

Arquitectura de aplicaciones web

Leandro Navarro Moldes

P07/M2106/02842



Universitat Oberta
de Catalunya

www.uoc.edu

Índice

Introducción	5
Objetivos	6
1. Características de la demanda de páginas web	7
2. Organización de las aplicaciones en servidores web	15
2.1. Organización del servidor web	15
2.2. Organización de las aplicaciones web	17
2.3. Interfaz común de pasarela (<i>common gateway interface</i> , CGI)	17
2.3.1. FastCGI	18
2.4. Servlet Java	19
2.4.1. La API de <i>servlets</i>	20
2.5. Resumen y comparación	22
3. Servidores <i>proxy-cache</i> web	23
4. Contenidos distribuidos	28
4.1. Redes de distribución de contenidos	29
5. Computación orientada a servicios	33
5.1. Computación bajo demanda	34
Resumen	35
Actividades	37
Ejercicios de autoevaluación	37
Solucionario	38
Glosario	38
Bibliografía	39

Introducción

En este módulo didáctico se van a tratar las formas de organizar aplicaciones web y de cómo hacer que puedan funcionar pese a estar sujetas al comportamiento caótico e imprevisible de Internet.

Primero se caracteriza la demanda de estos servicios y cómo medirla en la práctica. Después, se describen las formas de construir y la evolución de los servicios web (cgi, *servlets*, servidores de aplicaciones y servidores web), y se analizan los casos de distintos servidores web; para acabar hablando de formas distribuidas de servicio: servidores intermediarios *proxy-cache*, redes de distribución de contenidos, aplicaciones orientadas a servicios y computación bajo demanda.

La forma de adquirir los conocimientos pasa por realizar los pequeños experimentos que se ofrecen en el apartado de actividades y en la web de la asignatura, y que ayudan tanto a concretar las ideas centrales como a tener experiencias propias y personales de los fenómenos, técnicas y herramientas que se describen.

Objetivos

Los objetivos de este módulo didáctico son los siguientes:

- 1.** Conocer las características de la demanda que debe satisfacer un servidor web.
- 2.** Conocer las distintas maneras de organizar una aplicación web y los modelos que existen, según los distintos criterios.
- 3.** Conocer las características y el funcionamiento de cada modelo.
- 4.** Poder elegir la mejor opción en cada situación y valorar las implicaciones del montaje que hay que realizar.

1. Características de la demanda de páginas web

El tráfico de web es el responsable de un buen porcentaje del tráfico de Internet. Esta tendencia ha ido creciendo gradualmente desde que apareció la web (protocolo HTTP), y hoy día el tráfico HTTP predomina respecto del resto de los protocolos, y hay una gran población de usuarios “navegantes” que pueden generar una cantidad inmensa de peticiones si el contenido es interesante. La organización de un servicio web conectado a Internet requiere tener en cuenta las características de la demanda que pueda tener que atender.

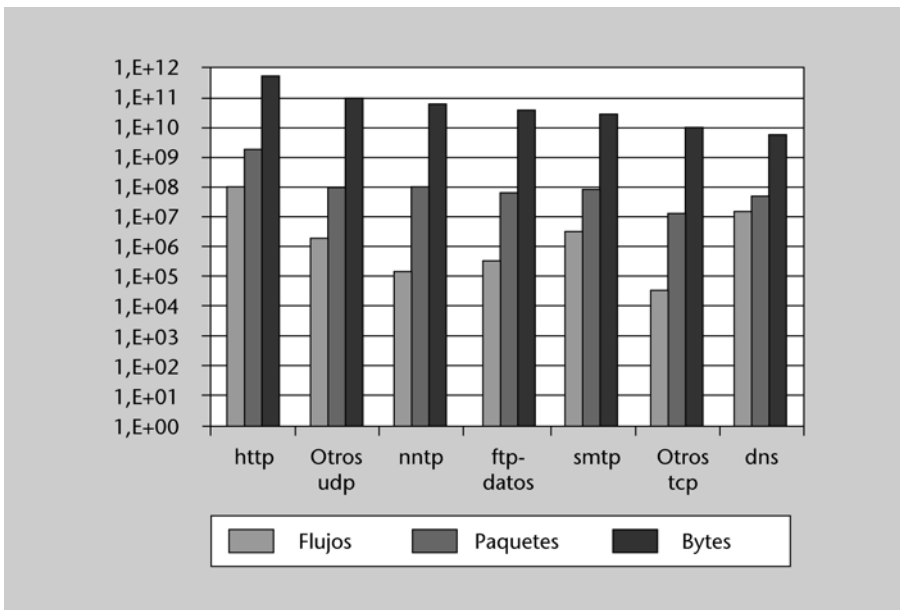
Arrecifes de coral y tráfico en Internet

La organización CAIDA (www.caida.org) se dedica al análisis del tráfico en Internet y ha desarrollado una herramienta denominada *Coral Reef* que toma trazas del tráfico de un enlace. Con ésta, en 1998 hicieron un estudio de tráfico por protocolos en el núcleo de la red del proveedor MCI.

El artículo que lo describe se presentó en la conferencia Inet 98, y se titulaba “The nature of the beast: recent traffic measurements from an Internet backbone”.

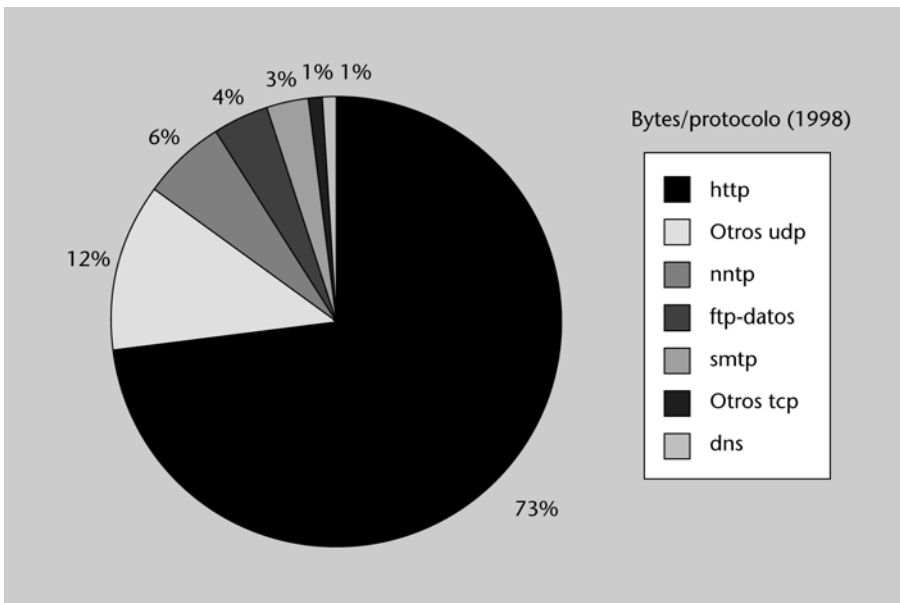
Las gráficas adjuntas se han obtenido de estas medidas.

Figura 1



Volumen de tráfico en escala logarítmica de flujos, paquetes y bytes intercambiados durante veinticuatro horas en un enlace del núcleo de la red de MCI/Worlcom (1998), organizado por el protocolo.

Figura 2



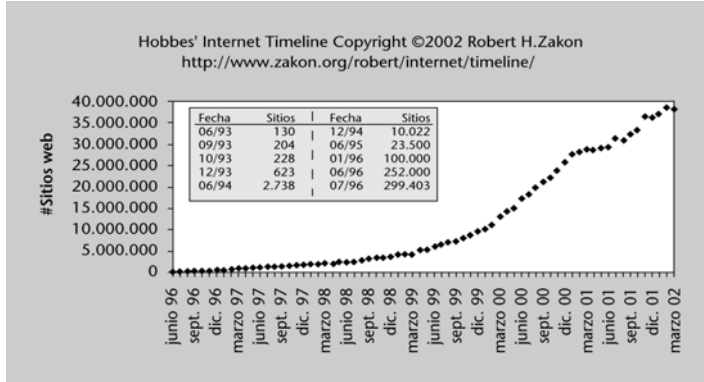
Porcentaje de tráfico en bytes de cada protocolo respecto al total medido. Puede apreciarse mejor que en la figura 1, en escala logarítmica, que el porcentaje de tráfico web domina el resto (73% del total).

