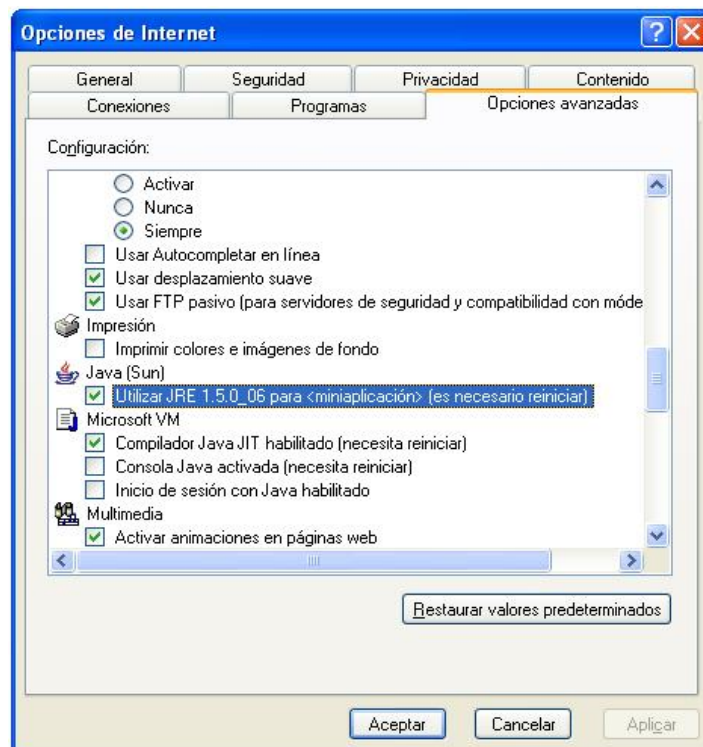


Figura 2.6. Habilitar en IIS el uso de Applets.

En el navegador Mozilla Firefox se requiere un proceso similar. En el menú *Herramientas* → *opciones* → *pestaña contenido* activamos la casilla cuya inscripción dice *Activar Java*. Con esto se obtiene el mismo resultado que en el proceso del IE.

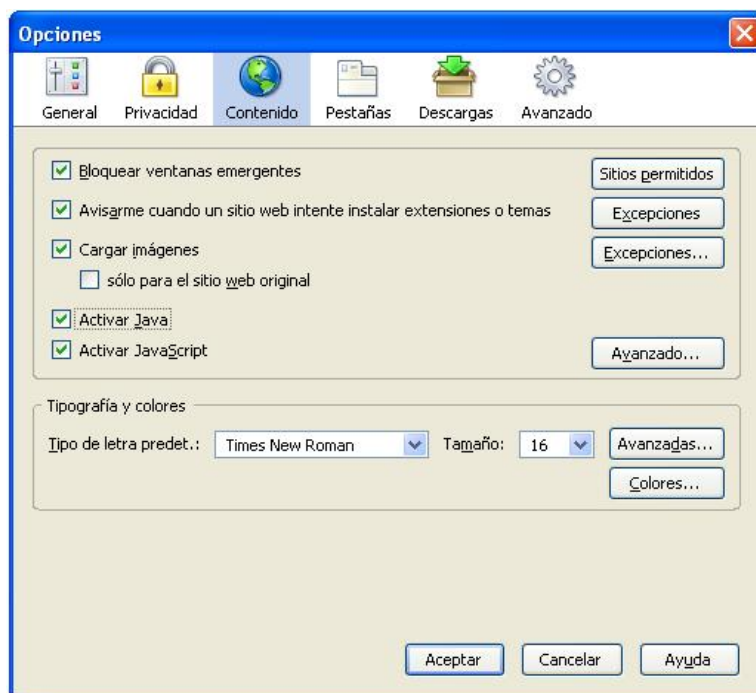


Figura 2.7. Habilitar en Firefox el uso de Applets.

El panel de control de java, el cual se puede encontrar en el panel de control del sistema operativo permite configurar y ver un amplio rango de parámetros de configuración para controlar las características de la máquina virtual de java en el propio entorno. Por tanto controlar también la configuración del plug-in. Para más información se puede acceder a la *URL* <http://java.sun.com/j2se/1.5.0/docs/guide/deployment/deployment-guide/jcp.html>.

2.4.2. ActiveX

Tecnología utilizada, entre otras cosas, para dotar a las páginas Web de mayores funcionalidades, como animaciones, vídeo, navegación tridimensional, etc. Los controles ActiveX son pequeños programas que se incluyen dentro de estas páginas. Lamentablemente, por ser programas, pueden ser el objetivo de algún virus.

ActiveX es una tecnología de Microsoft para el desarrollo de páginas dinámicas. Tiene presencia en la programación del lado del servidor y del lado del cliente, aunque existan diferencias en el uso en cada uno de esos dos casos.

Son pequeños programas que se pueden incluir dentro de páginas Web y sirven para realizar acciones de diversa índole. Por ejemplo hay controles ActiveX para mostrar un calendario, para implementar un sistema de FTP, etc.

Son un poco parecidos a los Applets de Java en su funcionamiento, aunque una diferencia fundamental es la seguridad, pues un Applet de Java no podrá tomar privilegios para realizar acciones malignas (como borrar el disco duro) y los controles ActiveX sí que pueden otorgarse permisos para hacer cualquier cosa.

Los controles ActiveX son particulares de Internet Explorer aunque existen plug-ins específicos para poder ejecutar controles ActiveX dentro del contexto de otros navegadores.

Los controles ActiveX son componentes de software que se pueden descargar y que se utilizan para ampliar las funciones de Internet Explorer a través de elementos de la interfaz de usuario, como menús emergentes, botones.

Mientras se explora la Web se tiene protección ante la descarga de controles ActiveX no deseados, ya que si una página intenta descargar uno de ellos, Internet Explorer le mostrará un certificado firmado con el nombre de la compañía que creó el control. Este certificado aparece como un cuadro de diálogo de advertencia de seguridad. Si se encuentra en un sitio Web de su confianza (en este caso, Windows Update) y el control ActiveX lo proporciona una compañía en la que confía (en este caso, Microsoft), puede aceptar el certificado y permitir la instalación del control.

Aunque en principio los controles ActiveX fueron creados para utilizarse con Internet Explorer, posteriormente se han creado plug-ins para otros navegadores de uso común como Netscape y Mozilla Firefox.

Uno de los problemas a diferencia de los Applets es que los controles ActiveX tienen acceso completo al sistema operativo con el peligro que ello puede conllevar. Para controlar esta falla Microsoft creó un sistema de registro para que los navegadores puedan identificar y autenticar un control ActiveX antes de descargarlo y ejecutarlo.

```
<html>
  <head>
    <title> Ejemplo simple de activeX </title>
  </head>
  <body>
    <object classid="clsid:02BF25D5-8C17-4B23-BC80-D3488ABDDC6B"
width="160" height="144"
codebase="http://www.apple.com/qtactivex/qtplugin.cab">
      <param name="SRC" value="sample.mov">
      <param name="AUTOPLAY" value="true">
      <param name="CONTROLLER" value="false">
    </object>
  </body>
</html>
```

Listado 2.23. Inserción de ActiveX en un documento HTML.

En el listado 2.23. se muestra un ejemplo para incluir un control ActiveX dentro de un documento HTML (el control ActiveX para Quicktime). El atributo `classid` identifica de forma única que control ActiveX se va a usar. Un atributo `classid` con el valor `clsid:02BF25D5-8C17-4B23-BC80-D3488ABDDC6B` indica al navegador Internet Explorer que use el control ActiveX para Quicktime. El desarrollador debe conocer este valor. El atributo `codebase` indica la localización del control ActiveX. El usuario hará uso de este valor únicamente si no tienen instalado el control de forma que el navegador accederá a la ubicación y descargará el control. El navegador Internet Explorer automáticamente se ofrecerá para descargar e instalar el control.

2.4.2.1. Configuración de Internet Explorer con ActiveX

Ya se ha comentado anteriormente los peligros que puede conllevar permitir la ejecución de controles ActiveX sin control. En el navegador Internet Explorer por defecto las opciones de descarga y ejecución de ActiveX se encuentran deshabilitadas dejando el control al usuario si desea descargar y ejecutar los controles. En función del nivel de seguridad que tengamos asignado o definido se actuará de diversas formas.

La opción `personalizar`, da un mayor control a los usuarios avanzados y a los administradores sobre todas las opciones de seguridad. Por ejemplo, la opción `descargar` los controles ActiveX sin firmar se deshabilita de forma predeterminada en la *zona Intranet local* (la seguridad media es la configuración predeterminada de la *zona Intranet local*). En este caso, Internet Explorer no puede ejecutar ningún control de ActiveX en la intranet de su organización porque la mayoría de las organizaciones no

firman los controles ActiveX que sólo se utilizan internamente. Para que Internet Explorer ejecute los controles ActiveX sin firmar en la intranet de una organización, cambie en el nivel de seguridad de la opción *Descargar*, los controles ActiveX sin firmar a *Preguntar* o *Habilitar* para la zona *Intranet local*. En las opciones de seguridad en la opción *Nivel personalizado* se puede establecer el acceso a los archivos, controles ActiveX y secuencias de comandos.

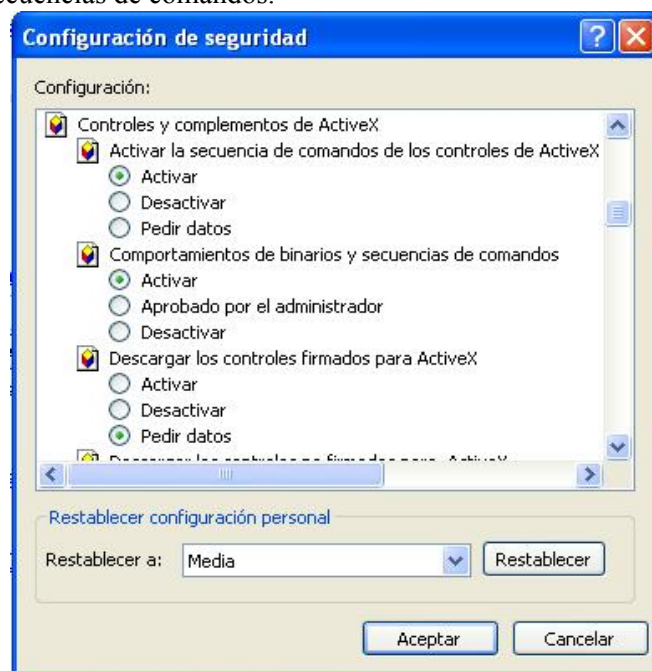


Figura 2.8. Opciones de seguridad para controles ActiveX en IE.

2.4.2.2. Plug-in de ActiveX para Mozilla.

El navegador Mozilla no soporta controles ActiveX de forma nativa. Se ha desarrollado un plug-in, el cual también trabaja con Netscape 4.x, que permite utilizar controles ActiveX en estos navegadores. Este plug-in se ha creado con ciertas limitaciones puesto que no permite descargar controles e instalarlos de forma automática. Solamente permite ejecutar controles que ya se encuentran instalados e identificados como seguro para ser incrustado.

Se puede obtener más información en el *Mozilla ActiveX Project* en la URL <http://www.iol.ie/~locka/mozilla/mozilla.htm>.

Se debe tener en cuenta que los controles ActiveX pueden suponer un riesgo para nuestro equipo.

A continuación se describe el proceso de instalación del plug-in para *Mozilla Firefox 1.5*. Se debe tener en cuenta que el plug-in se realiza para versiones específicas del navegador y que podría acarrear daños si se utiliza con versiones no adecuadas (se debería realizar una desinstalación del plug-in).

Se puede descargar el plug-in para la versión nombrada anteriormente en:

<http://www.iol.ie/~locka/mozilla/plugin.htm#download>

Para ello se debe descargar el archivo con extensión *xpi* y ejecutamos el navegador Firefox. En el menú *archivo* elegimos *abrir archivo* y seleccionamos el archivo *xpi* descargado (en este caso *mozactivex-ff-15.xpi*). Una vez seleccionado y abierto se no ofrece la opción de *instalar el plug-in*.

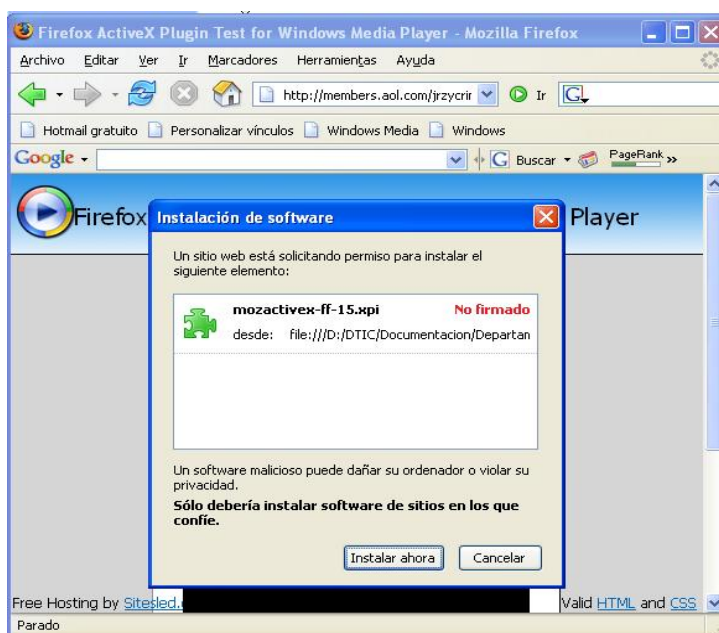


Figura 2.9. Opciones de seguridad para controles ActiveX en Mozilla.

Una vez instalado debemos cerrar el navegador y volverlo a ejecutar. Si no ha habido ningún problema se pueden visualizar controles ActiveX en este navegador.

Se puede realizar una prueba accediendo a la URL <http://members.aol.com/jrzycrim01/mozilla/wmp/wmpaxtest.html> y confirmar si se pueden ver los controles ActiveX que instancia el documento HTML descargado.

Para desinstalar el plug-in se deben seguir los siguientes pasos:

- Cerrar todas las instancias del navegador.
- Ir al directorio del *Mozilla Firefox*.
- Ir al directorio *Mozilla Firefox\plugins* y borrar el archivo *npmozax.dll*.
- Volver al directorio del *Mozilla Firefox* e ir al directorio *\components* para borrar los archivos *nsIMozAxPlugin.xpt* y *nsAxSecurityPolicy.js*.
- Acceder a la ruta *Mozilla Firefox\defaults\pref* y borrar el archivo *activex.js*.

2.4.3. Extensión del navegador mediante plug-ins

Los plug-ins son aplicaciones o módulos externos al navegador que permiten extender su funcionalidad para poder ejecutar mini-aplicaciones que doten de mayor potencia al navegador sin que el navegador tenga que soportarlas de una forma nativa.

Los plug-ins van a permitir que el navegador actúe como contenedor de otras aplicaciones gestionando su ciclo de vida. Esta técnica surge ante la imposibilidad de que un navegador Web sea capaz de procesar todos los tipos de datos que puede enviar como respuesta un servidor Web.