Por otro lado, la web (HTTP) es un servicio muy reclamado por todo tipo de organizaciones para publicar información, como puede verse en la tendencia de crecimiento del número de servidores web en Internet, que ha sido exponencial, tal y como muestra la figura 3.

La cronología de Internet

Un calendario de los eventos relacionados con Internet desde 1957 hasta hoy lo podéis ver en la dirección siguiente: <http://www.zakon.org/robert/internet/timeline/>

Figura 3



Crecimiento del número de sitios web durante los últimos seis años.

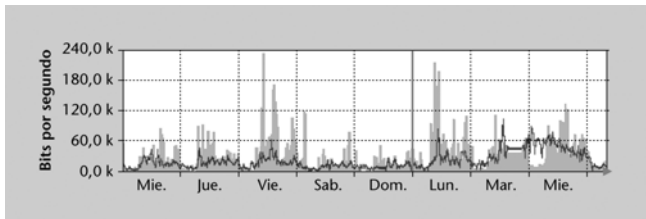
La popularidad de los servidores web también es muy variable. Un mismo sitio web puede recibir muy pocas visitas durante mucho tiempo y, de repente, recibir veces más peticiones de las que puede servir: es un tráfico a ráfagas.

Flash crowd

Un cuento de ciencia ficción de varias Larry Niven (1973) predijo que una consecuencia de un mecanismo de teletransporte barato sería que grandes multitudes se materializarían instantáneamente en los lugares con noticias interesantes. Treinta años después, el término se usa en Internet para describir los picos de tráfico web cuando un determinado sitio se hace popular de repente y se visita de manera masiva.

También se conoce como efecto "slashdot" o efecto "/.", que se da cuando un sitio web resulta inaccesible a causa de las numerosas visitas que recibe cuando aparece en un artículo del sitio web de noticias slashdot.org (en castellano, barrapunto.com).

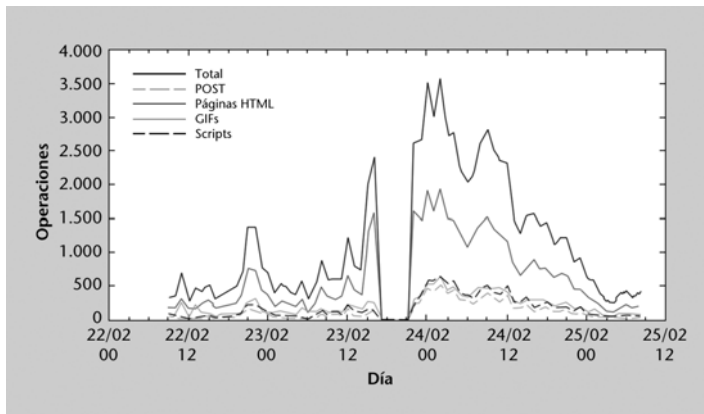
Figura 4



Evolución del tráfico entrante y saliente de un sitio web típico durante una semana. Podéis observar la gran variación horaria y la reducción de tráfico durante el fin de semana.

Un servidor puede recibir avalanchas repentinas de tráfico. Por ejemplo, por las estadísticas del siguiente servidor web sabemos que, después de ser anunciado en la páginas de noticias slashdot.org, sufrió un exceso de visitas tan alto que el servidor se bloqueó:

Figura 5



Peticiones web por hora, servidas por http://counter.li.org durante tres días. Puede verse que mientras que el número habitual de operaciones (peticiones web) estaba por debajo de 500, subió rápidamente a unas 2.500, lo cual provocó el fallo del sistema. Después de reconfigurarlo, estuvo soportando durante unas doce horas en torno a 3.000 peticiones/hora para bajar posteriormente a valores normales. La historia completa está en la dirección: <http://counter.li.org/slashdot/>

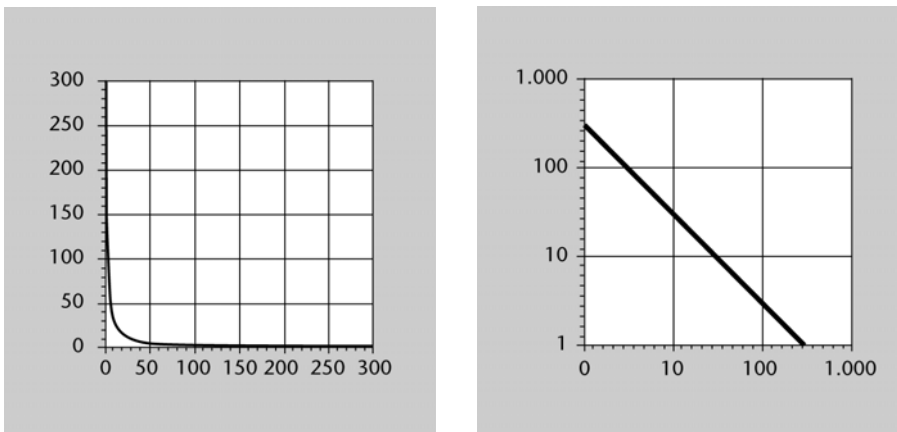
Un servidor web puede tener miles de documentos y, sin embargo, recibir la mayoría de las peticiones por un único documento. En muchos casos, la popularidad relativa entre distintos sitios web o entre diferentes páginas de un cierto sitio se rige por la ley de Zipf (George Kingsley Zipf, 1902-1950) que dice:

La frecuencia de suceso de un evento concreto (P) como función del rango (i) cuando el rango es determinado por la frecuencia de suceso es una función potencial $P_i \sim 1/i^a$, con el exponente a cercano a la unidad.

El ejemplo más famoso es la frecuencia de palabras en inglés. En 423 artículos de la revista *Time* (245.412 palabras), *the* es la que más aparece (15.861), *of* está en segundo lugar (7.239 veces), *to* en tercer lugar (6.331 veces), y con el resto forman una ley potencial con un exponente cercano a 1.

Una distribución de popularidad Zipf forma una línea recta cuando se dibuja en una gráfica con dos ejes en escala logarítmica, que resulta más fácil de ver y comparar que la gráfica en escala lineal, tal y como puede verse en la figura siguiente:

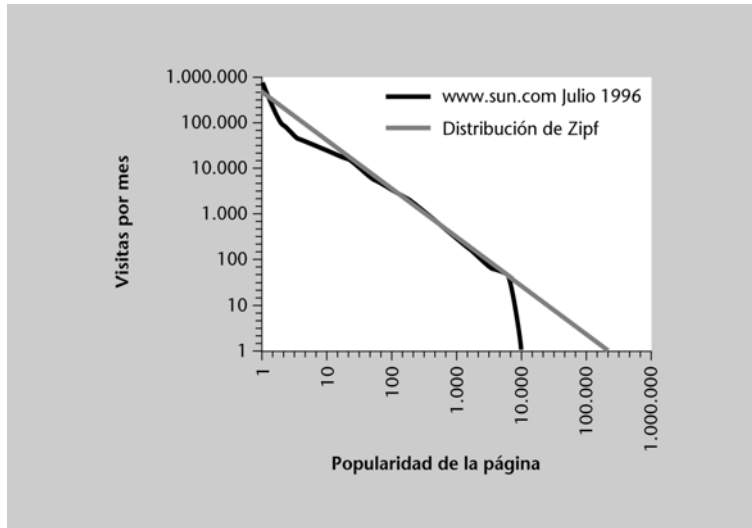
Figura 5



Una distribución de popularidad (casos ordenados por popularidad en el eje x, valor de popularidad en el eje y) que sigue la ley de Zipf a escala lineal queda "enganchada a los ejes": muy pocos casos tienen mucha popularidad y muchos casos tienen muy poca. Por este motivo, se suele representar en escalas logarítmicas (gráfica doble logarítmica: los dos ejes en escala logarítmica).

Muchos estudios muestran que las visitas de páginas web siguen una distribución de Zipf. La figura siguiente muestra las visitas en www.sun.com durante un mes de 1996. La página principal recibió prácticamente 1 millón de visitas, mientras que la página de la posición 10.000 de popularidad sólo recibió una visita aquel mes. La gráfica de visitas sigue la curva de Zipf excepto para los valores menos populares, lo cual seguramente se debe al hecho de que el periodo de observación no fue lo bastante largo.

Figura 6



Número de visitas de las páginas de www.sun.com ordenadas por popularidad. Puede verse cómo se ajusta a una distribución de Zipf.

Como resumen, diferentes estudios del tráfico web contribuyen a definir un perfil típico o reglas a ojo de la web –según M. Rabinovich y O. Spatscheck (2002). *Web Caching and Replication*. Addison Wesley. ISBN: 0201615703:

- El tamaño medio de un objeto es de 10-15 kbytes, y la media de 2-4 kbytes. La distribución se decanta claramente hacia objetos pequeños, aunque se encuentra una cantidad nada despreciable de objetos grandes (del orden de Mbytes).
- La mayoría de los accesos a la web son por objetos gráficos, seguidos de los documentos HTML. El 1-10% son por objetos dinámicos.
- Una página HTML incluye de media diez imágenes y múltiples enlaces a otras páginas.
- Un 40% de todos los accesos son para objetos que se considera que no se pueden inspeccionar.
- La popularidad de objetos web es muy diferente: una pequeña fracción de objetos es la responsable de la mayoría de los accesos, siguiendo la ley de Zipf.
- El ritmo de acceso para objetos estáticos es muy superior al ritmo de modificación.
- En una escala de tiempo inferior al minuto el tráfico web es a ráfagas, por lo cual valores medidos con medias durante algunas decenas de segundo son muy poco fiables.
- Un 5-10% de accesos a la web se cancelan antes de finalizar.
- Prácticamente todos los servidores utilizan el puerto 80.

Cada sitio web es un poco distinto, y puesto que un servidor web es un sistema complejo y, por lo tanto, difícil de modelar, resulta conveniente hacer experimen-

tos tanto para ver cómo los niveles de carga (peticiones de páginas web) crecientes afectan a nuestro servidor, como para observar periódicamente el comportamiento de un servidor web analizando los diarios (*logs*) que puede generar.

Para probar el rendimiento de un servidor web, normalmente se utiliza algún programa que, instalado en otro computador, genere un ritmo de peticiones equivalentes para medir el efecto de las visitas simultáneas desde distintos clientes. El ritmo de peticiones puede configurarse de una manera ligeramente distinta para cada herramienta, pero el objetivo es ser estadísticamente equivalente a la demanda real que el servidor pueda experimentar durante su funcionamiento normal. Esto recibe el nombre de *carga sintética*.

Hay muchas herramientas. A continuación se describen brevemente tres herramientas populares y gratuitas.

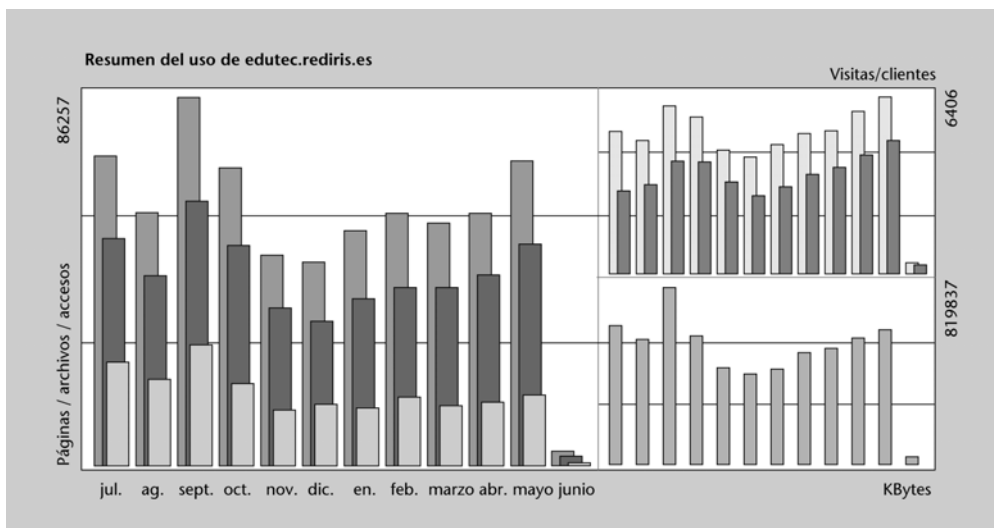
- *Microsoft Web Application Stress (WAS)* es una herramienta de simulación para Windows diseñada para medir el rendimiento de un sitio web. Tiene en cuenta las páginas generadas dinámicamente (ASP) en un servidor web de Microsoft.
- *Apache JMeter*, una aplicación Java para medir el rendimiento de documentos y recursos estáticos y dinámicos (archivos, *servlets*, *scripts* Perl, objetos Java, consultas de bases de datos, servidores FTP, etc.), que simula distintos tipos de carga extrema de la red, del servidor o de un objeto concreto.
- *Surge* de la Universidad de Boston: una aplicación Java que genera peticiones web con características estadísticas que simulan con mucha precisión la demanda típica de un servidor web.

Visitas web sobre generadores de carga

Apache JMeter:
jakarta.apache.org/jmeter/
 Surge:
www.cs.bu.edu/faculty/crovella/links.html

Durante el funcionamiento normal del servidor, es conveniente supervisar la demanda y el rendimiento del servicio para detectar la degradación del servicio (responde muy lentamente por exceso de peticiones o tráfico) o situaciones críticas (sobrecarga: no responde).

Figura 7



Una de las muchas gráficas que genera una herramienta de estadísticas web popular y gratuita denominada *webalizer*, que puede encontrarse en la dirección <http://www.mrunix.net/webalizer/> y que facilita mucho el análisis de la actividad de un servidor web.

Las herramientas de visualización y análisis de actividad del servidor se basan en el hecho de que prácticamente todos los servidores son capaces de generar archivos históricos de la actividad del servidor (conocidos como *diarios*, en inglés *logs*) en un formato conocido como *common log format* o CLF.

En CLF cada línea (a veces denominada *entrada*) registra una petición recibida por el servidor. La línea está formada por distintos elementos separados por espacio:

máquina ident usuarioautorizado fecha petición estado bytes

Si un dato no tiene valor, se representa por un guión (-). El significado de cada elemento es el siguiente.

- Máquina: el nombre DNS completo o su dirección IP si el nombre no está disponible.
- Ident: si está activado, la identidad del cliente tal y como lo indica el protocolo identd. Puede no ser fiable.
- UsuarioAutorizado: si se pidió un documento protegido por contraseña, corresponde al nombre del usuario utilizado en la petición.
- Fecha: la fecha y hora de la petición en el formato siguiente:

```
date = [día/mes/año:hora:minuto:segundo zona]
día = 2*digit
mes = 3*letter
año = 4*digit
hora = 2*digit
minuto = 2*digit
segundo = 2*digit
zona = ('+' | '-') 4*digit
```

- Petición: el URL solicitado por el cliente, delimitado por comillas ("").
- Estado: el código de resultado de tres dígitos devuelto al cliente.
- Bytes: el número de bytes del objeto servido, sin incluir cabeceras.

Este formato es adecuado para registrar la historia de las peticiones, pero no contiene información útil para medir el rendimiento del servidor. Por ejemplo, no indica el tiempo transcurrido en servir cada URL.

Para permitir construir un formato de entrada de diario (*log*) que contenga la información necesaria, puede definirse un formato particular que puede contener otra información. A continuación se muestra una lista de las variables que el servidor web Apache puede guardar (*mod_log*).