Un plug-in es cargado por el navegador si los datos enviados por el servidor son del tipo asociado al plug-in (la asociación se realiza mediante el tipo MIME).

2.4.3.1. Funcionamiento de los plug-ins

Se ha dividido la forma de actuar del navegador en referencia a los plug-ins en dos categorías. La primera en la cual se recibe un flujo de datos el cual requiere de la actuación de un plug-in directamente y no está insertado en código HTML. La segunda consiste en la inserción de objetos que requieren el uso de un plug-in por parte del navegador en código HTML.

En el primer caso, el cliente hace una petición mediante una URL a un servidor Web (<http://localhost/prueba.pdf>).

El servidor responde al cliente con el contenido mostrado en el listado 2.24. Cuando el navegador recibe el flujo de datos analiza la

cabecera (Content-Type) y dependiendo del tipo MIME que se indica el navegador invocará al plug-in correspondiente para mostrar el contenido. En este caso se ejecutaría el plug-in de *Adobe Acrobat Reader*.

```
HTTP/1.1 200 OK
Date: Mon, 30 Oct 2006 20:17:17 GMT
Server: Apache/2.2.3 (Win32)
Last-Modified: Mon, 30 Oct 2006 19:52:12 GMT
ETag: "132f6-16e3-9b217560"
Accept-Ranges: bytes
Content-Length: 5859
Connection: close
Content-Type: application/pdf

%ÔÏÏÏÏ1.4 /L 5859/O 8/E 1750/N 1/T 5693/H [ 476 149]>>
xref
6 9
0000000016 00000 n
0000000625 00000 n
0000000701 00000 n
0000000833 00000 n
0000000916 00000 n
trailer
----8CFDA75A284D5A8033E016C87CBCE897-
.....
.....
```

Listado 2.24. Contenido de un archivo *pdf*.

El segundo caso se produce cuando se insertan objetos dentro de un documento HTML. En la especificación de HTML 4.01 se ha definido el elemento *Object* como mecanismo para invocar a los plug-ins. Este elemento es usado de forma diferente en el navegador Internet Explorer que en los navegadores basados en Mozilla. Para conocer más sobre este elemento se puede consultar la especificación de HTML 4.01 en <http://www.w3.org/TR/1999/PR-html40-19990824/>.

```
<!ELEMENT OBJECT - - (PARAM | %flow;)*

<!ATTLIST OBJECT
%attrs;
declare          (declare)          #IMPLIED
classid          %URI;              #IMPLIED
codebase         %URI;              #IMPLIED
data             %URI;              #IMPLIED
type             %ContentType;      #IMPLIED
codetype         %ContentType;      #IMPLIED
archive          %URI;              #IMPLIED
standby          %Text;              #IMPLIED
height           %Length;           #IMPLIED
width            %Length;           #IMPLIED
usemap           %URI;              #IMPLIED
name             CDATA              #IMPLIED
tabindex         NUMBER             #IMPLIED>
```

Listado 2.25. DTD del elemento object.

En primer lugar, el navegador Internet Explorer invoca a un plug-in creado como ActiveX. Esto implica que se debe indicar el identificador del ActiveX dentro del elemento `object` (atributo `classid`). Esto implica que el programador debe conocer el identificador único de cada plug-in. El atributo `codebase` usado apunta a la localización donde está el archivo *CAB* que contiene el control del ActiveX que actúa como plug-in. En este contexto, el atributo `codebase` se usa como *mecanismo de obtención*, lo cual quiere decir, que se trata de una forma de obtener el controlador si no está presente. Por ejemplo, si el control de ActiveX de Flash no está instalado, IE irá entonces a la *URL* indicada en el atributo `codebase` y obtendrá el control de ActiveX que permita visualizar la película.

Los atributos `param` especifican los parámetros de configuración para el plug-in.

```
<object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/s
wflash.cab#version=5,0,0,0"
width="366" height="142" id="myFlash">
  <param name="movie" value="javascript-to-flash.swf" />
  <param name="quality" value="high" />
  <param name="swliveconnect" value="true" />
</object>
```

Listado 2.26. Plugins en Internet Explorer.

Los navegadores basados en Mozilla soportan la arquitectura de plug-in de Netscape, los cuales no están basados en COM como el ActiveX (y por ello, no son llamados vía identificador único) si no, basados en el tipo MIME. Cuando el navegador interpreta un documento HTML y se encuentra con un elemento `object` este hace uso del atributo `type` el cual contiene el tipo MIME que identifica el tipo de objeto y por tanto le indica al navegador el plug-in al cual debe invocar. Puede suceder que el plug-in no esté instalado y se nos ofrezca un mensaje indicando la ubicación para descargarlo o que el propio navegador la conozca e indique si la queremos instalar.

```
<object type="application/x-shockwave-flash" data="javascript-to-
flash.swf"
width="366" height="142" id="myFlash">
  <param name="movie" value="javascript-to-flash.swf" />
  <param name="quality" value="high" />
  <param name="swliveconnect" value="true" />
  <p>You need Flash -- get the latest version from
  <a href= "http://www.macromedia.com/downloads/">here.</a></p>
</object>
```

Listado 2.27. Plug-ins en Firefox.

Para usar el plug-in de java en los navegadores e introducir Applets en los documentos HTML se sigue utilizando en la mayoría de las aplicaciones la etiqueta `<APPLET>` en lugar de `<OBJECT>` aunque en la especificación de HTML 4.1 se ha desechado esta opción y se recomienda usar el elemento `Object`.

2.5. MANTENIMIENTO DE LA SESIÓN: COOKIES

Puesto que HTTP es un *protocolo petición/respuesta* las peticiones son tratadas de manera individual. Las aplicaciones Web necesitan un mecanismo que les permita identificar un determinado cliente y el estado de cualquier conversación que se mantiene con los clientes. Una sesión es una corta secuencia de peticiones de servicio realizadas por un único usuario por medio de un único cliente para acceder al servidor. El estado de la sesión es la información mantenida en la sesión a través de las peticiones.

Una de las técnicas para el almacenamiento de esta información ha sido las denominadas *Cookies*. Se trata de una potente herramienta empleada por los servidores Web para almacenar y recuperar información acerca de sus visitantes.

Las *Cookies* son pequeñas porciones de información que se almacena en el disco duro del visitante de una página web a través de su navegador, a petición del servidor de la página. Esta información puede ser luego recuperada por el servidor en posteriores visitas. Al ser el protocolo HTTP incapaz de mantener información por sí mismo, para que se pueda conservar información entre una página vista y otra (como *login* de usuario, preferencias de colores, etc.), ésta debe ser almacenada, ya sea en la *URL* de la página, en el propio servidor, o en una *Cookie* en el ordenador del visitante.

Una *Cookie* no es más que un archivo de texto que algunos servidores piden a nuestro navegador que escriba en nuestro disco duro, con información acerca de lo que hemos estado haciendo por sus páginas.

Entre las mayores ventajas de las *Cookies* se cuenta el hecho de ser almacenadas en el disco duro del usuario, liberando así al servidor de una importante sobrecarga. Es el propio cliente el que almacena la información y quien se la devolverá posteriormente al servidor cuando éste la solicite. Además, las cookies poseen una fecha de caducidad, que puede oscilar desde el tiempo que dure la sesión hasta una fecha futura especificada, a partir de la cual dejan de ser operativas.

Entre las tareas que realiza una *Cookie* podemos destacar:

- Llevar el control de usuarios: cuando un usuario introduce su nombre de usuario y contraseña, se almacena una *Cookie* para que no tenga que estar introduciéndolas para cada página del servidor. Sin embargo, una *Cookie* no identifica a una persona, sino a una combinación de computador y navegador.
- Ofrecer opciones de diseño (colores, fondos, etc.) o de contenidos al visitante.

- Conseguir información sobre los hábitos de navegación del usuario, e intentos de *spyware*, por parte de agencias de publicidad y otros. Esto puede causar problemas de privacidad y es una de las razones por la que las *Cookies* tienen sus detractores.

2.5.1. Funcionamiento de las Cookies

El protocolo HTTP ha creado una serie de cabeceras que permite realizar operaciones con *Cookies* (crear, obtener *Cookies*, cambiar el tiempo de expiración, borrar, etc.).

Cuando un cliente realiza una petición a un servidor Web, este como respuesta puede enviar información en la cabecera del documento HTTP para la creación de *Cookies* en el cliente. Esta operación se realiza mediante la directiva de cabecera `Set-Cookie`.

Es el servidor el que inicia la sesión al responder al cliente con un mensaje que tiene una cabecera para establecer una *Cookie* (`Set-Cookie`). Cuando el cliente recibe esta respuesta, en la siguiente petición incorporará una cabecera para informar de las *Cookies* que tiene (`Cookie`). Ante esta petición puede tener en cuenta la *Cookie* o no para dar la respuesta, y para ello puede establecer el mismo valor u otro valor para la *Cookie* o no enviar ninguna *cookie*. Para acabar la sesión, el servidor debe establecer una *Cookie* con duración "0".

El servidor puede tener varias cabeceras para establecer varias *Cookies* en una misma respuesta, aunque si el mensaje pasa por un proxy o una pasarela, pueden convertirlo en una sola cabecera para establecer *Cookies*. Para establecer una *Cookie* se usa la cabecera `Set-Cookie`, cuya sintaxis es:

```
Set-Cookie: nombre=valor *[:modificador]
```

Con esto se establece que por lo menos ha de haber un par nombre/valor y cada par puede tener o no modificadores.

Los modificadores de un par nombre, valor pueden ser:

- `Comment` = valor, el servidor informa del uso de la *Cookie*.
- `Domain` = valor, indica el dominio en el que la *Cookie* es válida.
- `Max-Age` = valor, validez de la *Cookie* en segundos, después se descarta.
- `Path` = valor, indica el subconjunto de *URL* a las que afecta la *Cookie*.
- `Secure`, indica que el cliente debe usar en entornos seguros para contactar con el servidor.

- `Version = valor`, la versión de la especificación de mantenimiento de estado que es.

Se pueden enviar varias *Cookies* en una misma cabecera separando los pares nombre, valor con “;”.

Además de enviar la *Cookie*, el servidor puede establecer las condiciones para que la *Cookie* se ponga en una *caché* o no.

Un ejemplo de la cabecera de una respuesta:

```
Set-Cookie: NOMBRE=VALOR; expires=FECHA; path=PATH;
domain=NOMBRE_DOMINIO; secure
Set-Cookie: ID=789; path=/
Set-Cookie: NOMBRE=Juan; path=/
.....
<html>
.....
</html>
```

Listado 2.28. Ejemplo de respuesta del servidor.

El cliente trata por separado las distintas informaciones de estado que le llegan por medio de respuestas que establecen *Cookies*. Además toma valores por defecto para los atributos que no están presentes de los pares nombre/valor.

Además el cliente puede rechazar una *Cookie* por razones de seguridad o violaciones de la privacidad. Para rechazar una *Cookie* tiene una serie de reglas que aplica en cada caso.

Si el cliente recibe una *Cookie* con un nombre que ya tiene asociado a otra *Cookie* y cuyos atributos `Domain` y `Path` son exactamente iguales, entonces la nueva *cookie* reemplaza a la anterior. Además si la nueva *Cookie* tiene el atributo `Max-Age` con un valor de 0, entonces elimina la *Cookie* en vez de reemplazarla.

Como el cliente tiene un espacio limitado para las *Cookies*, ha de ir eliminando *Cookies* antiguas para hacer sitio a las nuevas. Esta operación la puede realizar siguiendo algún algoritmo como el *LRU* (*Least Recently Used*), o *FIFO* (*First In First Out*).

Si una *Cookie* tiene el atributo `Comment`, entonces el cliente ha de almacenar el comentario para que el usuario (humano) pueda leerlo.

Cuando el cliente realiza una petición al servidor, lo hace por medio de una *URI*. Para cada petición, además del mensaje envía una cabecera *Cookie* para informar al servidor de las *cookies* que tiene y que afectan a esa petición. La sintaxis para esta cabecera es la siguiente:

```
Cookie: versión *((;|,)nombre = valor
                        [;path] [;dominio])
```

Donde versión es:

```
Version = valor
```

path es:

```
Path = valor
```

Y dominio es:

```
Domain = valor
```

El valor de la versión es el del atributo Versión si alguna de las *Cookies* lo impone, si no es 0. El valor de `path` ha de ser el impuesto por las *cookies* si alguna lo establece, si no se omite. El caso del dominio se trata igual que el del `path`.

En la lista de *Cookies* aparecen las que satisfacen los criterios:

- El nombre del servidor ha de tener el mismo dominio que la *Cookie*.
- El `path` de la *Cookie* ha de ser un prefijo de la *URI* que representa la petición.
- El límite de tiempo de la *cookie* no ha sido sobrepasado.

Cuando el servidor recibe una petición con una cabecera *Cookie*, ha de interpretar los pares nombre/valor teniendo en cuenta que los nombres que comienzan por el símbolo "\$" son especiales y hacen referencia a la *Cookie* anterior, y si no hay una *Cookie* anterior, hacen referencia a todas las *Cookies* de esta cabecera.

En la parte del cliente se ha establecido la cabecera *Cookie* que permite informar de las *Cookies* existentes.

En la cabecera de la siguiente solicitud:

```
NOMBRE1=VALOR1; NOMBRE2=VALOR2; ...
Cookie: ID=789; NOMBRE=JuanBla, bla, ...
.....
<html>
.....
</html>
```

Listado 2.29. Ejemplo de petición del cliente.

2.5.2. Habilitación de las Cookies

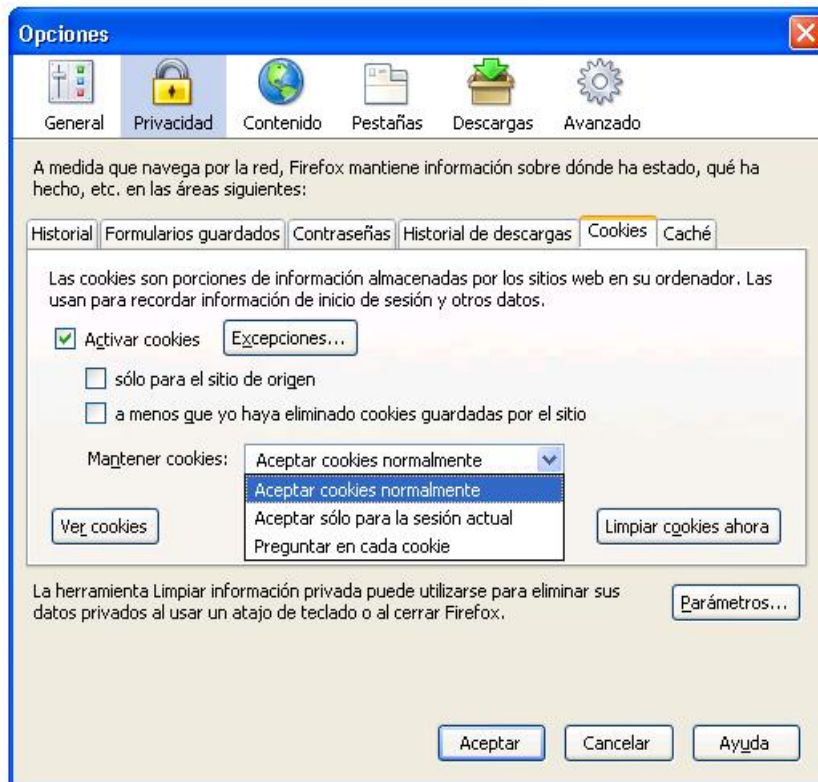
La mayor parte de los navegadores modernos soportan *Cookies*. Sin embargo, un usuario puede normalmente elegir si las *Cookies* deberían ser utilizadas o no. A continuación, las opciones más comunes:

- Las *Cookies* no se aceptan nunca.
- El navegador pregunta al usuario si se debe aceptar cada *Cookies*.
- Las *Cookies* se aceptan siempre.