Nombre	Descripción de la variable
%a	Dirección IP remota.
%A	Dirección IP local.
%B	Bytes enviados, excluyendo las cabeceras HTTP.
%b	Bytes enviados, excluyendo las cabeceras HTTP. En formato CLF: un '-' en lugar de un 0 cuando no se ha enviado ningún byte.
%c	Estado de la conexión cuando la respuesta se acaba: 'X' = conexión abortada antes de acabar la respuesta. '+' = conexión que puede quedar activa después de haber enviado la respuesta. '-' = conexión que se cerrará después de haber enviado la respuesta.
%{NOMBRE}e	El contenido de la variable de entorno NOMBRE.
%f	Nombre del fichero.
%h	Nombre de la máquina remota.
%H	El protocolo de la petición.
%{Nombre}i	El contenido del encabezado o encabezados "Nombre:" de la petición enviada al servidor.
%l	Usuario remoto (de <i>identd</i> , si lo proporciona).
%m	El método de la petición.
%{Nombre}n	El contenido de la "nota" "Nombre" desde otro módulo.
%{Nombre}o	El contenido de la cabecera o cabeceras "Nombre:" de la respuesta.
%p	El puerto del servidor sirviendo la petición.
%P	El identificador del proceso hijo que sirvió la petición.
%q	El texto de una consulta o <i>query string</i> (precedido de ? si la consulta existe, si no un texto vacío).
%r	Primera línea de la petición.
%s	Estado de peticiones que fueron redireccionadas internamente, el estado de la petición original - %>s para el de la última.
%t	Tiempo (fecha) en formato de LOG (formato estándar inglés).
%{format}t	El tiempo (hora), en el formato especificado, que debe expresarse en formato strftime (posiblemente localizado).
%T	El tiempo de servicio de la petición, en segundos.
%u	Usuario remoto (de autenticación; puede ser incorrecto si el estado de la respuesta %s es 401).
%U	La parte de camino (<i>path</i>) del URL, sin incluir el texto de la consulta (<i>query string</i>).
%v	El nombre original o <i>canónico</i> del servidor dependiente de la petición.
%V	El nombre del servidor según el valor del orden <i>UseCanonicalName</i> .

Según estas variables, el formato CLF sería:

```
"%h %l %u %t \"%r\" %>s %b"
```

El formato CLF incluyendo el servidor web virtual solicitado (un servidor web puede servir a diferentes dominios DNS o servidores virtuales):

```
"%v %h %l %u %t \"%r\" %>s %b"
```

- El formato NCSA extendido/combinado sería:

```
"%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-agent}i\""
```

Analizar la demanda y rendimiento de un servidor es una tarea necesaria, ya que los servidores web están sujetos a variaciones de demanda muy extrema. Después del análisis de los ficheros de diario del servidor puede ser necesario limitar, resituar y ampliar los recursos del servidor y la red de acceso a Internet para poder atender aceptablemente el “extraño y voluble” tráfico de peticiones que visita los servidores web.

2. Organización de las aplicaciones en servidores web

Las aplicaciones web –aplicaciones que van asociadas o son extensiones de un servidor web– pueden necesitar un diseño y ajuste muy cuidadosos para ofrecer un rendimiento adecuado en situaciones de alta demanda, o simplemente para responder rápidamente o aprovechar de manera adecuada los recursos de la máquina en la que están instalados.

En primer lugar, hay que saber cómo está organizado un servidor web para atender peticiones HTTP de la manera más eficiente.

En segundo lugar, se debe conocer cómo puede extenderse el servidor web para ofrecer otros servicios gestionados por un código adicional.

2.1. Organización del servidor web

Para caracterizar cómo se organiza un servidor web para atender peticiones de una manera eficiente y económica, es necesario definir algunos términos.

- Proceso: la unidad más “pesada” de la planificación de tareas que ofrece el sistema operativo. No comparte espacios de direcciones ni recursos relacionados con ficheros, excepto de manera explícita (heredando referencias a ficheros o segmentos de memoria compartida), y el cambio de tarea lo fuerza el núcleo del sistema operativo (*preemptivo*).
- Flujo o *thread*: la unidad más “ligera” de planificación de tareas que ofrece el sistema operativo. Como mínimo, hay un flujo por proceso. Si distintos flujos coexisten en un proceso, todos comparten la misma memoria y recursos de archivos. El cambio de tarea en los flujos lo fuerza el núcleo del sistema operativo (*preemptivo*).
- Fibra: flujos gestionados por el usuario de manera cooperativa (*no preemptivo*), con cambios de contexto en operaciones de entrada/salida u otros puntos explícitos: al llamar a ciertas funciones. La acostumbran a implementar librerías fuera del núcleo, y la ofrecen distintos sistemas operativos.

Para ver qué modelos de proceso interesan en cada situación, hay que considerar las combinaciones del número de procesos, flujo por proceso y fibras por flujo. En todo caso, cada petición la sirve un flujo que resulta la unidad de ejecución en el servidor.

Arquitectura del servidor web

El apartado “4.4 Server Architecture” del libro *Krishnamurthy, Web Protocols and Practice* (págs. 99-116), describe un estudio sobre el funcionamiento de Apache 1.3.

Los modelos que se pueden construir son (U: único, M: múltiple):

Proceso	Flujo	Fibra	Descripción
U	U	U	Es el caso de los procesos gestionados por <i>inetd</i> . Cada petición genera un proceso hijo que sirve la petición.
M	U	U	El modelo del servidor Apache 1.3 para Unix: distintos procesos preparados que se van encargando de las peticiones que llegan. Lo implementa el módulo Apache: <i>mpm_prefork</i> .
M	U	M	En cada proceso, una librería de fibras cambia de contexto teniendo en cuenta la finalización de operaciones de entrada/salida. En Unix se denomina <i>select-event threading</i> y lo usan los servidores Zeus y Squid. Ofrece un rendimiento mejor en Unix que en MUU. Lo implementa el módulo Apache <i>state-threaded multi processing</i> .
M	M	U	El modelo MMU cambia fibras por flujos. Lo implementan los módulos Apache: <i>perchild</i> y <i>mpm_worker_module</i> . El número de peticiones simultáneas que puede atender es $ThreadsPerChild \times MaxClients$.
U	M	U	El modelo más sencillo con diferentes flujos. Se puede montar en Win32, OS2, y con flujos POSIX. Lo implementa el módulo Apache: <i>mpm_netware</i> .
U	M	M	Probablemente el que proporciona un rendimiento más alto. En Win32 se puede conseguir con los denominados <i>completion ports</i> . Se usan los flujos necesarios para aprovechar el paralelismo del <i>hardware</i> (número de procesadores, tarjetas de red), y cada flujo ejecuta las fibras que han completado su operación de entrada/salida. Es el modelo con un rendimiento más alto en Windows NT. Lo implementa el módulo Apache: <i>mpm_winnt_module</i> . Muchos servidores como Internet Information Server o IIS 5.0 utilizan este modelo con Windows NT. El servidor web interno al núcleo de Linux <i>Tux</i> también utiliza este modelo.
M	M	M	Puede ser una generalización de UMM o MUM, y en general la presencia de distintos procesos hace que el servidor se pueda proteger de fallos como consecuencia de que un proceso tenga que morir por fallos internos, como por ejemplo el acceso a memoria fuera del espacio del proceso.

En general, los modelos con muchos procesos son costosos de memoria (cada proceso ocupa su parte) y de carga (creación de procesos). En servidores de alto rendimiento, los modelos con flujos parecen mejores, aunque tienen el problema de la portabilidad difícil y la posible necesidad de mecanismos que anticipan el coste de creación de procesos o flujos y, por lo tanto, son muy rápidos atendiendo peticiones.

Cuando la red limita el servicio web, lanzar procesos para atender peticiones puede funcionar razonablemente bien, pero en redes de alta velocidad, como ATM o Gigabit Ethernet, la latencia en iniciar un proceso al recibir una petición es excesivo.

En máquinas uniprocador, los modelos con un solo flujo funcionan bien. En máquinas multiprocador, es necesario usar múltiples flujos o procesos para aprovechar el paralelismo del *hardware*.

El mayor obstáculo para el rendimiento es el sistema de ficheros del servidor, ya que la función principal del servidor web es “trasladar” ficheros del disco a la red. Si éste se ajusta, el siguiente obstáculo es la gestión de la concurrencia.

Además, los estudios de tráfico web indican que la mayoría de las peticiones son de ficheros pequeños, por lo cual sería conveniente optimizar el servidor para poder priorizar estas peticiones que son más frecuentes y mejoran el

TUX: servidor web en el núcleo de Linux

Tux es un servidor web incorporado al núcleo de Linux que usa un *pool* con muy pocos flujos del núcleo (un flujo por procesador), lee directamente de la memoria del sistema de ficheros de dentro del núcleo, usa su propio algoritmo de planificación de fibras y usa el TCP/IP “zero-copy” que minimiza las veces que los datos que vienen de la red se copian en la memoria (en redes de alta velocidad como Gigabit Ethernet, las copias de bloques de memoria son el factor limitante). Puede servir entre dos y cuatro veces más peticiones/segundo que Apache o IIS.