El navegador también puede incluir la posibilidad de especificar mejor qué *Cookies* tienen que ser aceptadas y cuáles no. En concreto, el usuario puede normalmente aceptar alguna de las siguientes opciones: rechazar las *Cookies* de determinados dominios; rechazar las *Cookies* de terceros; aceptar *Cookies* como no persistentes (se eliminan cuando el navegador se cierra); permitir al servidor crear *Cookies* para un dominio diferente. Además, los navegadores pueden también permitir a los usuarios ver y borrar *Cookies* individualmente.

La mayoría de los navegadores que soportan JavaScript permiten a los usuarios ver las *Cookies* que están activas en una determinada página escribiendo `javascript:alert("Cookies: "+document.cookie)` en el campo de dirección.

En el navegador Mozilla Firefox podemos configurar los parámetros para permitir el almacenamiento de *Cookies*. Accediendo al menú de *herramientas* → *opciones* → *pestaña de privacidad* y en la *sub-pestaña de cookies* podemos configurar todas las posibilidades.

Figura 2.10. Habilitar en Firefox para el uso de *Cookies*.

Para configurar el almacenamiento de *Cookies* en el navegador Internet Explorer debemos realizar los siguientes pasos. Ir al menú de *herramientas* → *opciones de Internet* → *pestaña de privacidad* → *opciones avanzadas*. Ahora se pueden configurar las opciones de las *Cookies* en el cliente.

Si utilizamos en navegador Mozilla Firefox las *Cookies* son almacenadas en un único archivo llamado `cookies.txt` en `%HOME-PATH%\Datos de programa\Mozilla\Firefox\Profiles\hjp0b0b.default`.

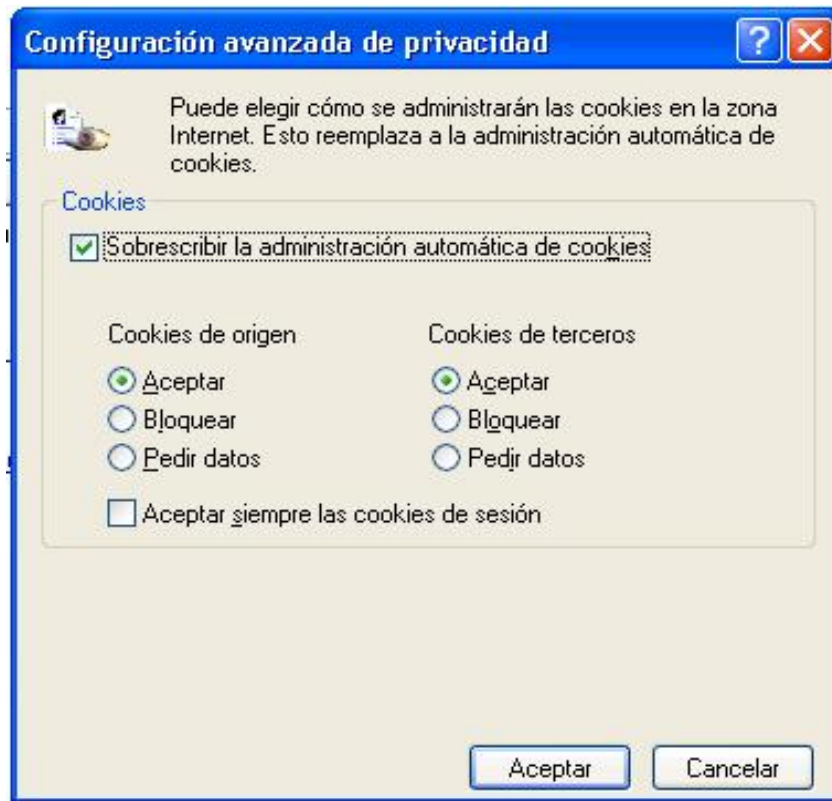
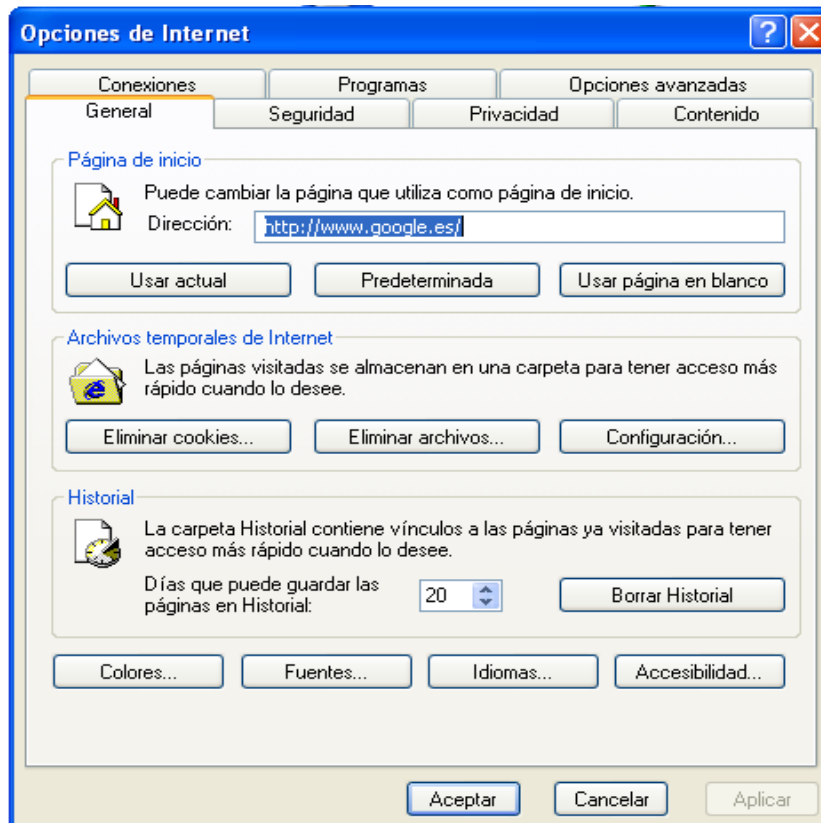


Figura 2.11. Habilitar en IE para el uso de *Cookies*.

Las cookies en Windows, utilizando el navegador Internet Explorer, normalmente se almacenan en %HOMEPATH%/cookies. En este directorio se pueden encontrar un elevado número de archivos con la información mantenida con cada sesión que se ha establecido con el servidor.

IE ofrece también una interfaz menos completa que solo permite eliminar todas las *Cookies*.

Figura 2.12. Interfaz que permite la eliminación de *Cookies*.

En ciertas ocasiones, por motivos de seguridad, no se permiten las *Cookies*, y por tanto, se debe acudir a otro tipo de técnicas como puede ser la reescritura de la *URL*. Esta técnica consiste en la modificación de la *URL* por parte del servidor para identificar cada *URL* con una sesión de un cliente.

3. AMPLIACIONES EN EL SERVIDOR

Si en el apartado anterior nos centrábamos en la posibilidad de personalizar nuestro cliente ligero engordándolo mediante diversas técnicas para dotar al servicio http de mayor funcionalidad, en este capítulo haremos lo propio con el servidor Web.

La mayor parte de las técnicas analizadas hasta el momento se basan en trasladar más o menos cantidad de conocimiento (*know-how*) desde el servidor hacia el cliente. Aunque esta técnica es muy interesante en numerosas ocasiones —sobre todo cuando se está buscando un control más fino de la interfaz y una respuesta más rápida al usuario—, en general resultan muy insuficientes para descargar al servidor de las sucesivas oleadas de demanda a la que debe enfrentarse a medida que se adentra en las turbulentas aguas del mundo de los negocios.

Las aplicaciones CGI nos permiten salir del paso, pero no están preparadas para soportar un gran número de transacciones, sobre todo porque el servidor no tiene el control de la ejecución de estas aplicaciones: no sabe realmente cómo están funcionando y no puede realizar optimizaciones de carga y ejecución.

Nuevamente, al no existir un estándar ampliamente reconocido, cada fabricante se lanzó a una carrera para proponer soluciones que pudieran lograr hacerse con la aceptación del mercado y, por lo tanto, convertirse con el tiempo en un estándar *de facto*. La realidad es que, aunque muchas ya han sido desechadas por este mercado, muchas otras se han afianzado en uno u otro contexto y, en la actualidad, conviven de una forma más o menos digna formando parte de las aplicaciones que operan sobre la Red.

3.1. EXTENSIONES DEL SERVIDOR

Una de las primeras ideas que se les ocurrió a los fabricantes de servidores Web consistió en permitir que sus clientes pudieran «construirse» su

servidor «a medida». Mediante determinados lenguajes nativos de la plataforma y siguiendo determinadas especificaciones podremos realizar filtros o extensiones al servidor Web, a fin de dotarle de funcionalidades de las que en principio carece. Obsérvese que la intención es la misma que en el caso de la interfaz CGI, pero aquí lo que se va a construir es una librería de acceso dinámico (.dll en Windows, .so en sistemas Unix). Esto significa que, a diferencia de los programas CGI que se instancian cada vez que son llamados, estas librerías, una vez cargadas en memoria, van a permanecer allí para atender cualquier petición posterior que se requiera de ellas (figura 3.1). Y, lo que es más interesante, se ejecutan como un módulo más, perfectamente entrelazado, del propio servidor.

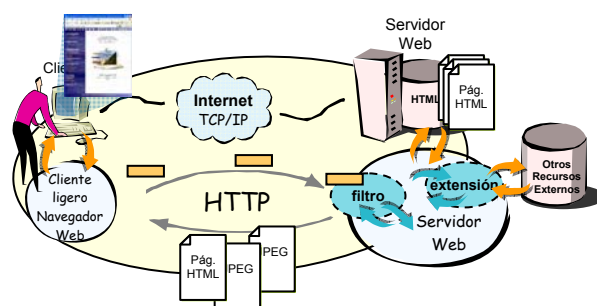


Figura 3.1. Escenario genérico para un modelo de filtros y extensiones del Servidor Web.

Aunque a nivel de desarrollo son equivalentes, funcionalmente un filtro se emplea para realizar un trabajo previo sobre la solicitud HTTP que llega al servidor Web mientras que las extensiones son invocadas (generalmente a través de estas mismas solicitudes HTTP) a posteriori para acceder de una forma eficiente a recursos o funcionalidades externas.

Para facilitar la labor, los diferentes fabricantes que han optado por esta técnica proporcionan una interfaz de programación de aplicaciones (API –*Application Programming Interface*) adecuada para su servidor Web. De esta forma, *Information Server API* (ISAPI) está orientado a *Internet Information Server* (IIS) de Microsoft y *Netscape API* (NSAPI), a los servidores Web de Netscape (Enterprise Netscape Server).

Por tanto, mediante esta tecnología ganamos sustancialmente en rendimiento y aprovechamiento de los recursos de la máquina. Sin embargo, nos encontramos con el gran inconveniente de estar cerrada, no sólo a una determinada plataforma, sino también a un determinado servidor Web e, incluso, a una determinada versión del mismo.

Otro inconveniente que ha contribuido a que esta técnica no sea muy empleada en la práctica es que la programación de estas API es bastante compleja o, como mínimo, con una curva de aprendizaje mayor que la que se requiere para realizar un programa CGI mediante un lenguaje y técnicas más o menos familiares para los desarrolladores.

3.2. PÁGINAS ACTIVAS

Esta tecnología está basada en incrustar código escrito en un lenguaje tipo *script* dentro de las propias páginas HTML con el fin de generar páginas dinámicas. Para ello, el servidor Web interpretará este código, accederá, si es necesario, a los recursos externos que precise y generará dinámicamente código en lenguaje HTML antes de servir la página al cliente (figura 3.2).

Esta forma de desarrollar aplicaciones basadas en Web es de las más aceptadas actualmente ya que existe una amplia alternativa de lenguajes más o menos dependientes de la plataforma y a que, debido a su concepción —código script incrustado en HTML—, se puede emplear cualquier editor HTML como herramienta de desarrollo.

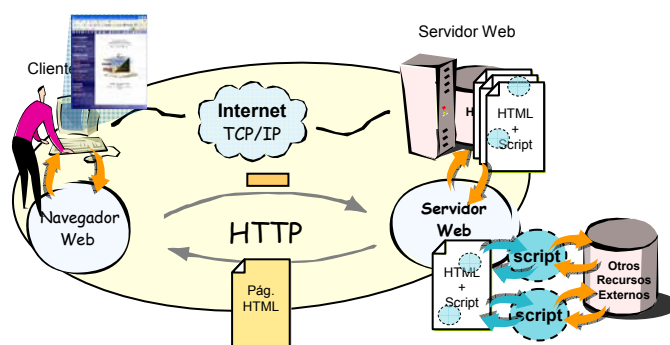


Figura 3.2. Escenario de desarrollo para una página activa.

A estas características que lo hacen popular, debemos unir otras de carácter más operativo, como el control que se obtiene entre la aplicación (el código script interpretado) y el servidor Web. Además, esta tecnología habilita el concepto de sesión de usuario, es decir, podemos conocer el estado de la conexión de un usuario en un momento determinado. Al mismo tiempo, gestiona bien los recursos externos, como es el caso de accesos a bases de datos, permite una gran flexibilidad para compartir los

mismos y para la gestión de la petición y respuesta Web. Finalmente, cómo el código se interpreta y ejecuta en el servidor, se mantiene la independencia del cliente Web.

A cambio, el desarrollador deberá aprender el lenguaje script correspondiente, el cual es propietario del servidor Web que lo ha de interpretar. Según esto, podemos encontrar diferentes alternativas como son: PHP, ASP, *LiveWire* o JSP.

A continuación analizaremos muy brevemente cada una de estas propuestas y mostraremos un sencillo ejemplo de cómo se puede presentar una página de saludo equivalente en cada una de ellas y cuya representación gráfica puede observarse en la figura 3.3.

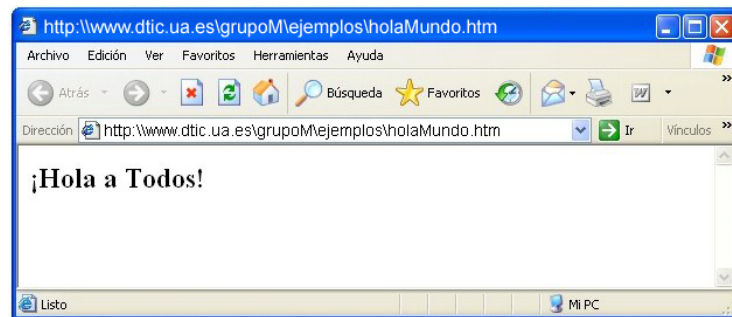


Figura 3.3. Representación por un navegador de la página HTML `holaMundo.htm`.

3.2.1. PHP

PHP es el acrónimo recursivo de «*PHP: Hypertext Preprocessor*» y puede considerarse como la propuesta pionera para el desarrollo de páginas dinámicas de una forma sistemática mediante el concepto de «páginas activas».

Puesto que esta característica no estaba incluida en los primeros servidores Web, se desarrolló mediante lenguaje PERL, como una serie de aplicaciones CGI que realizaban el preprocesamiento de páginas HTML con un código incrustado en etiquetas especiales al estilo de Perl. En la actualidad, diversos servidores Web, como Apache, ya incluyen esta característica de forma nativa.

La familiaridad de los desarrolladores en este tipo de lenguajes, la facilidad de uso y desarrollo, incluso, mediante un sencillo editor HTML, la potencia y el control que proporciona junto con su buena integración con el servidor Web, han contribuido notablemente a que esta tecnología sea

plenamente aceptada por la comunidad Web para el desarrollo de sitios dinámicos y aplicaciones Web en general.

En el listado 3.1 puede observarse una sencilla página HTML denominada `holaMundo.php` con código PHP incrustado dentro de la etiqueta especial `<?php ...?>`, cuya interpretación por un navegador Web convencional puede observarse en la figura 3.3.

```
<html>
<head>
</head>
<body>
<h2>
<?php echo "¡Hola a Todos!"; ?>
</h2>
</body>
</html>
```

Listado 3.1. Contenido de la *página activa* **holaMundo.htm** diseñada para mostrar en el navegador un mensaje de bienvenida generado dinámicamente mediante *php*.

3.2.2. LiveWire

Este lenguaje fue desarrollado originalmente por Netscape en 1995, con la finalidad de proporcionar un mayor dinamismo a las páginas Web.

Los servidores Web de *Netscape* e *Iplanet* incluyen LiveWire que representa un entorno de desarrollo que utiliza JavaScript ejecutado desde el servidor para crear aplicaciones (de manera similar a CGI, JSP y ASP). A diferencia del código JavaScript que se ejecuta en el cliente (en algún *Browser* como Navigator o Explorer) las aplicaciones LiveWire JavaScript se ejecutan en el propio servidor. Para ello, primero son compiladas y almacenadas en un archivo ejecutable binario. Durante el proceso de ejecución, Livewire genera código HTML de manera dinámica. Por supuesto este código también puede incluir código Javascript para el navegador. Finalmente, el resultado es enviado al cliente a través del servidor Web para que éste interprete y muestre el resultado.

En el listado 3.2 puede observarse una sencilla página HTML denominada `holaMundo.htm` con código Livewire Javascript incrustado entre las etiqueta `<server>` y `</server>`, cuya interpretación por un navegador Web convencional puede apreciarse en la figura 3.3. Es muy importante no confundir Livewire javascript con el código javascript que se ejecuta en el cliente y que, aunque es el mismo, iría entre las etiquetas `<script>` y `</script>`.

```

<html>
<head>
</head>
<body>
<h2>
<SERVER>
    write("¡Hola a Todos!")
</SERVER>
</h2>
</body>
</html>

```

Listado 3.2. Contenido de la *página activa* **holaMundo.htm** que muestra en el navegador un mensaje de bienvenida generado dinámicamente mediante *Liveware Javascript*.

3.2.3. ASP

Los servidores Microsoft implementan la tecnología de páginas activas bajo el nombre de ASP (Active Server Pages). ASP se apoya en *Visual Basic Script* (VBScript) como lenguaje de *scripting* y está orientada al servidor *Internet Information Server* (IIS).

Las páginas pueden ser generadas incrustando código en las páginas HTML. Este código es interpretado por el servidor y desde el mismo se puede, como en el resto de casos, acceder a recursos y funcionalidades del equipo servidor o, incluso, de equipos remotos, incluyendo bases de datos.

En el listado 3.3 se muestra una sencilla página HTML denominada `holaMundo.asp` con código ASP incrustado dentro de la etiqueta especial `<% ... %>`, cuya interpretación por un navegador Web convencional puede apreciarse en la figura 3.3.

```

<html>
<head>
</head>
<body>
<h2>
<% Response.Write "¡Hola a Todos!" %>
</h2>
</body>
</html>

```

Listado 3.3. Contenido de la *página activa* **holaMundo.asp** que muestra en el navegador un mensaje de bienvenida generado dinámicamente mediante *ASP*.

3.2.4. JSP

JavaServer Pages (JSP) es la propuesta desarrollada por Sun Microsystems para generar páginas Web de forma dinámica en el servidor. Esta tecnología está basada en Java y permite a los programadores generar dinámicamente HTML, XML o algún otro tipo de página Web. Para ello, como en el resto de casos, se inserta código, en este caso Java, dentro del contenido estático.

En versiones más recientes de la especificación se añadió la posibilidad de ejecutar *clases java* referenciadas desde la página HTML, mediante etiquetas denominadas «taglib». La asociación de las etiquetas con las clases java se declara en archivos de configuración escritos en XML.

La principal ventaja que presenta JSP frente a otras tecnologías es la posibilidad de integrarse con clases Java (`.class`). Esto permite organizar las aplicaciones Web en diferentes niveles, almacenando en clases java las partes que consumen más recursos o las que requieren más seguridad, manteniendo únicamente la parte encargada de proporcionar formato al documento HTML dentro del archivo JSP. Además, Java se caracteriza por ser un lenguaje que puede ejecutarse en cualquier sistema que incorpore una máquina virtual de java (JVM). Teniendo en cuenta que en la actualidad se pueden encontrar implementaciones de JVM para casi cualquier plataforma, la portabilidad de las aplicaciones es realmente considerable.

Como ocurría en el caso de LiveWire, antes de que el servidor Web pueda ejecutar el código java, debe compilarlo, generando un objeto «servlet» (ver apartado 3.3.1). Aunque este proceso ralentiza inicialmente la ejecución, en posteriores invocaciones no sólo no se producirá retardo, sino que se ejecutará más rápido y podrá disponer del API de Java en su totalidad.

En el listado 3.4 se presenta el código fuente de una sencilla página HTML denominada `holaMundo.jsp` con código JSP incrustado dentro de la etiqueta especial `<% ... %>`, cuyo resultado, interpretado por un navegador Web convencional, puede verse en la figura 3.3.

```
<html>
<head>
</head>
<body>
<h2>
<%= "¡Hola a Todos!" %>
</h2>
</body>
</html>
```

Listado 3.4. Contenido de la página activa **holaMundo.jsp** que muestra en el navegador un mensaje de bienvenida generado dinámicamente mediante JSP.

3.2.5. ASP .NET

ASP.NET es la plataforma unificada de Microsoft para el desarrollo Web que proporciona a los desarrolladores los servicios necesarios para crear aplicaciones Web empresariales.

Microsoft ha visto en la estrategia de separar la lógica de la aplicación del contenido estático y de los aspectos de presentación desarrollada por SUN, su más directa competidora, una seria amenaza. Esta es la principal motivación que le ha llevado a incorporar en su plataforma .NET un lenguaje de *scripts ASP.NET* que permite ser integrado con clases .NET (ya estén desarrolladas en C++, VisualBasic o C#) del mismo modo que JSP se integra con *clases Java*.

Esta nueva tecnología se ha desarrollado como parte de la estrategia .NET para el desarrollo Web, con el objetivo de resolver las limitaciones de ASP y posibilitar la creación de *software como servicio*.

Para distinguir unas versiones ASP de otras, las versiones «pre-.NET» se denominan actualmente «ASP clásico».

3.3. COMPONENTES EN LA PARTE DEL SERVIDOR

Un componente en la parte del servidor o un «servicio ligero» conceptualmente no difiere mucho de los componentes en la parte del cliente o aplicaciones ligeras tratadas en el apartado 2.3. En cualquier caso, la diferencia fundamental radica en que este software ahora no se transfiere al cliente, sino que se ejecuta directamente en el propio servidor Web.

Según esta definición, los componentes en la parte del servidor también podrían confundirse con las aplicaciones CGI analizadas en el

punto 1.3, sin embargo, también existe una diferencia notable con respecto a éstas y es que su ejecución sí se desarrolla dentro del contexto del propio servidor Web —recordemos que en el caso de las aplicaciones CGI, dicha ejecución era gestionada por el sistema operativo en un contexto diferente al del servidor Web—. Esto dota al servidor y a la propia aplicación de un control mutuo que permite superar las limitaciones del modelo CGI. Un servidor Web capaz de ejecutar este tipo de componentes se denomina también, «contenedor Web».

Funcionalmente, los componentes software en la parte del servidor se pueden considerar *piezas de software* creadas para encapsular una determinada *lógica de negocio*, un determinado servicio o, incluso, ser empleados como almacén de datos. Estos componentes se utilizan en la construcción de aplicaciones como si fueran piezas de un mecano y están diseñados para ser reutilizables.

Si bien con las tecnologías revisadas hasta el momento se pueden desarrollar básicamente los mismos servicios, a medida que estos servicios o aplicaciones Web adquieren relevancia, no están preparadas para hacerlo con un rendimiento adecuado o para poder aplicar técnicas de ingeniería del software que faciliten el desarrollo inicial y su posterior mantenimiento. Con la apuesta firme del mundo de los negocios por las tecnologías Web para desarrollar sus aplicaciones, han ido surgiendo y evolucionando diversas propuestas encaminadas a suplir estas necesidades operativas, como *servlets*, *java beans* o *COM*. Durante los siguientes apartados iremos analizando las opciones disponibles más extendidas.

3.3.1. Servlets

Los *servlets* son programas escritos en Java, se pueden invocar desde un cliente, de forma similar a como se invocan los programas CGI; se ejecutan en el seno del servidor Web, que actúa como un «contenedor de componentes»; realizan una determinada tarea; y generan una salida (página Web dinámica) que es recogida por el servidor Web y posteriormente reenviada al navegador que realizó la invocación de este componente.

Teniendo en cuenta que un *servlet* es un objeto software que se invoca externamente, representa un modelo mucho más cercano al *modelo CGI* que al *modelo de páginas activas*, donde este objeto se encuentra incrustado dentro de código HTML estático. Sin embargo, la diferencia fundamental con respecto al modelo CGI clásico se encuentra en que un

servlet se ejecuta dentro del contexto, y por lo tanto del control, del servidor o, ahora, contenedor Web (como por ejemplo: *Apache Tomcat*).

En el listado 3.5 se presenta el código fuente de un servlet, cuya interpretación por un navegador Web convencional puede verse en la figura

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class HolaMundoServlet extends HttpServlet {

    public void init(ServletConfig conf)
        throws ServletException {
        super.init(conf);
    } // de initServlet

    public void service(HttpServletRequest req, \
        HttpServletResponse res)
        throws ServletException, IOException {
        res.setContentType("text/html");
        PrintWriter salida = res.getWriter();