Para más información:
www.redhat.com/docs/manuals/tux/

tiempo de respuesta percibido por los usuarios, por ejemplo al cargar páginas web con muchos gráficos pequeños insertados.

Por esta razón, Apache 2.0 es un servidor web modular en el cual la gestión del proceso está concentrada en un módulo que puede seleccionarse en la instalación, según las características del sistema operativo, la máquina, los recursos que tendrá que utilizar el servidor, etc.

2.2. Organización de las aplicaciones web

Los servidores web se encargan de atender y servir peticiones HTTP de recursos, que en su forma más simple acostumbran a ser documentos guardados en el sistema de ficheros. Sin embargo, la otra función importante de un servidor web es la de actuar de mediador entre un cliente y un programa que procesa datos. Recibe una petición con algún argumento, la procesa y devuelve un resultado que el servidor web entrega al cliente. La interacción entre el servidor web y los procesos que tiene asociados es otro aspecto que hay que considerar.

Existen distintas maneras de organizar una aplicación web. A continuación, por orden cronológico de complejidad y rendimiento creciente, se presentan distintos modelos de organización.

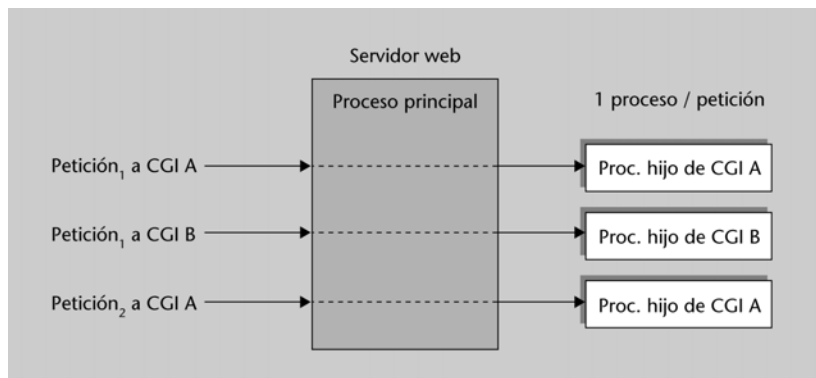
- CGI: es el modelo más antiguo y simple. Para cada petición HTTP se invoca un programa que recibe datos por las variables del entorno y/o de entrada estándar, y devuelve un resultado por la salida estándar. Consumir un proceso por cada petición genera problemas importantes de rendimiento, que el modelo FastCGI intenta mejorar.
- *Servlets*: es un modelo diseñado para Java, más eficiente y estructurado, que permite elegir distintos modelos de gestión de flujos o *threads*, duración de procesos del servidor, etc. Partiendo de este modelo, se han construido servidores de aplicaciones con múltiples funciones adicionales que facilitan el desarrollo de aplicaciones web complejas.

2.3. Interfaz común de pasarela (*common gateway interface*, CGI)

La interfaz común de pasarela es un estándar para proporcionar una interfaz web a programas que se ejecutan en cada petición y que, por lo tanto, pueden devolver información *dinámica*. Puesto que un programa CGI es ejecutable –es como dejar que todo el mundo ejecute un programa en el servidor–, hay que tomar muchas precauciones al hacer accesibles programas CGI por vía del servidor web.

Un CGI es un programa externo que se ejecuta y responde a cada petición del servidor web dirigida al CGI. El modo de funcionamiento es el siguiente:

Figura 8



Un servidor web que utiliza procesos (comandos) CGI.

Cada petición crea un proceso que recibe los datos de entrada por medio de la entrada estándar y del entorno, y genera una respuesta por la salida estándar.

Por ejemplo, si se quiere “conectar” una base de datos a la web, para que todo el mundo la pueda consultar, se debería escribir un CGI. Al pedir al servidor web un URL dirigido al CGI, el programa consultará la base de datos y devolverá un texto, posiblemente HTML, que presente el resultado de la pregunta.

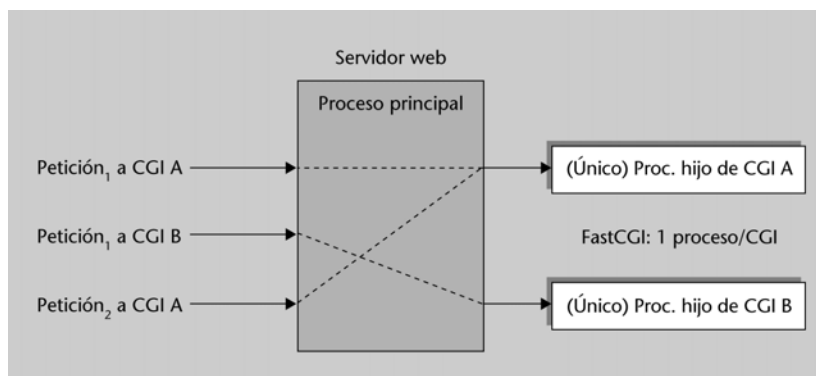
Se puede conectar cualquier programa que siga las reglas sencillas de los CGI para ser invocado por el servidor web, siempre que el programa no tarde mucho en responder, ya que el usuario o el navegador se pueden cansar de esperar.

Este procedimiento consume muchos recursos del servidor y es lento. Esta información se puede ampliar en: <http://www.w3.org/CGI/>.

2.3.1. FastCGI

Para solucionar el problema anterior, se creó una mejora de los CGI que hace que un solo proceso cargado vaya sirviendo peticiones sin descargarse (un proceso persistente o *daemon*), pero a veces no basta con un proceso y es necesario tener distintos procesos al mismo tiempo atendiendo diferentes peticiones que se dan de manera simultánea.

Figura 9

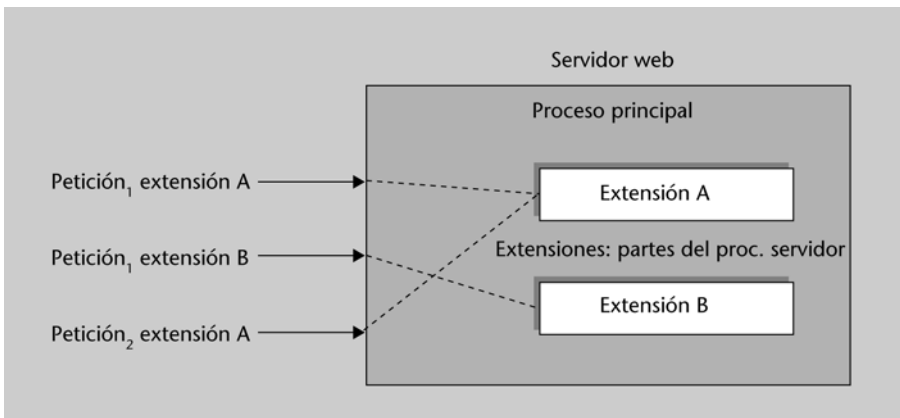


Un servidor que utiliza procesos persistentes FastCGI.

Tanto los CGI como los FastCGI no pueden interactuar con el interior del servidor (por ejemplo, generar una línea en los ficheros de diario del servidor). Más detalles en <http://www.fastcgi.com>.

Otra alternativa para extender el servidor de una manera más eficiente consiste en utilizar las API de extensión de cada servidor (NSAPI en el servidor de Netscape/Sun, ISAPI en el servidor de Netscape, o un módulo en Apache). Las extensiones forman parte del proceso servidor y estas API ofrecen mucha más funcionalidad y control sobre el servidor web, además de velocidad, al estar compiladas y formar parte del mismo ejecutable.

Figura 10



Un servidor que incorpora código externo usando la API de extensión.

Sin embargo, tienen tres problemas importantes: son extensiones no portables, específicas para un único servidor; son más complejas de desarrollar y mantener; e introducen riesgo en el proceso servidor –peligro en lo que respecta a la seguridad y fiabilidad (un error de la extensión puede detener el proceso servidor de web).

2.4. Servlet Java

Un *servlet* es una extensión genérica del servidor: una clase Java que se puede cargar dinámicamente en el servidor para “extender” la funcionalidad del servidor web. En muchos casos, sustituye con ventajas los CGI.