        salida.println("<html>");
        salida.println("<head>");
        salida.println("</head>");
        salida.println("<body>");
        salida.println("<h2>¡Hola a Todos!</h2>");
        salida.println("</body>");
        salida.println("</html>");
    } // de service
} // de HolaMundoServlet
```

3.3.

Listado 3.5. Contenido de la página activa `holaMundo.js` que muestra en el navegador un mensaje de bienvenida.

Sus características y método de trabajo son los siguientes:

- **Portabilidad.** Al estar escritos en Java, los servlets garantizan la portabilidad entre diferentes plataformas que dispongan de la *Máquina Virtual Java*. También existe portabilidad entre los diferentes servidores Web, ya que el API Servlet define una interfaz estándar entre el servlet y el servidor Web que es independiente del fabricante. Por lo tanto, no hay especificaciones dependientes del

servidor, como ocurre con la interfaz ISAPI/NSAPI o con las *páginas activas*.

- **Rendimiento.** Al igual que las extensiones del servidor ISAPI/NSAPI y a diferencia de CGI, los servlets son instanciados una única vez, pudiendo, a partir de entonces, atender diferentes peticiones. La instanciación del servlet se puede producir cuando tiene lugar la primera petición o estar ya instanciado desde la propia inicialización del servidor Web.
- **Concepto de sesión.** El servlet puede mantener información acerca de la sesión de usuario, variables o recursos —como por ejemplo: una conexión a la base de datos— entre diferentes conexiones dirigidas al mismo servlet e, incluso, entre varios de ellos.
- **Software distribuido.** Se pueden cargar indiferentemente y de forma transparente tanto desde un disco local como desde una dirección remota, respondiendo a la nueva filosofía de software distribuido. Los servlets pueden comunicarse entre sí y, por tanto, es posible una reasignación dinámica de la carga de proceso entre diversas máquinas, es decir, un servlet podría pasarle trabajo a otro servlet residente en otra máquina.
- **Ventajas inherentes a Java.** Además de la portabilidad, se hereda otras ventajas de este lenguaje idóneo para desarrollos en Internet, como es la orientación a objetos, la modularidad y la reutilización. Por otra parte, la responsabilidad de la gestión de memoria, tarea a cargo del programador en otros lenguajes, recae en el propio Java, facilitando la recolección automática de memoria no utilizada e impidiendo el acceso directo a la misma.
- **Multithread.** Dado que los servlets pueden manejar múltiples peticiones concurrentemente, es posible implementar aplicaciones cooperativas, como por ejemplo una aplicación de videoconferencia.

3.3.2. JavaBeans

Un *JavaBean*, o sencillamente un «bean», representa un modelo de componentes software especificado por Sun Microsystems dentro de su plataforma *Java 2 Platform, Standard Edition* (J2SE) para la construcción de aplicaciones Java. Estos componentes están pensados para poder ser reutilizados fácilmente en la creación de aplicaciones.

A diferencia de los servlets, los beans están diseñados para actuar de forma autónoma, ejecutándose dentro del contexto de un contenedor más genérico que el servidor Web. Se podría decir que esta propuesta marca el

inicio de la imparable estrategia por desacoplar la lógica de negocio, no sólo del código HTML, sino también del propio servidor Web. Este nuevo contexto se denomina «Servidor de Aplicaciones». Sin embargo, dejaremos su desarrollo para el próximo apartado, en el que analizaremos propuestas mucho más específicas y maduras, inspiradas desde el comienzo para proporcionar soporte a este nuevo concepto.

Los JavaBeans, además de exponer sus propiedades para poder ser personalizados y utilizar los eventos para comunicarse con el sistema o con otros beans, también utilizan la serialización de objetos Java para soportar la persistencia —lo que les permite guardar su estado y restaurarlo posteriormente—. Además, sus métodos, que no son diferentes de otros métodos Java, pueden ser invocados desde otros beans o desde páginas JSP.

Los beans han sido diseñados para que todas las claves de su API puedan ser entendidas y gestionadas por herramientas de desarrollo visual, incluyendo el soporte para eventos, las propiedades y la persistencia. Son los llamados *beans visuales*, y están íntimamente relacionados con la interfaz gráfica de usuario.

En cualquier caso, en el contexto del servidor de aplicaciones hablamos de los *beans no-visuales*, que pueden ser utilizados de forma programática, conociendo su funcionalidad e interfaz. En este grupo se distinguen dos tipos por su funcionalidad: los que contienen una determinada lógica de negocio y los que sirven como almacén de datos a los que se accederá posteriormente, cuando se necesite la información que albergan, en otro punto de la aplicación Web.

Aunque los beans individuales pueden variar ampliamente en funcionalidad, desde los más simples a los más complejos comparten las siguientes características:

- **Introspección (introspection):** permite que la herramienta de programación o IDE pueda analizar cómo trabaja el bean.
- **Personalización (customization):** el programador puede alterar la apariencia y la conducta del bean y se permite cambiar los valores de las propiedades del bean para personalizarlo.
- **Eventos (events):** informa al IDE de los sucesos que puede generar en respuesta a las acciones del usuario o del sistema, y también los sucesos que puede manejar.
- **Persistencia (persistente):** un bean tiene que tener persistencia, es decir, implementar el interfaz *Serializable*.

Cuando el IDE carga un bean, usa el mecanismo denominado *reflection* para examinar todos los métodos, fijándose en aquellos que

empiezan por **set** y **get**. El IDE añade las propiedades que encuentra a la hoja de propiedades para que el programador personalice el bean.

3.3.3. Objetos OLE, ActiveX y COM

Microsoft fue uno de los pioneros en poner en práctica las ventajas del concepto de objeto implementando el intercambio dinámico de datos (*Dynamic Data Exchange* —DDE) desde las primeras versiones de Windows. Como evolución de este concepto desarrolló un sistema de objeto distribuido junto con un protocolo que denominó (*Object Linking and Embedding* —OLE 1.0). Mientras que DDE se limitaba a transferir una cantidad concreta de información entre dos aplicaciones, OLE fue capaz de mantener enlaces activos entre dos documentos o incluso incrustar un tipo de documento dentro de otro.

El principal cometido de OLE era la gestión de documentos compuestos, pero también se podía emplear para transferir datos entre aplicaciones mediante la técnica de «arrastrar y soltar» (*drag and drop*) y operaciones del portapapeles (*clipboard*). Sin embargo, el concepto que popularizó esta tecnología en el entorno Web era el de incrustar (*embedding*) objetos multimedia (animaciones flash, archivos de vídeo, etc.) en páginas Web, dentro del código HTML.

El problema es que OLE 1.0 es una tecnología del lado del cliente, pues el objeto incrustado debía estar situado en el equipo en el que se iba a trabajar.

Posteriormente OLE 1.0 evolucionó a OLE 2.0 que, por motivos probablemente comerciales, pasó a denominarse ActiveX. Además, y para confundir más aun las cosas, OLE 2.0 se volvió a implementar sobre COM (*Component Object Model*).

COM representa la primera arquitectura software de componentes de Microsoft, diseñada para facilitar la reutilización de software y la interoperatividad de los componentes mediante la comunicación entre procesos y la creación dinámica de objetos, en cualquier lenguaje de programación que soporte dicha tecnología.

El término COM es a menudo usado en el mundo del desarrollo de software como un término que abarca tecnologías anteriores a las que había absorbido o de las que había evolucionado: OLE, *OLE Automation* o ActiveX; pero también podemos encontrar referencias a otras tecnologías que han evolucionado a partir de ella, como DCOM y COM+, que estudiaremos con más detalle en el capítulo 4.

4. SERVIDOR DE APLICACIONES

Hasta el momento, la mayor parte de las tecnologías que se han ido introduciendo no han hecho otra cosa que engordar y engordar el servidor Web, cargándolo con todo el trabajo adicional a base de parches o añadidos de diversa naturaleza —justo en contra de la idea original, donde un protocolo muy básico y una sencilla aplicación servidora gestionaban la presentación de contenidos a, potencialmente, cualquier equipo que se pudiera conectar a la red y dispusiera de un sencillo navegador Web.

A medida que las primeras y sencillas páginas Web se han transformado en sofisticadas «aplicaciones Web» que proporcionan soporte a organizaciones de cualquier tamaño, parece una mejor opción devolver este escenario a su estado inicial y buscar soluciones más ajustadas a los nuevos requerimientos. Según esto, para gestionar todas las posibles tareas de índole dinámica que un servidor Web deba realizar, se propone la colocación, *detrás de él*, un nuevo sistema especialmente concebido para resolverlas con solvencia y liberar al servidor Web de toda carga extra. Es lo que se conoce como «servidor de aplicaciones».

Este enfoque no implica que no se vayan a poder seguir empleando todas las tecnologías anteriormente estudiadas, sino que la tendencia será ir migrando las aplicaciones hacia la utilización del servidor de aplicaciones a medida que vayan creciendo los requerimientos de escalabilidad, eficiencia y adaptabilidad, por mencionar algunos de ellos, de las nuevas aplicaciones Web.

La filosofía general de funcionamiento del nuevo tándem servidor Web más servidor de aplicaciones es la siguiente: las peticiones realizadas al servidor Web que deban generar contenido estático serán despachadas por el propio servidor Web, como de costumbre, pero las peticiones que deban generar contenido dinámico serán delegadas en el servidor de aplicaciones; éste interactuará con los recursos que se precisen, ejecutará la

En un entorno de estas características seguimos interesados en perfeccionar el modelo de componentes pero, sobre todo, estamos más que interesados en que la complejidad de las infraestructuras que tienen que servir de soporte quede oculta para las aplicaciones y para los desarrolladores, y que su gestión no se convierta en un verdadero calvario para los administradores de sistemas.

De igual forma que en un equipo aislado es el sistema operativo el responsable de ocultar a las aplicaciones los detalles concretos de los dispositivos y componentes físicos, el conjunto de tecnologías y servicios que proporcionan la transparencia deseada (ver figura 4.2), ya no sólo sobre un único equipo, sino sobre todo un conjunto de equipos, así como de la propia red de comunicaciones e, incluso, de los sistemas operativos de cada una de ellos, lo denominamos «middleware».

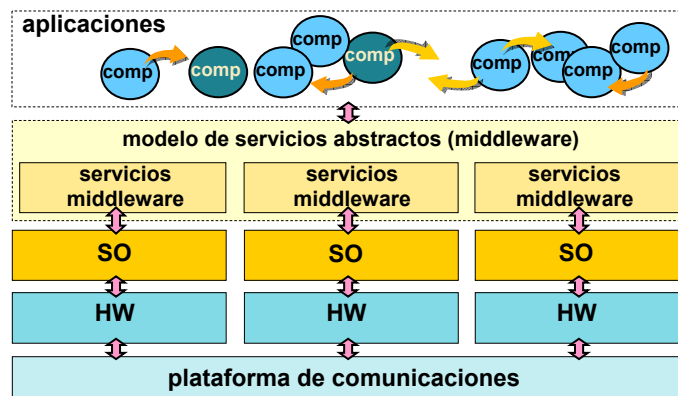


Figura 4.3. Modelo general de capas del sistema con middleware.

4.1. MIDDLEWARE

El middleware se define como el nivel lógico del sistema que proporciona una abstracción sobre la infraestructura que le da soporte a las aplicaciones, dotando de transparencia de ubicación e independencia de los detalles del hardware de los computadores, del sistema operativo, de la red de interconexión y de los protocolos de comunicaciones (figura 4.3) e, incluso, del lenguaje de programación utilizado para su desarrollo. Desde el punto de vista del programador de aplicaciones, el middleware establece un modelo de programación sobre bloques básicos arquitectónicos, utilizando protocolos basados en mensajes para proporcionar abstracciones de nivel superior.

En cualquier caso, el middleware se puede estudiar desde dos puntos de vista bien diferenciados: desde el punto de vista de la infraestructura y servicios que lo conforman o desde el punto de vista del desarrollador de aplicaciones. Veamos a continuación cada uno de ellos.

4.1.1. Infraestructura de servicios

Desde el punto de vista de la propia infraestructura que conforma el *middleware*, éste viene definido por los elementos que componen el núcleo de servicios, ubicado entre el sistema operativo local y las aplicaciones, y en la que resaltan los siguientes elementos (figura 4.4): el modelo de representación de datos, el modelo de comunicaciones —en el que se incluye el protocolo de comunicaciones y el modelo de invocación de métodos remotos— y los servicios fundamentales que proporciona la plataforma.

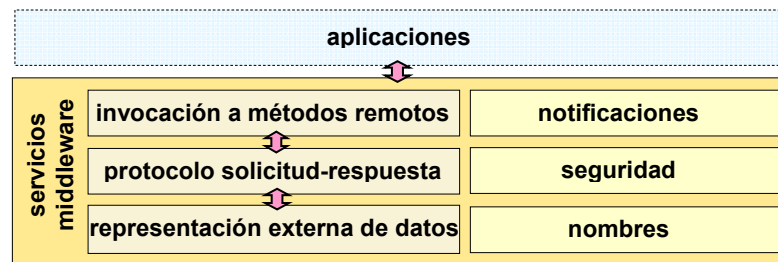


Figura 4.4. Alguno de los principales componentes que forman la capa de servicios middleware que proporciona soporte y transparencia a las aplicaciones.

4.1.1.1. Modelo de representación de datos

El modelo de comunicación proporciona los mecanismos para que dos aplicaciones potencialmente ubicadas en equipos diferentes puedan interactuar entre sí de forma transparente. Al mismo tiempo, estos mecanismos determinan cómo se deberá establecer la comunicación y cómo tendrán que prepararse cliente y servidor para poder realizarla.

Independientemente de la forma de comunicación utilizada, las estructuras de datos y los atributos de los objetos deben ser convertidos en una secuencia de *bytes* antes de su transmisión y reconstruidos posteriormente en el destino. Para hacer posible que dos computadores puedan intercambiar información, deben acordar previamente un esquema común o incluir la especificación del formato de emisión en el propio mensaje.

Al estándar acordado para la representación de los valores y estructuras de datos a transmitir en los mensajes se denomina *representación externa de datos* —por ejemplo, CDR de CORBA.

4.1.1.2. Modelo de comunicación

El protocolo petición-respuesta define el nivel adecuado para establecer una comunicación típica según la arquitectura cliente/servidor en sistemas distribuidos basados en UDP o TCP. El propio mensaje de respuesta se considera como un reconocimiento del mensaje de petición, evitando de este modo las sobrecargas de los mensajes de reconocimiento adicionales.

Sobre esta capa de protocolo se construyen los modelos de comunicación de llamada a procedimiento remoto (RPC) y el modelo de invocación a un método remoto (RMI). RPC permite a programas cliente invocar procedimientos pertenecientes a programas servidor que generalmente ejecutan en computadores distintos. Este modelo evoluciona, posteriormente, permitiendo que objetos de diferentes procesos se comuniquen con los métodos de otros objetos.

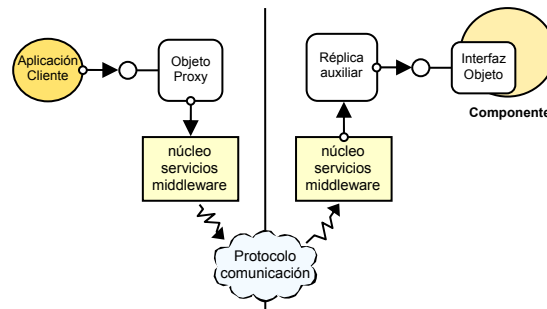


Figura 4.5. Arquitectura de comunicaciones middleware.

En la figura 4.5 se puede observar una arquitectura de comunicaciones más o menos genérica de un middleware. Cuando se crea un componente que debe ser accedido de forma remota, debe crearse también un objeto *proxy* de la interfaz de dicho componente. De esta forma, el middleware puede presentar a la aplicación cliente una interfaz uniforme, con independencia de dónde se encuentre realmente el componente al que desea acceder, encargándose de resolver todos los problemas de redirección de las solicitudes y respuestas, de localización de los componentes o de que los componentes se encuentran cargados en memoria y listos para responder.

4.1.1.3. Servicios comunes

Para las aplicaciones que se ejecutan apoyándose en el middleware, éste representa fundamentalmente un conjunto de servicios a los que pueden acceder o a partir de los cuales pueden acceder a otros servicios y recursos. El mecanismo de comunicación estudiado en el punto anterior es uno de estos servicios, quizá uno de los más importantes, pero no el único. De hecho, cada fabricante propondrá un conjunto de servicios que estime convenientes y que no tienen por qué coincidir de un middleware a otro. Por suerte, existe una serie de servicios que se pueden encontrar de una forma u otra en las diferentes propuestas de middleware para componentes software distribuidos. A continuación se realiza una descripción breve de los más comunes: servicios de nombres, de eventos y notificaciones y de seguridad.

Servicio de nombres

Su objetivo es almacenar los atributos de los objetos en un sistema distribuido —nombre y dirección entre otros—, devolviendo esos atributos cuando se realiza una búsqueda sobre cierto objeto. Los principales requisitos que debe poseer un servicio de nombres son: habilidad para manejar un número arbitrario de nombres, persistencia, alta disponibilidad y tolerancia a fallos. Ejemplos de este tipo de servicio son: X.500, Jini, DNS, LDAP, etc.

Servicio de eventos y notificaciones

Este servicio extiende el modelo local de eventos al permitir que varios objetos en diferentes ubicaciones puedan ser notificados de los eventos que tienen lugar en un objeto. Emplean el paradigma *del tablón de anuncios*, en el que un objeto que genera eventos publica el tipo de eventos que ofrece para su observación. Los objetos que desean recibir notificaciones de otro objeto se suscriben a los tipos de eventos que les interesan.

Servicio de seguridad

Los mecanismos de seguridad se basan esencialmente en la criptografía de clave pública y de clave secreta. Los recursos se protegen mediante mecanismos de control de acceso. Se pueden mantener los derechos de acceso en listas de control (ACL) asociadas a conjuntos de objetos. Ejemplos de este servicio son: el protocolo de autenticación Needham-Schroeder, Kerberos, SSL y Millicent.

4.1.2. Modelo de programación y de componentes

Desde el punto de vista del diseñador de aplicaciones, el middleware representa un modelo de programación al que le incumbe definir nítidamente la estructura que deben tener los componentes software a diseñar, así como los mecanismos que deben emplearse para acceder a los recursos y a los servicios que proporciona la plataforma.

Los componentes software se pueden considerar piezas de software diseñadas para ser reutilizadas en diferentes planos: reutilización de código, de funcionalidad, de tipos. Los componentes software distribuidos, además, hacen especial hincapié en su ejecución transparente sobre entornos distribuidos. Están creados para encapsular una determinada lógica de negocio, un determinado servicio (como puede ser acceso a una determinada arquitectura) o, incluso, ser empleados como almacén de datos. Estos componentes se utilizan en la construcción de aplicaciones como si fueran piezas de un mecano.

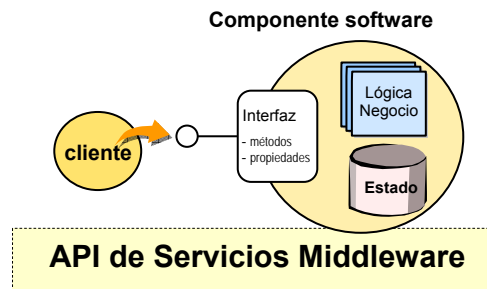


Figura 4.6. Estructura básica de un componente software y su relación con la plataforma.

Todas estas características resultan fundamentales para diseñar, desarrollar y mantener las actuales aplicaciones Web, sobre todo en el ámbito empresarial. Gracias a ellas se pueden aplicar técnicas de ingeniería del software sofisticadas y se facilita enormemente la adaptación de la aplicación a la situación cambiante del entorno. Una aplicación distribuida basada en el modelo de middleware permite una asignación de recursos muy ajustada a medida que éstos vayan necesitándose y, siempre que los cambios puedan localizarse en unos componenetes software determinados, su modificación tiene un impacto relativamente escaso sobre el global de la aplicación.

Un componente software debe poseer una estructura bien definida (ver fig. 4.6): una *interfaz* en la que se definen los métodos y propiedades que soporta el componente y a través de la cual los clientes pueden acceder;

una *lógica de negocio* o lógica de aplicación que recoge la funcionalidad o el comportamiento del componente, tanto la que muestra al exterior a través de su interfaz, como la que presenta internamente; y un *estado* que le permite almacenar información que permite que dos instancias de un mismo componente puedan adquirir su propia entidad.

Por otra parte, el modelo de programación también determina cómo accederán estos componentes software a los servicios que ofrece el middleware y que, en este caso, consiste en la provisión de una *interfaz de programación (API) de servicios*.

4.1.2.1. Estructura de un componente

Un sistema orientado a objetos consta de un conjunto de componentes que interaccionan entre sí, cada uno de los cuales consiste en un conjunto de propiedades y un conjunto de métodos. Un objeto se comunica con otro objeto invocando sus métodos, generalmente pasándole argumentos y recibiendo resultados. Así, cada componente se puede estructurar en tres elementos: una interfaz, una lógica de aplicación o de negocio y un estado:

Interfaz

La interfaz de un componente proporciona una definición de las firmas de sus métodos —los tipos de sus argumentos, valores devueltos, excepciones, etc.— sin especificar su implementación. Establece, asimismo, qué métodos y propiedades están accesibles para el resto de componentes y objetos. Por supuesto, nada impide que un mismo objeto implemente distintas interfaces a la vez.

Cada componente se implementa de forma que oculta todo su estado y funcionalidad, excepto aquél que se hace visible a través de su interfaz en términos de *propiedades* y *métodos*, respectivamente. Esto permite modificaciones en la implementación de la lógica del componente o de su estructura interna —siempre que su interfaz se mantenga inalterada— con un impacto mínimo sobre el resto de la aplicación de la que forma parte.

Para que distintos componentes —implementados por distintos equipos de desarrollo y, posiblemente, con diferentes lenguajes de programación— puedan interactuar entre sí es necesario que sus interfaces estén definidas de forma homogénea. Para facilitar esto, el modelo de programación introduce los lenguajes de definición de interfaces (IDL) que proporcionan una notación específica en la que, por ejemplo, los parámetros de un método se describen como de entrada o salida y utilizando su propia especificación de tipos.

Estado

El estado de un componente consta de los valores de sus variables de instancia. En el paradigma de la programación orientada a objetos, el estado de un programa se encuentra fraccionado en partes separadas, cada una de las cuales está asociada con un objeto. El estado de un objeto está accesible sólo para los métodos del objeto, es decir, que no es posible que métodos no autorizados actúen sobre su estado.

Algunas implementaciones de middleware, como CORBA, permiten empaquetar y almacenar los objetos junto con su estado en un momento dado —en los llamados almacenes de objetos persistentes—, de forma que métodos de otros objetos puedan activarlos por invocación a través de su interfaz. En general, se almacenan en cualquier momento en el que se encuentren en un estado consistente, con lo que dotan al sistema de tolerancia a fallos.

Lógica de negocio o comportamiento

El comportamiento define la funcionalidad del componente a través de una serie de métodos que recogen la lógica de aplicación o de negocio que encapsula y a la cual se puede acceder, en caso de estar disponible para otros componentes, a través de las interfaces definidas.

4.1.2.2. Acceso a los servicios

Una plataforma middleware debe proporcionar un mecanismo que permita a los componentes software acceder a los servicios que proporciona. El método más habitual, sobre todo con el objetivo de dotar a los desarrolladores de una independencia adecuada con respecto a los lenguajes y herramientas de programación que utilicen, es a través de interfaces de programación de aplicaciones (API).

4.2. PLATAFORMAS ACTUALES

Aunque desde el punto de vista más amplio podríamos definir un *servidor de aplicaciones* como «uno o más servidores de red, dedicados a ejecutar ciertas aplicaciones software, de forma que sus componentes pueden comunicarse entre sí de forma transparente», el término también hace referencia al propio software instalado en estos servidores cuyo objetivo es servir de soporte para la ejecución de las aplicaciones mencionadas. Es por esta función de soporte por la que este software de apoyo se denomina también *plataforma middleware*.

En la práctica, una de las plataformas que más ha contribuido a esta visión es J2EE de *Sun Microsystems*, razón por la cual en muchos entornos se considera un sinónimo de *servidor de aplicaciones*. Sin embargo, puesto que J2EE en realidad es un conjunto de especificaciones, en la actualidad podemos encontrar múltiples alternativas de implementación de otros fabricantes y, lo que es más interesante, otras especificaciones válidas.

Los servidores de aplicación típicamente incluyen el middleware que les permite intercomunicarse con diferentes servicios de forma segura y proporcionan a los desarrolladores un API que aísla a las aplicaciones del sistema operativo del servidor y brindan soporte a una gran variedad de estándares, tales como HTML, XML, IIOP, JDBC, SSL, etc., que facilitan la interoperatividad de elementos heterogéneos y la conexión a una gran variedad de fuentes de datos, sistemas y dispositivos.

A continuación se abordan algunas de las plataformas que están más extendidas en la actualidad y que proporcionan estos servicios middleware.

4.2.1. Plataforma J2EE

J2EE son las siglas de *Java 2 Enterprise Edition*, es decir, la edición empresarial del paquete Java creada y distribuida por *Sun Microsystems*. Comprende un conjunto de especificaciones y funcionalidades orientadas al desarrollo de aplicaciones empresariales. Debido a que J2EE no deja de ser la definición de un estándar, existen otros productos desarrollados a partir de la misma, aunque no exclusivamente.

Algunas de las funcionalidades más importantes que aporta esta plataforma son las siguientes:

- Proporciona acceso a bases de datos (JDBC).
- Disponibilidad de implementaciones de la plataforma por diferentes fabricantes: BEA, IBM, Oracle, Sun, y Apache Tomcat entre otros.
- Permite la utilización de directorios distribuidos (JNDI).
- Proporciona acceso a la invocación de métodos remotos (RMI, CORBA).
- Dispone de funciones de correo electrónico (JavaMail).
- Aplicaciones Web (JSP y Servlet).
- Puede emplear componentes software como: Beans, objetos CORBA, etc.

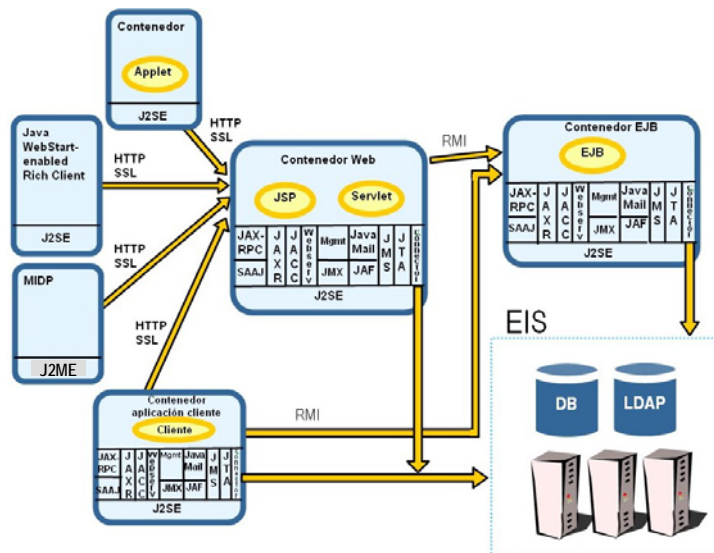


Figura 4.7. Arquitectura J2EE.

La Plataforma J2EE desarrolla el concepto de contenedores a partir de la tecnología *Java 2 Estándar Edition* (J2SE) propuesta también por Sun Microsystems, integra los elementos existentes (JSP, applet, servlet, Javabeans), lo extiende al de *Enterprise Java Bean* (EJB) como modelo de componentes software distribuidos y basa la comunicación en el estándar RMI (*Remote Method Invocation*) tratado en temas anteriores.

En la figura 4.7 se presenta un diagrama con los componentes que forman parte de esta plataforma. Se puede observar cómo los diferentes objetos del sistema (JSP, servlet, Applet o EJB) se ejecutan en *contenedores* concretos: de aplicación cliente, Web, EJB y navegador Web compatible. Todos los contenedores están basados en J2SE como núcleo básico. Se puede encontrar una excepción en el caso de dispositivos móviles o sistemas embebidos de muy poca capacidad, para los cuales se ha tenido que proponer y desarrollar un núcleo de J2SE mucho más compacto y ligero que se denomina *Java 2 Mobile Edition* (J2ME). Sobre este núcleo, se agregan aquellos servicios que puedan resultar útiles a las aplicaciones que se ejecuten en cada contenedor (como JAX-RPC, SAAJ, JTA, JAF o JMX; detallados más adelante dentro de este mismo punto) junto con una serie de conectores que facilitan el acceso de las aplicaciones a sistemas de información empresarial (bases de datos, sistemas heredados, etc.).

4.2.1.1. Servicios midlleware en J2EE

J2EE propone el acceso a los servicios que define a través de una interfaz de programación. Las principales APIs de la plataforma J2EE de SUN son las siguientes:

- **JCA** Arquitectura que para interactuar con una variedad de EIS, incluye ERP, CRM y otra serie de sistemas heredados.
- **JDBC** Acceso a base de datos relacionales.
- **JTA** Manejo y la coordinación de transacciones a través de EIS heterogéneos.
- **JNDI** Acceso a información en servicios de directorio y servicios de nombres.
- **JMS** Envío y recepción de mensajes.
- **JMail** Envío y recepción de correo.
- **JIDL** Mecanismo para interactuar con servicios CORBA.

Por supuesto, se dispone de muchos otros APIs orientados a aspectos como: tratamiento de XML, integración con sistemas heredados utilizando Servicios Web, etc.

4.2.1.2. Enterprise JavaBeans

Los EJBs proporcionan un modelo de componentes software distribuido normalizado para el lado del servidor. El objetivo de los Enterprise beans es dotar al programador de un modelo que le permita abstraerse de los problemas generales de una aplicación empresarial (conurrencia, transacciones, persistencia, seguridad, ...) para centrarse en el desarrollo de la lógica de negocio en sí. El hecho de estar basado en componentes nos permite que éstos sean flexibles y, sobre todo, reutilizables.