Es una extensión que se ejecuta en una máquina virtual Java (JVM) dentro del servidor: es seguro y transportable. Puesto que se ejecuta desde el servidor, el cliente web lo invocará como si fuese un CGI, y en la respuesta sólo verá el texto HTML, sin que el cliente deba tratar con Java (como sí hace en los *applets*, que es código Java que se ejecuta en una máquina virtual Java del cliente web).

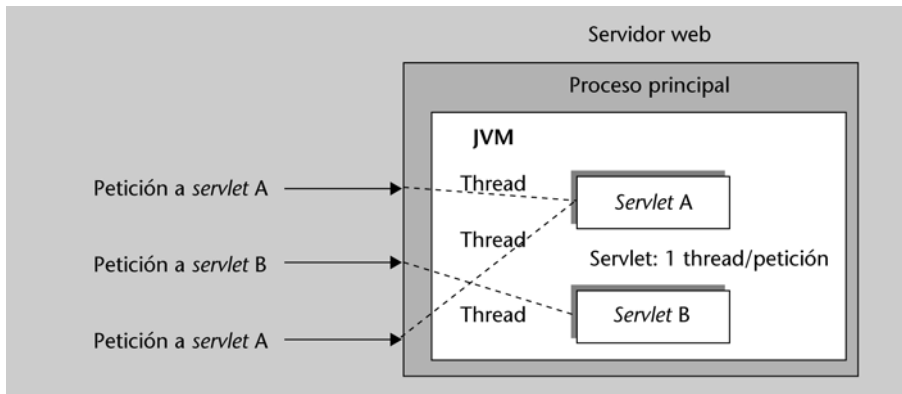
En general, un *servlet* es un trozo de código que se ejecuta en un servidor: se puede usar para “extender” servidores de web, pero también sirve para otros servidores (p. ej., FTP para añadir comandos nuevos, correo para filtrar o detectar virus, etc.).

Servlets: una aportación de Sun

La especificación de Servlets continúa evolucionando.

Los *servlets* son una extensión estándar de Java, y las clases acostumbran a venir con las distribuciones de Java SDK (Equipo de Desarrollo Java) o en la dirección siguiente: <http://java.sun.com/products/servlet>

Figura 11

Un servidor con *servlets* en su JVM.

Para usar los *servlets*, es necesario un “motor” en el que probarlos y ponerlos en servicio. Pueden ser servidores que ya soporten *servlets* (por ejemplo, Tomcat de Apache, Domino Go de Lotus, Website de O’Reilly, Jigsaw del Consorcio Web), o módulos que se pueden añadir a servidores que inicialmente no los soportaban (por ejemplo, Jserv para Apache: <http://tomcat.apache.org>).

Las ventajas principales de los *servlets* son las siguientes:

- **Portabilidad.** Siempre usan las mismas llamadas (API) y circulan sobre Java. Esto hace que sean verdaderamente portátiles entre entornos (que soporten *servlets*).
- **Versatilidad.** Pueden usar toda la API de Java (excepto AWT), además de comunicarse con otros componentes con RMI, CORBA, usar Java Beans, conectar con bases de datos, abrir otros URL, etc.
- **Eficiencia.** Una vez cargado, permanece en la memoria del servidor como una única instancia. Distintas peticiones simultáneas generan diferentes flujos sobre el *servlet*, que es muy eficiente. Además, al estar cargado en la memoria puede mantener su estado y mantener conexiones con recursos externos, como bases de datos que puedan reclamar un cierto tiempo para conectar.
- **Seguridad.** Además de la seguridad que introduce el lenguaje Java (gestión de memoria automática, ausencia de punteros, tratamiento de expresiones), el gestor de seguridad o *security manager* puede evitar que *servlets* malintencionados o mal escritos puedan dañar el servidor web.
- **Integración con el servidor.** Pueden cooperar con el servidor de unas maneras en las que los CGI no pueden, como cambiar el camino del URL, poner líneas de diario en el servidor, comprobar la autorización, asociar tipos MIME a los objetos o incluso añadir usuarios y permisos al servidor.

Tomcat

El servidor tomcat es un servidor web escrito con Java que soporta directamente *servlets* y es de uso libre: <http://jakarta.apache.org/tomcat>.

2.4.1. La API de *servlets*

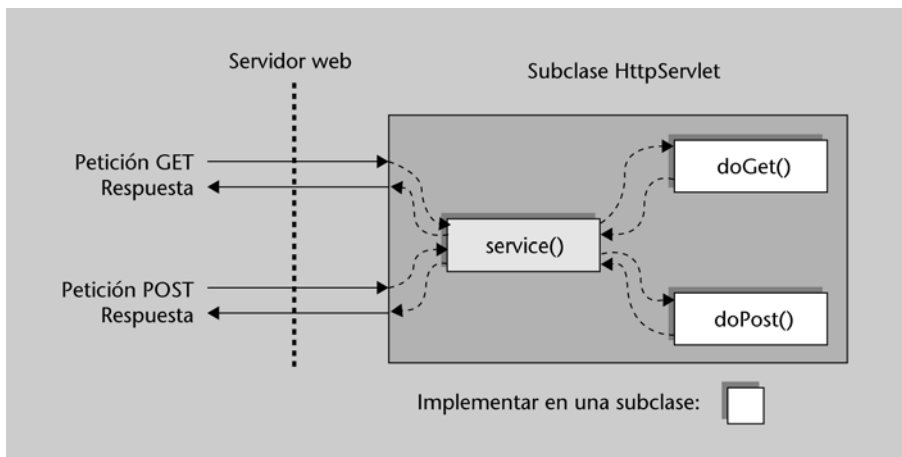
Los *servlets* usan clases e interfaces de dos paquetes: `javax.servlet` que contiene clases para *servlets* genéricos (independientes del protocolo que usen) y

`javax.servlet.http` (que añade funcionalidad particular de HTTP). El nombre `javax` indica que los *servlets* son una extensión.

Los *servlets* no tienen el método `main()` como los programas Java, sino que se invocan unos métodos cuando se reciben peticiones. Cada vez que el servidor envía una petición a un *servlet*, se invoca el método `service()` que se tendrá que reescribir (*override*). Este método acepta dos parámetros: un objeto petición (*request*) y un objeto respuesta.

Los *servlets* HTTP, que son los que usaremos, ya tienen definido un método `service()` que no es necesario redefinir y que llama el `doXXX()` con `XXX`, el nombre del orden que viene con la petición del servidor web: `doGet()`, `doPost()`, `doHead()`, etc.

Figura 12



Un *servlet* HTTP que trata peticiones GET y POST.

A continuación se puede observar el código de un *servlet* HTTP mínimo que genera una página HTML que dice “¡Hola amigos!” cada vez que se invoca en el cliente web.