No hay que confundir a los *Enterprise JavaBeans* con los *JavaBeans*. Los JavaBeans también son un modelo de componentes creado por *Sun Microsystems* para la construcción de aplicaciones, pero no pueden utilizarse en entornos de objetos distribuidos al no soportar nativamente la *invocación a métodos remotos* (RMI).

El API Enterprise JavaBeans (EJB) permite escribir componentes software en Java, lo que les confiere la funcionalidad de portabilidad e independencia de la plataforma, a diferencia de los componentes ActiveX o COM+, propios de los sistemas Windows, o los VCL, asociados a los entornos de desarrollo Delphi, por citar los más conocidos.

Existen tres tipos de EJBs:

- **EJBs de Entidad** (*Entity EJBs*): su objetivo es encapsular los objetos de lado de servidor que almacenan los datos. Los EJBs de

entidad presentan la característica fundamental de la persistencia, ya sea gestionada por el propio contenedor (CMP) o por el bean (BMP).

- **EJBs de Sesión** (*Session EJBs*): gestionan el flujo de la información en el servidor. Generalmente sirven a los clientes como una fachada de los servicios proporcionados por otros componentes disponibles en el servidor. Puede haber dos tipos: «con estado» (*stateful*) o «sin estado» (*stateless*).
- **EJBs dirigidos por mensajes** (*Message-driven EJBs*): los únicos beans con funcionamiento asíncrono. Usando el *Java Messaging System* (JMS), se suscriben a un «tópico» (*topic*) o a una «cola» (*queue*) y se activan al recibir un mensaje dirigido a dicho tópico o cola. No requieren de su instanciación por parte del cliente.

4.2.1.3. Funcionamiento de un Enterprise JavaBean

Los EJBs se disponen en un *contenedor EJB* dentro del *servidor de aplicaciones*. La especificación describe cómo el EJB interactúa con su contenedor y cómo el código cliente interactúa con la combinación del EJB y el contenedor.

Cada EJB debe facilitar una clase de implementación Java y dos interfaces Java. El contenedor EJB creará instancias de la clase de implementación Java para facilitar la implementación EJB. Las interfaces Java son utilizadas por el código cliente del EJB. Las dos interfaces, conocidas como «interfaz home» e «interfaz remota», especifican las firmas de los métodos remotos del EJB.

El servidor invocará a un método correspondiente a una instancia de la clase de implementación Java para manejar la invocación del método remoto. Las llamadas a métodos en la interfaz remota se remiten al método de implementación correspondiente del mismo nombre y argumentos.

Dado que se trata simplemente de interfaces Java y no de clases concretas, el contenedor EJB genera clases para esas interfaces que actuarán como intermediarias —o como un «proxy»— entre los clientes y los componentes. El cliente invoca un método en los proxies generados que, a su vez, sitúa los argumentos método en un mensaje y envía dicho mensaje al servidor EJB. Los proxies usan RMI-IIOP para comunicarse con el servidor EJB.

RMI es el mecanismo que permite realizar invocaciones a métodos de objetos remotos situados en distintas (o en la misma) máquinas virtuales de Java, compartiendo así recursos y carga de procesamiento a través de varios sistemas.

La arquitectura RMI puede verse como un modelo de cuatro capas (ver figura 4.8):

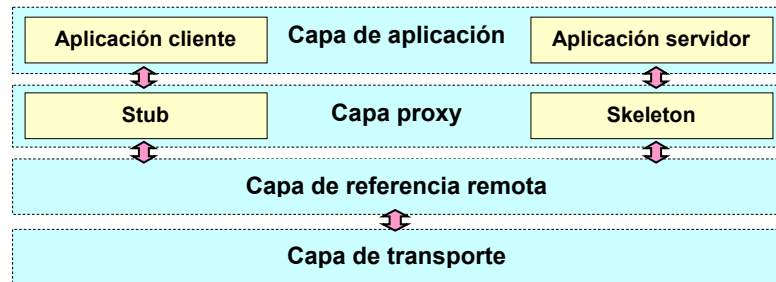


Figura 4.8. Arquitectura RMI en la que J2EE basa la comunicación de sus componentes.

La primera capa es la de aplicación y se corresponde con la implementación real de las aplicaciones cliente y servidor. Aquí tienen lugar las llamadas a alto nivel para acceder y exportar objetos remotos. Cualquier aplicación que quiera que sus métodos estén disponibles para su acceso por clientes remotos debe declarar dichos métodos en una interfaz que extienda `java.rmi.Remote`. Dicha interfaz se usa básicamente para *marcar* un objeto como remotamente accesible. Una vez que los métodos han sido implementados, el objeto debe ser exportado. Esto puede hacerse de forma implícita si el objeto extiende la clase `UnicastRemoteObject` (paquete `java.rmi.server`), o puede hacerse de forma explícita con una llamada al método `exportObject()` del mismo paquete.

La segunda capa es la capa proxy, o capa *stub-skeleton*. Esta capa es la que interactúa directamente con la capa de aplicación. Todas las llamadas a objetos remotos y acciones junto con sus parámetros y retorno de objetos tienen lugar en esta capa.

La tercera capa es la de referencia remota, y es responsable del manejo de la parte semántica de las invocaciones remotas. También es responsable de la gestión de la replicación de objetos y realización de tareas específicas de la implementación con los objetos remotos, como el establecimiento de las persistencias semánticas y estrategias adecuadas para la recuperación de conexiones perdidas. En esta capa se espera una conexión de tipo *stream* (*stream-oriented connection*) desde la capa de transporte.

La cuarta y última capa es la de transporte. Es la responsable de realizar las conexiones necesarias y manejo del transporte de los datos de una máquina a otra. El protocolo de transporte subyacente para RMI es

JRMP (*Java Remote Method Protocol*), que solamente es válido para aplicaciones Java.

Toda aplicación RMI normalmente se descompone en 2 partes:

- Un servidor, que crea algunos objetos remotos, crea referencias para hacerlos accesibles, y espera a que el cliente los invoque.
- Un cliente, que obtiene una referencia a objetos remotos en el servidor, y los invoca.

4.2.1.4. Servidores de aplicación J2EE

Sun Microsystems, con su servidor de aplicaciones *Sun Java System Application Server*, no es la única que ha desarrollado su especificación J2EE. Bajo «certificados oficiales de compatibilidad», fabricantes diversos han planteado sus propias implementaciones y pueden garantizar que se ajustan completamente al estándar J2EE. *WebSphere (IBM)*, *Oracle Application Server (Oracle Corporation)* y *WebLogic (BEA)* están entre los servidores de aplicación J2EE propietarios más conocidos. *EAServer (Sybase Inc.)* es también conocido por ofrecer soporte a otros lenguajes diferentes a Java, como *PowerBuilder*. El servidor de aplicaciones *JOnAS*, desarrollado por el consorcio *ObjectWeb*, fue el primer servidor de aplicaciones libre en lograr certificación oficial de compatibilidad con J2EE. *Tomcat (Apache Software Foundation)* y *JBoss* son otros servidores de aplicación libres muy populares en la actualidad.

La portabilidad de Java también ha permitido que los servidores de aplicación J2EE se encuentren disponibles sobre una gran variedad de plataformas, como *Microsoft Windows*, *Unix* y *GNU/Linux*.

4.2.2. Plataforma .NET

.NET es un proyecto de *Microsoft* para crear una nueva plataforma de desarrollo de software con énfasis en transparencia de redes, con independencia de plataforma y que permita un rápido desarrollo de aplicaciones.

.NET podría considerarse una respuesta de *Microsoft* al creciente mercado de los negocios en entornos Web, como competencia a la plataforma Java de *Sun Microsystems*.

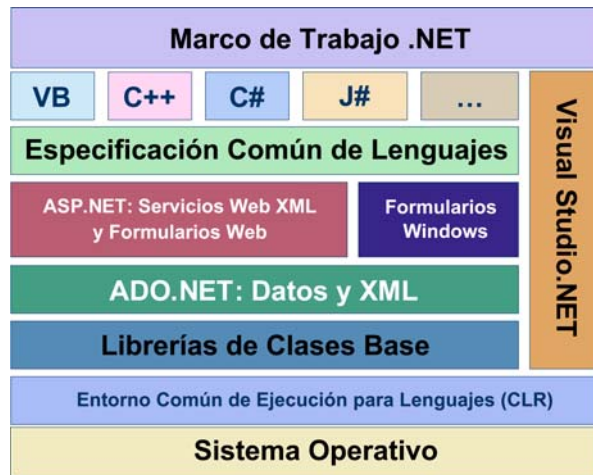


Figura 4.9. Componentes de .NET Framework.

4.2.2.1. .NET Framework

El «framework» o marco de trabajo .NET constituye la base de la plataforma .NET y denota la infraestructura sobre la cual se reúnen un conjunto de lenguajes, herramientas y servicios que simplifican el desarrollo de aplicaciones en entornos de ejecución distribuidos. Este marco también reúne una serie de normas impulsadas por varias compañías además de Microsoft (como Hewlett-Packar, Intel, IBM, Fujitsu Software, Plum Hall, la Universidad de Monash e ISE).

Los principales componentes del marco de trabajo son:

- El conjunto de lenguajes de programación.
- La Biblioteca de Clases Base o BCL.
- El Entorno Común de Ejecución para Lenguajes (CLR).

Debido a la publicación de la *norma para la infraestructura común de lenguajes (Common Language Infrastructure, CLI)*, se ha facilitado enormemente el desarrollo de lenguajes de programación compatibles, por lo que el marco de trabajo .NET soporta ya más de 20 lenguajes de programación y es posible desarrollar cualquiera de los tipos de aplicaciones soportados en la plataforma con cualquiera de ellos, lo que elimina las diferencias que existían entre lo que era posible hacer con uno u otro lenguaje.

Algunos de los lenguajes desarrollados para el marco de trabajo .NET son: *C#*, *Visual Basic*, *Turbo Delphi for .NET*, *C++*, *J#*, *Perl*, *Python*, *Fortran* y *Cobol.NET*.

4.2.2.2. Common Language Runtime

El entorno común para la ejecución (*Common Language Runtime*, CLR) es el verdadero núcleo del marco de trabajo .NET. El CLR es el entorno de ejecución en el que se cargan las aplicaciones desarrolladas en los distintos lenguajes, ampliando el conjunto de servicios del sistema operativo.

La herramienta de desarrollo compila el código fuente de cualquiera de los lenguajes soportados por .NET en un código intermedio (*Microsoft Intermediate Language*, MSIL), similar al *BYTECODE* de Java. Para generar dicho código el compilador se basa en el *Common Language Specification* (CLS) que determina las reglas necesarias para crear ese código MSIL compatible con el CLR.

Para ejecutarse se necesita un segundo paso. Un compilador JIT (*Just-In-Time*) genera el código máquina real que se ejecuta en la plataforma del cliente; de esta forma se consigue con .NET independencia de la plataforma hardware.



Figura 4.10. Diagrama de la estructura interna del CLR.

La compilación JIT la realiza el CLR a medida que el programa invoca métodos. El código ejecutable generado se almacena en la memoria caché del ordenador, siendo recompilado de nuevo sólo en el caso de producirse algún cambio en el código fuente.

4.2.2.3. Objetos COM+

Bajo las siglas COM+ (*Common Object Model Plus*) se define el modelo de componentes software basados en servicios, propuesto por Microsoft que combina, extiende y unifica los servicios propuestos por los anteriores modelos COM, DCOM y MTS:

- COM (*Component Object Model*). Modelo de objeto componente. Arquitectura software de componentes de Microsoft diseñada para facilitar la reutilización de software y la interoperatividad de los componentes.
- DCOM (*Distributed COM*). Extiende las prestaciones ofrecidas por el modelo COM a través de las redes de computadores.
- MTS (*Microsoft Transaction Server*). Servidor de transacciones de Microsoft. Permite la administración de hilos, gestión de transacciones, operación cooperativa de conexiones de bases de datos y seguridad.

Además de integrar los servicios descritos, añade nuevos modelos: de seguridad, de hilos, administración de transacciones, de administración simplificada; y nuevos servicios: cola de componentes (QC), sucesos débilmente vinculados (LCE), operación cooperativa de objetos y administrador de compensación de recursos (CRM).

El modelo COM+ identifica los siguientes elementos:

- Componente COM: módulo software o archivo binario, por ejemplo los controles ActiveX (generalmente una DLL) o los componentes que encapsulan lógica de negocio en una aplicación distribuida.
- Servidor COM: aplicación que ofrece servicios.
- Cliente COM: aplicación que demanda servicios.
- Interfaz COM: conjunto de métodos y propiedades que definen el comportamiento de un componente y que posibilita el acceso a su funcionalidad.

4.2.2.4. .NET Remoting

.NET Remoting proporciona un marco para que los objetos de distintos dominios de aplicaciones, procesos y equipos se puedan comunicar entre sí con compatibilidad en tiempo de ejecución, permitiendo que estas interacciones se realicen de forma transparente. En un sentido más amplio, también podemos entender *.NET Remoting* como una evolución más de los modelos de componentes antes analizados.

Existen tres tipos de objetos que se pueden configurar como objetos de servicios remotos .NET, y entre los que se puede seleccionar el tipo que mejor se adapte a los requisitos de la aplicación:

- **Objetos Single Call.** Los objetos *Single Call* sólo pueden cubrir una solicitud entrante. Este tipo de objetos resulta bastante útil en aquellos escenarios en los que se debe realizar una determinada cantidad de trabajo y, por lo general, no precisan, ni pueden, almacenar información de estado entre las distintas llamadas a métodos. No obstante, se pueden configurar de forma que permitan el equilibrio de la carga.
- **Objetos Singleton.** Los objetos *Singleton* sirven a varios clientes y, por lo tanto, pueden compartir información al almacenar el estado entre las llamadas de los clientes. Resultan bastante útiles cuando los datos se deben compartir obligatoriamente entre los clientes, y en aquellas situaciones en las que se produce un uso significativo de los recursos derivados de la creación y mantenimiento de los objetos.
- **Objetos activados en el cliente (CAO).** Los objetos activados en el cliente son objetos del lado del servidor que se activan cuando el cliente realiza una solicitud. Este método de activación de los objetos de servidor es muy similar al de la clásica activación de una clase asociada «*CoClass*» de COM.

Los dominios de aplicaciones y las aplicaciones .NET se comunican entre sí a través de mensajes. Los denominados *servicios del canal de .NET* facilitan el medio de transporte necesario para establecer estas comunicaciones.

El marco .NET proporciona dos tipos de canales: los *canales HTTP* y los *canales TCP*, aunque los productos de terceros pueden escribir y conectarse a sus propios canales. El canal HTTP utiliza el protocolo SOAP de forma predeterminada para establecer la comunicación, mientras que el canal TCP recurre a la carga binaria.

Los objetos de servicios remotos .NET se pueden alojar en:

- **Un ejecutable administrado.** Los objetos de servicios remotos .NET se pueden alojar en cualquier archivo .NET EXE normal o en un servicio administrado.
- **Internet Information Server (IIS).** De forma predeterminada, estos objetos reciben los mensajes a través del canal HTTP. Para alojar objetos de servicios remotos en IIS se debe crear una raíz virtual, en la que se copiará un archivo `remoting.config`. El ejecutable o la DLL que contiene el objeto remoto se debe colocar en el directorio

bin, en el directorio al que señala la raíz de IIS. Se pueden exponer los objetos de servicios remotos .NET como servicios Web.

- **Servicios de componentes .NET.** Los objetos de servicios remotos .NET se pueden alojar en la infraestructura de servicios de componentes .NET para aprovechar los diferentes servicios de COM+, tales como las transacciones, JIT, la agrupación de objetos, etc.

4.2.3. CORBA

CORBA (*Common Object Request Broker Architecture*) es un diseño de middleware que, análogamente a J2EE o a .Net framework, permite que los programas de aplicación se comuniquen unos con otros con independencia de sus lenguajes de programación, sus plataformas hardware y software, las redes sobre las que se comunican, su ubicación y sus implementadores. De hecho, CORBA representa la primera propuesta de middleware aceptado como tal.

Esta arquitectura común para la negociación de solicitudes de objetos fue propuesta por el grupo de administración de objetos OMG. Su principal cometido es trasladar una petición de un objeto cliente —componente CORBA— a una implementación de un objeto, proporcionando los mecanismos por los que los objetos hacen peticiones y reciben respuestas de forma transparente.

Para ello proporciona un modelo de programación sobre bloques básicos arquitectónicos y un nivel de abstracción que permite a las aplicaciones desarrolladas sobre él obtener independencia con respecto a las plataformas sobre las que actúan.

CORBA define los siguientes elementos:

- IDL (*Interface Definition Language*). Lenguaje de definición de interfaces.
- CDR (*CORBA Data Representation*). Representación de datos de CORBA.
- ORB (*Object Request Broker*). Intermediario de petición de objetos.
- IIOP (*Internet Inter ORB Protocol*). Protocolo Inter-ORB para Internet.
- GIOP (*General Inter ORB Protocol*). Protocolo General Inter-ORB.
- RMI (*Remote Method Invocation*). Invocación a métodos remotos.

Las aplicaciones se construyen mediante objetos CORBA, que implementan interfaces definidas en IDL. Los clientes acceden a los

métodos de las interfaces de los objetos CORBA mediante RMI. El componente middleware que da soporte a RMI es ORB.

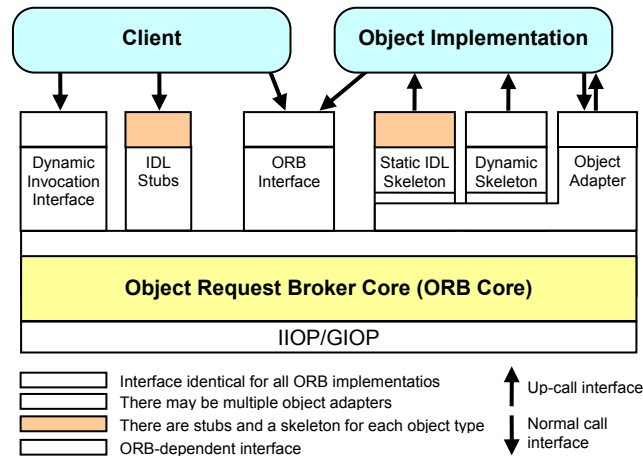


Figura 4.11. Arquitectura CORBA.

Los servicios CORBA proporcionan un equipamiento genérico que puede utilizarse en una gran variedad de aplicaciones. Estos servicios incluyen el servicio de nombres, los servicios de eventos y notificación, el servicio de seguridad, los servicios de transacción y concurrencia y el servicio de comercio.

4.3. INTEGRACIÓN DE TECNOLOGÍAS

Durante estos primeros cuatro capítulos del libro hemos ido desgranando, una tras otra, múltiples tecnologías, herramientas e implementaciones orientadas al desarrollo de grandes aplicaciones distribuidas sobre Internet. El problema es que este tipo de aplicaciones son complejas y, en muchas ocasiones, disponer de un abanico tecnológico tan amplio puede resultar un verdadero quebradero de cabeza a la hora de contestar a preguntas como: qué tecnología es la más adecuada en cada momento, cómo emplearla de la mejor forma y, sobre todo, cómo trabajar con diferentes propuestas de diferentes fabricantes al mismo tiempo.

Una de las mejores opciones para afrontar este tipo de problemas es aplicar soluciones que ya han demostrado ser exitosas en otros supuestos. Este tipo de soluciones se han ido recogiendo desde los años 60 en especificaciones que denominamos «patrones». Existen patrones en muy

diversos ámbitos, como en el diseño, en la optimización o en la organización de las aplicaciones.

En este apartado veremos cómo un patrón de diseño conocido como MVC, aplicado a las diferentes tecnologías que propone J2EE, puede resultar de gran ayuda. Por supuesto, la propuesta es directamente aplicable a otras tecnologías como .NET.

4.3.1. El paradigma MVC

Las tecnologías vistas se enmarcan dentro del desarrollo de aplicaciones Web orientado a objetos y, por lo tanto, deberían responder al *patrón de diseño Modelo-Vista-Controlador* (MVC). Este patrón propone dividir la aplicación en tres partes diferenciadas:

- **El Modelo** o la lógica de negocio de la aplicación. De entre todos los elementos tecnológicos que han ido apareciendo alrededor del modelo Web básico, y que hemos estudiado en capítulos anteriores, el papel de *modelo* queda reservado a los *servidores de aplicación*, gestionados mediante un *marco de trabajo middleware* y basando los desarrollos en el modelo de *componentes software distribuidos* al que estos marcos proporcionan servicios.
- **La Vista** es responsable de gestionar el nivel de presentación, proporcionando una interfaz de usuario altamente desacoplada, tanto de la lógica de negocio como, sobre todo, del cliente. En la práctica, resulta muy aconsejable poder aislar al máximo este apartado del resto de la aplicación, de forma que se pueda dedicar personal especializado en aspectos como el diseño gráfico y de interfaces de usuario más que en el de aplicaciones.
- **El Controlador**. Es el componente que se encarga de manejar la interacción entre la *vista* y el *modelo*, y tiene definido el flujo de la aplicación. Actúa como catalizador necesario para desarrollar las dos partes anteriores. De su elección depende principalmente que se puedan diferenciar nítidamente los distintos roles que deben establecerse para el desarrollo de aplicaciones Web.

Como se ha comentado ya, la lógica de negocio o *Modelo* deberá estar codificada en forma de componentes software, que la encapsulen y la hagan portable y reutilizable entre diferentes sistemas. Sin embargo, la manera en la que repartimos los otros dos papeles es algo abierto a discusión.

Veamos a continuación las distintas posibilidades utilizando como ejemplo de tecnología: servlets por parte de la ejecución de programas, JSP por parte de las páginas activas y Enterprise JavaBean por parte del middleware y de los componentes. En general, la mayor parte de las reflexiones que aquí se realizan son válidas para otros tandems —por ejemplo, el basado en ASP y COM+ o el basado en Liveware y CORBA, por no mencionar todas las posibilidades de interacción entre todos ellos empleando lenguajes de definición normalizados como XML.

4.3.1.1. Aplicación Web basada en JSP

Utilizando únicamente esta tecnología, la lógica de programación se implementa directamente en las páginas JSP. Las páginas JSP harán el papel de controlador y vista a la vez. Si esto es así, los navegadores deberán hacer peticiones directamente a estas páginas, las cuales interactuarán con los componentes (Beans) necesarios e insertarán los resultados obtenidos en formato HTML, el cual volverá al cliente.

Las páginas JSP se encuentran, además, perfectamente integradas con el entorno general del servidor de aplicaciones, pudiendo relacionarse con las otras *piezas* que integran el mismo.

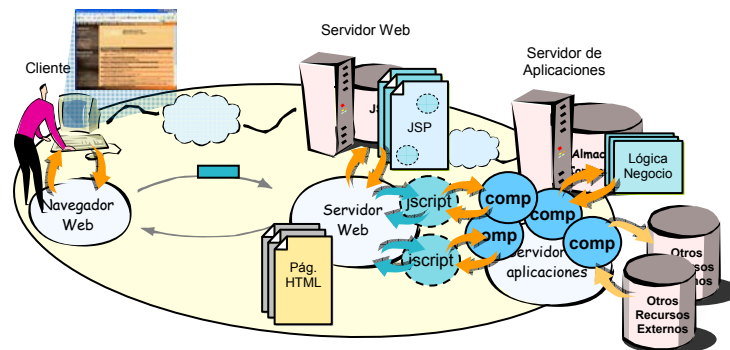


Figura 4.12. Escenario en el que se despliega una aplicación distribuida organizada según un patrón MVC implementado con tecnología JSP.

Es importante recordar que el código fuente (los *scripts*) incrustado en las páginas JSP se extrae y, a partir del mismo, se generarán sus respectivos objetos *servlets*, que es lo que realmente ejecutará el contenedor JSP.

4.3.1.2. Aplicación Web basada en Servlets

En este modelo, los servlets hacen el papel de controlador y de vista a la vez, es decir, contienen la lógica de programación y el resultado o presentación final codificados en Java. Esto implica que cada vez que haya una modificación de formato, aunque no la haya de lógica, debemos modificar y recompilar el servlet correspondiente.

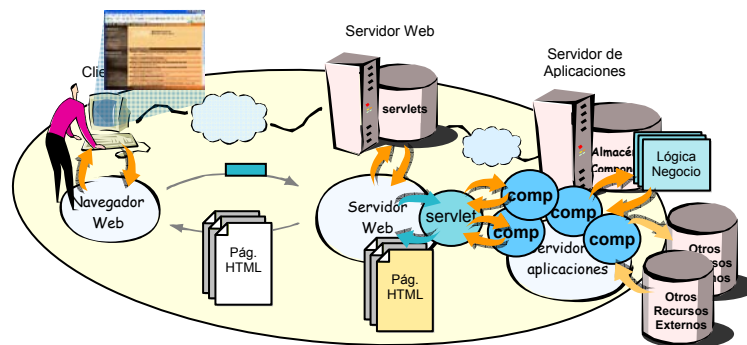


Figura 4.13. Escenario en el que se despliega una aplicación distribuida organizada según un patrón MVC implementado con tecnología *Servlet*.

Aunque aporta un mecanismo muy potente, en la práctica, no facilita la independencia necesaria entre los diferentes papeles de equipo de trabajo que deben confluir en el desarrollo de este tipo de sistemas.

4.3.2. MVC aplicado a J2EE

En este caso, la especificación J2EE fue realizada por SUN para, precisamente, abordar este tipo de problemas inherentes al desarrollo de aplicaciones distribuidas. Por lo tanto, cuenta con todos los ingredientes para desplegar totalmente el patrón MVC. De hecho, la propia SUN, consciente de la complejidad que entraña la utilización de su plataforma, fue la primera en facilitar a sus usuarios su propia versión sobre cómo aplicar el patrón MVC con J2EE. Esta especificación se conoce como «el modelo 2».

Según el *modelo 2 de SUN*, la lógica de programación irá ubicada en servlets, que posteriormente invocan a páginas JSP. Podemos colocar la lógica de programación en los servlets, de forma que los clientes les

invoquen y éstos, a su vez, a páginas JSP, las cuales accederán a la información que necesiten sólo para presentar resultados.

Serán los servlets los que interactuarán con los JavaBeans con el fin de realizar cualquier cálculo o acceder a la lógica de negocio. Los servlets podrán además crear Beans donde almacenar los resultados obtenidos.

Posteriormente, las páginas JSP extraerán la información requerida y almacenada en estos Beans, con el fin de insertarla en su HTML. El resultado final, tras mezclar el contenido dinámico con el estático, se envía al cliente o navegador.

Las páginas JSP constituyen una tecnología mediante la cual podemos presentar páginas con contenido dinámico en base a etiquetas (*tags*) especiales embebidas en el código HTML de la página. Mediante estas etiquetas podremos incrustar código Java para ejecutar lógica de aplicación directamente o bien acceder a un componente externo que realice esta tarea por él, devolviéndole posteriormente un resultado.

Al actuar los servlets como controladores, las páginas JSP, al margen del HTML normal de la página, deben llevar sólo *tags* especiales para su relación con los JavaBeans.

Usando los llamados «*scriptlets*» podremos introducir código Java en nuestras páginas JSP. Pero no debemos abusar de esta funcionalidad, ya que cuanto más lo hagamos, más dificultoso será separar los diferentes roles o perfiles en un equipo de desarrollo. En la medida de lo posible, la lógica de programación deberá ir en los servlets, la lógica de negocio en los JavaBeans y la presentación o interfaz gráfica en páginas JSP.

Este último enfoque es el método que juzgamos más idóneo, ya que separa la vista o presentación (en páginas JSP) de la lógica de programación (en servlets) y aporta la indudable ventaja de que podemos separar el contenido de presentación o estático del contenido dinámico, estando la lógica de negocio en componentes o Beans externos a la página.

Según este esquema, en la figura 4.14 podemos apreciar una representación gráfica de la política de trabajo propuesta en este apartado. El cliente interactúa con los servlets, en los que va a estar albergada la lógica y el flujo de programación. Desde ellos se accede a las clases que encapsulan toda la lógica de negocio (*Business Objects*), que son independientes de la filosofía Web.

Los resultados que se obtengan se almacenan en Beans (*View Objects*) y se presentan a través de páginas JSP, las cuales tienen acceso a los mismos para obtener la parte dinámica. Si la presentación es estática, se presentan simples páginas HTML.

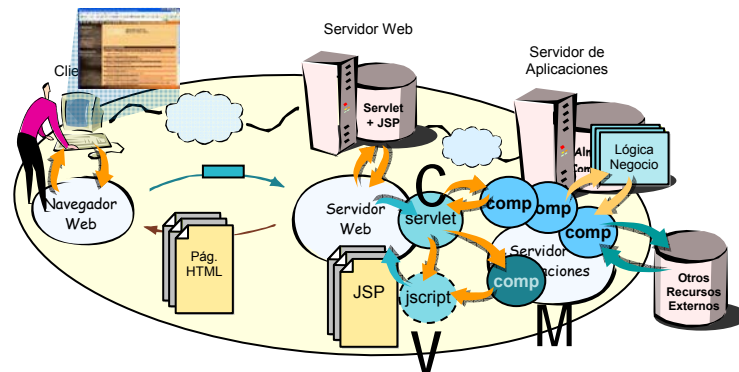


Figura 4.14. Escenario en el que se despliega una aplicación distribuida organizada según un modelo 2 propuesto por SUN en el que se identifica el (M)odelo, la (V)ista y el (C)ontrolador.

Si somos capaces de separar en la medida de lo posible estos dos mundos —presentación y lógica—, estaremos diferenciando también entre el papel del personal dedicado al diseño y a la usabilidad, de aquél dedicado a plasmar el modelo de negocio, lo que redundará en su especialización, pudiendo cada uno de ellos emplear para su cometido las mejores herramientas en cada uno de los campos.

4.3.3. Escenario de desarrollo

En la figura 4.15 se presenta un posible escenario de desarrollo, más o menos complejo, en el que confluyen la mayor parte de los elementos abordados en esta lección.

Por parte de los clientes, podemos observar diferentes tipos de dispositivo y mecanismos de interconexión a la Red desde PCs convencionales, hasta PDAs o teléfonos móviles. Por parte del servidor Web, se plantea un sencillo cluster (posiblemente soportando balanceo de carga) que representará el punto de entrada a la aplicación; por supuesto, apoyándose desde el punto de vista operativo en los dispositivos típicos de red y seguridad como *routers*, *proxys* o *firewalls*. En lo que respecta a los servidores de aplicaciones, se han separado los nodos de ejecución de la aplicación o de los componentes software de aquellos nodos que pueden servir como almacén de componentes. Finalmente, se proporciona un entorno de almacenamiento tipo NAS (*Network Attached Storage*) o SAN (*Storage Area Networks*), que puede corresponder tanto a aplicaciones y

sistemas heredados como a almacenes de persistencia de los propios componentes.

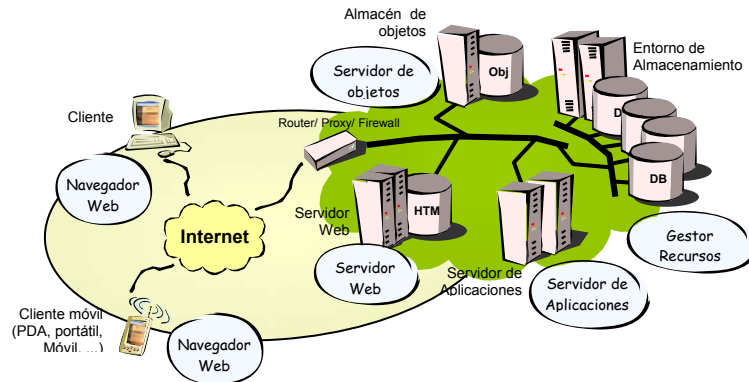


Figura 4.15. Escenario general en el que se recogen los principales componentes físicos y lógicos que proporcionan soporte a una aplicación distribuida sobre Internet.

4.4. CASO DE USO: JBOSS

JBoss es un servidor de aplicaciones J2EE de código abierto implementado en Java puro. Al estar basado en Java, JBoss puede ser utilizado en cualquier sistema operativo que soporte la máquina virtual de java. Los principales desarrolladores trabajan para una empresa de servicios, JBoss Inc., adquirida por Red Hat en abril del 2006, fundada por *Marc Fleury*, el creador de la primera versión de JBoss. El proyecto está apoyado por una red mundial de colaboradores. Los ingresos de la empresa están basados en un modelo de negocio de servicios.

JBoss AS fue el primer servidor de aplicaciones de código abierto, preparado para la producción y certificado J2EE 1.4, disponible en el mercado, ofreciendo una plataforma de alto rendimiento para aplicaciones de negocio electrónico (actualmente se está desarrollando la versión para J2EE 5). Combinando una arquitectura orientada a servicios revolucionaria con una licencia de código abierto, JBoss AS puede ser descargado, utilizado, incrustado, y distribuido sin restricciones por la licencia. Por este motivo es la plataforma más popular de middleware para desarrolladores, vendedores independientes de software y, también, para grandes empresas.

Las características destacadas de JBoss incluyen:

- Producto de licencia de código abierto sin coste adicional.