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

public class hola extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.println("<html><big>¡Hola amigos!</big></html>");
    }
}
```

El servidor extiende la clase `HttpServlet` e implementa el método `doGet()`. Cada vez que el servidor web recibe una petición GET para este *servlet*, el servi-

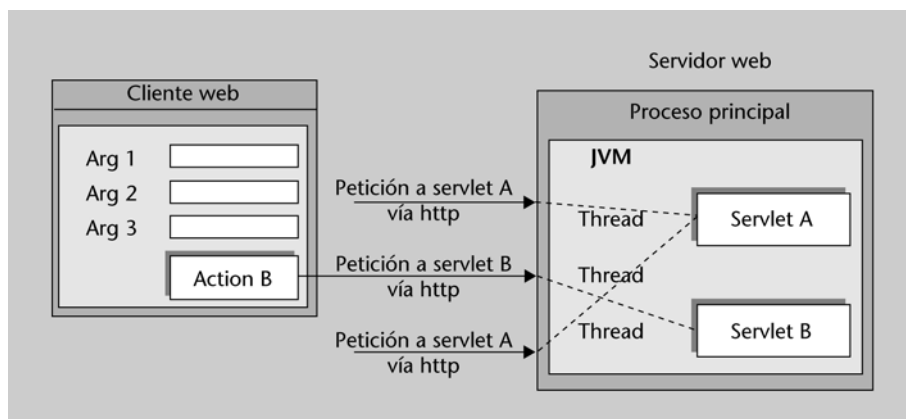
El cliente web invoca su método `doGet()` y le hace llegar un objeto con datos de la petición `HttpServletRequest` y con otro objeto `HttpServletResponse` para devolver datos en la respuesta.

El método `setContentType()` del objeto respuesta (`res`) establece como texto/HTML el contenido MIME de la respuesta. El método `getWriter()` obtiene un canal de escritura que convierte los caracteres Unicode que usa Java en el juego de caracteres de la operación HTTP (normalmente ISO-8859-1). Aquel canal se usa para escribir el texto HTML que verá el cliente web.

2.5. Resumen y comparación

Por lo tanto, es posible extender la funcionalidad de un servidor web incorporando *servlets* que atienden peticiones web. Usan los métodos de HTTP GET, con los argumentos codificados en el URL (*URL encoded*), y POST, donde los argumentos viajan al cuerpo de la petición. En general, los *servlets* sirven para extender la funcionalidad de cualquier servidor añadiendo nuevos servicios o “comandos”.

Figura 13



Interacción del navegador con el formulario y del servidor web con los *servlets*.

Una ventaja importante de los *servlets* respecto a los CGI es que se puede seleccionar la política de servicio (flujos e instancias): una vez instanciado un *servlet* en la máquina virtual Java (JVM), puede servir diferentes peticiones usando un flujo para cada una o, al contrario, un objeto (con un solo flujo) para cada petición. Un *servlet* también puede guardar información que persiste durante la vida del objeto o conexiones en bases de datos. Además, la librería de *servlets* facilita y abstrae el paso de parámetros y su codificación, el seguimiento de sucesivas visitas de un mismo cliente (sesiones), la transformación de juegos de caracteres entre cliente y servidor, etc.

El modelo cliente web + formulario HTML / servidor web + extensiones (CGI o *servlet*) es adecuado para aplicaciones en las cuales el usuario utiliza un cliente web que invoca una única operación al servidor con un conjunto de parámetros textuales y sin estructura que recoge el cliente web mediante un formulario HTML.

Servidores de aplicaciones

Son servidores que incorporan muchos servicios y componentes reutilizables que permiten desarrollar aplicaciones complejas accesibles por HTTP.

Algunos ejemplos de servidores de aplicaciones muy populares (los dos implementan el modelo *Java 2 Enterprise Edition* o J2EE):

www.jboss.org de código libre, www.bea.com: WebLogic Server, comercial.

3. Servidores *proxy-cache web*

Para la web, se diseñó un protocolo simple (HTTP) de acceso a documentos sobre un transporte fiable como TCP. Un objetivo de diseño era la interactividad: el cliente se conecta con el servidor web, solicita el documento (petición) e inmediatamente lo recibe del servidor. Este esquema es rápido en situaciones de tráfico en la red y carga de los servidores reducida, pero no es eficiente en situaciones de congestión.

Para todas aquellas situaciones en las cuales la comunicación directa cliente-servidor no es conveniente, se ha introducido un tipo de servidores que hacen de intermediarios entre los extremos.

Un servidor intermediario (*proxy*) es un servidor que acepta peticiones (HTTP) de clientes u otros intermediarios y genera a su vez peticiones hacia otros intermediarios o hacia el servidor web destino. Actúa como servidor del cliente y como cliente del servidor.

La idea del *proxy* se usa en muchos protocolos además del HTTP. En general, se sitúan en una discontinuidad para realizar en la misma una función. Por ejemplo:

Cambio de red. Una máquina conectada a la red interna de una organización que usa direcciones IPv4 privadas (por ejemplo, 10.*.*), y también en Internet, puede hacer de intermediario para traducción de direcciones IP entre las dos redes (NAT).

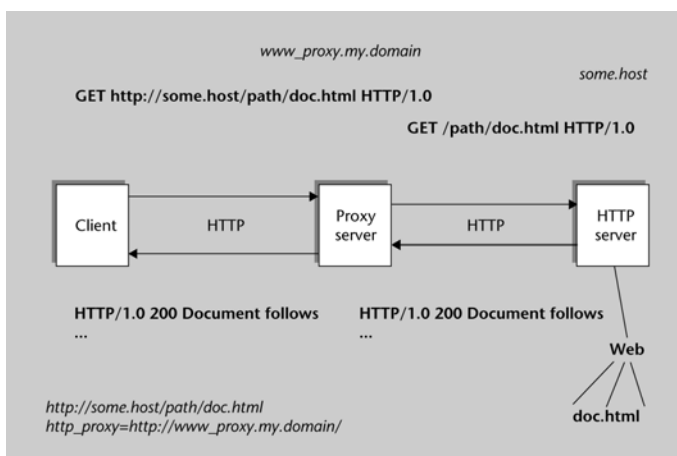
Si no se desea usar este mecanismo, que tiene ciertas limitaciones, se puede salvar la discontinuidad al nivel del HTTP poniendo un intermediario HTTP: una máquina conectada a las dos redes que recibiría peticiones de páginas web desde cualquier cliente interno por una dirección interna y volvería a generar la misma petición desde el intermediario, pero hacia Internet, con la dirección externa.

NAT (*network address translation*)

En los paquetes IP salientes: sustituir la dirección IP de máquinas internas (no válidas en Internet) por la suya propia, en los paquetes IP entrantes: sustituir su propia dirección IP por la de una máquina interna y reenviar el paquete hacia la red interna.

Para poder saber a quién entregarlo, el intermediario debe asociar cada uno de sus puertos a máquinas internas, pues una dirección de transporte es una pareja (dirección IP, puerto).

Figura 14




Interacción entre cliente-servidor intermediario-servidor tal y como aparece en la publicación original (Luotonen, 94) describiendo el servidor intermediario HTTP del CERN (Centro Europeo de Investigación en Física), en el que fue inventada la web.

Como podéis suponer, aprovechando que tanto la petición como el resultado (la página web) pasarán por el intermediario, se puede aprovechar para ofrecer algunas funciones:

Control de acceso a contenidos. El intermediario consulta si la página solicitada está o no permitida por la política de la organización. Puede hacerse consultando una tabla de permisos o usando el mecanismo denominado *PICS*.

Control de seguridad. El intermediario genera todas las peticiones que salen a Internet, lo que oculta información y evita ataques directos sobre las máquinas internas de la organización. Además, puede existir un cortafuego que no permita acceder directamente hacia el exterior, con lo que el uso del intermediario es imprescindible.

Aprovechar peticiones reiteradas (función caché). El intermediario puede almacenar (en memoria o disco) una copia de los objetos que llegan como resultado de peticiones que han pasado por el intermediario. Estos objetos se pueden usar para ahorrar peticiones hacia Internet y servir peticiones sucesivas de un mismo objeto con una copia almacenada en el intermediario, sin tener que salir a buscarlo a Internet. Los *proxy-cache* son el caso más frecuente de servidor intermediario, y son los que detallaremos aquí. 

Adaptar el contenido. Un intermediario podría también adaptar los objetos que vienen del servidor a las características de su cliente. Por ejemplo, convertir los gráficos al formato que el cliente entiende, o reducir su tamaño para clientes como teléfonos móviles con reducida capacidad de proceso, comunicación o presentación.

El intermediario puede ser transparente, invisible para cliente y servidor: se obtiene el mismo resultado con el mismo o sin el mismo, aunque en los navegadores web habitualmente el usuario debe configurar expresamente su navegador para usarlo, pues el navegador no tiene un mecanismo para detectarlo y usarlo automáticamente.

En algunas instalaciones, se puede instalar un *proxy* transparente que son *routers* extensiones del *software* del que hacen que éste intercepte las conexiones TCP salientes hacia el puerto 80 (HTTP) y las redirija a un *proxy-cache*. Tiene el peligro de que el usuario no es consciente de esto, y puede llevar a situaciones equívocas si el *proxy-cache* falla o trata mal la petición.

En la figura siguiente podemos ver cómo hay servidores intermediarios para distintos protocolos además de HTTP, y que un *proxy* puede comunicarse con el servidor origen (el que tiene el documento original que el usuario ha solicitado) o pedirlo a otro *proxy*, formando una jerarquía de *proxies*.

PICS: Platform for Internet Content Selection



El PICS define mecanismos para que se puedan catalogar sitios y páginas web según ciertos criterios y, de esta manera, controlar el acceso a contenidos no deseados. Es útil en escuelas y hogares para controlar el acceso a Internet de los menores.

Control de contenidos: ciertos contenidos que se consideren no apropiados por la organización pueden ser bloqueados o redirigidos.

Podéis encontrar más información en la dirección siguiente:
<http://www.w3.org/PICS/>.

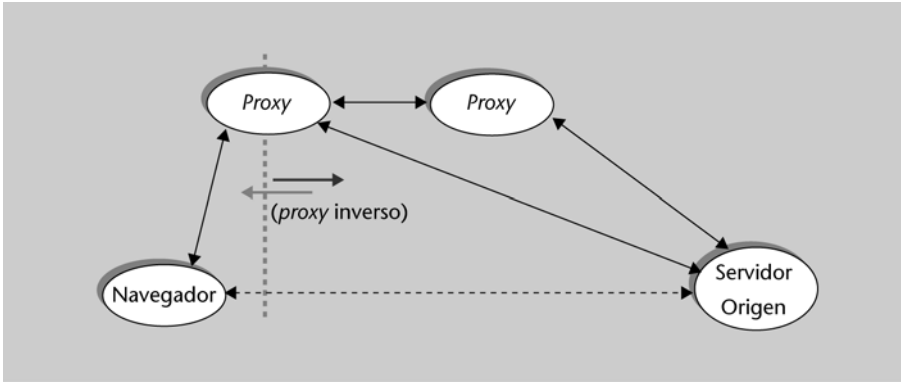
El protocolo HTTP 1/1...

... tiene definido un código de respuesta a una petición:

305 Use proxy

El problema es que no se llegó a un acuerdo al escribir la especificación, y la respuesta hace que simplemente en el navegador aparezca el mismo mensaje de error, en lugar de tratar de buscar un *proxy* y reintentar la conexión mediante el mismo. ¿Cómo arreglarlo?

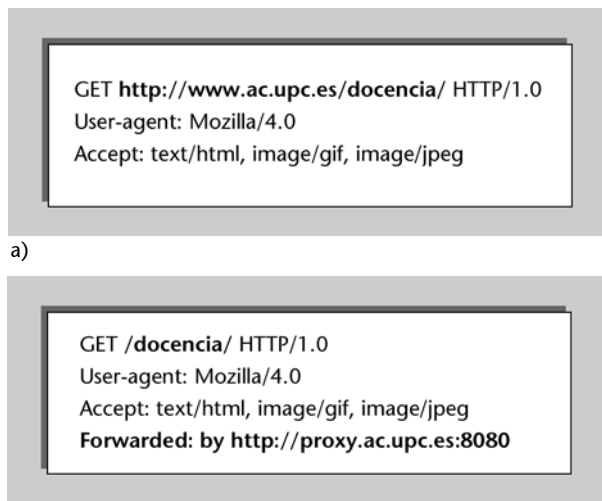
Figura 15



Un servidor intermediario se suele situar con proximidad a un cliente y puede colaborar con otros servidores intermediarios, pero también puede estar cerca de un servidor (servidor intermediario inverso).

Se usan también *proxies* en la proximidad de un servidor para reducir la carga de peticiones sobre el mismo. Son los *proxies* inversos: puede ser más sencillo y barato colocar uno o varios *proxy-cache* que reciban todas las peticiones: responderán directamente a las peticiones ya cacheadas y al servidor sólo le llegarán unas pocas que no están aún en el *proxy-cache*, o han expirado o son contenidos dinámicos.

Figura 16



a) Petición HTTP 1.0 cliente → servidor intermediario, b) petición HTTP 1.0 servidor intermediario → servidor. Podemos observar cómo el servidor intermediario introduce un campo "reenviado" (*forwarded*) para notificar su presencia al servidor.

Debido a la gran difusión del uso de servidores *proxy-cache*, sobre todo en entornos académicos, la especificación HTTP/1.1 define una nueva cabecera que permite controlar el comportamiento de un servidor *proxy-cache* tanto desde el cliente en la petición como desde el servidor en una respuesta.

Cache-control (petición):

No-cache	Cliente↔origen (las memorias caché se inhiben).
No-store	El <i>proxy</i> no debe almacenar permanentemente petición/respuesta.
Max-age = sgs	La máxima "edad" aceptable de los objetos en la <i>caché</i> .


Max-stale	Se aceptan objetos viejos.
Max-stale = sgs	Se aceptan objetos sgs segundos viejos.
Min-fresh = sgs	Al objeto deben quedarle sgs de vida.
Only-If-Cached	Petición si sólo está en el servidor <i>proxy-cache</i> .

Cache-control (respuesta):

Public	Se puede cachear por <i>proxies</i> y cliente.
Private	Sólo se puede guardar en la memoria caché del cliente.
Private="cab"	Todos pueden mediar el objeto, excepto la cabecera cab : sólo en la memoria caché del cliente.
No-cache	No se puede mediar ni en servidores intermediarios ni en el cliente.
No-cache="cab"	Combinación de los dos anteriores.
No-store	Nadie puede almacenar permanentemente (sólo en la memoria del navegador).
No-transform	Los servidores intermediarios no pueden transformar el contenido.
Must-revalidate	Revalidar (con origen) si es necesario.
Max-age	Margen de edad en segundos.

Los servidores *proxy-cache* tienen algoritmos para decidir si, cuando llega una petición de un contenido que ya tienen, es o no necesario confirmar que no haya cambiado en el servidor origen, y el campo caché-control permite influir en la decisión. Otro problema grave es la pérdida de privacidad potencial si se guardan contenidos en el *proxy-cache*, más cuando se almacenan objetos en disco. Para esto también sirve el campo cache-control.

Un *proxy-cache* está formado por un almacén de mensajes de respuesta (objetos), un algoritmo de control del almacén (entrar, salir, borrar o reemplazar) y un algoritmo rápido de consulta (*lookup*) de un mapa del contenido; toma la decisión de servirlo del almacén o pedirlo al servidor origen o a otro *proxy-cache*.

Su uso puede producir una reducción de tráfico en la red y en el tiempo de espera si los objetos que se piden están en la memoria y el contenido se puede mediar. Estudios en distintas organizaciones muestran con frecuencia tasas de acierto en la memoria del 15-60% de objetos, aunque esto pueda variar mucho con los usuarios y el contenido. 

Los servidores *proxy-cache* son sistemas pasivos que aprovechan para guardar las respuestas a peticiones de usuarios. Automáticamente no intentan guardar contenidos que parecen privados, dinámicos o de pago: los detectan por la presencia de campos en la cabecera como por ejemplo: *WWW-Authenticate*, *Cache-Control:private*, *Pragma:no-cache*, *Cache-control:no-cache*, *Set-Cookie*.

Cookies (galletas): estado y privacidad

El protocolo HTTP no tiene estado: cada interacción (petición + respuesta) no tiene relación con las demás y cada una puede usar una conexión TCP distinta.

Para poder relacionar varias interacciones HTTP y pasar información de estado entre las mismas, Netscape inventó las *cookies*: un servidor web puede enviar al cliente un objeto de hasta 4096 bytes, que contiene datos que el navegador devolverá a ese mismo servidor en el futuro (o a otros servidores que indique la *cookie*).

Las *cookies* han sido muy usadas para cuestiones publicitarias (qué sitios web visita la gente y en qué orden), para guardar contraseñas, identificar usuarios, etc. sin que el usuario sea consciente ni de que está recibiendo estas "galletas", ni de que las está enviando cuando visita la web.

Hay quien dice que son un error llevado a la perfección. ¿Qué opináis? ¿Habéis mirado alguna vez qué *cookies* tenéis en vuestro navegador?

Se han propuesto mejoras para hacer de los servidores *proxy-cache* intermedarios activos: acumular documentos de interés en horas de bajo tráfico para tener el contenido preparado y de esta manera ayudar a reducir el tráfico en horas de alta demanda, o mecanismos de *pre-fetch* en los que la memoria caché se adelanta a traer, antes de que se lo pidan, páginas web que con gran probabilidad el usuario va a solicitar a continuación. Sin embargo, no son de uso común pues no siempre suponen un ahorro o mejora del tiempo de respuesta, sino que pueden llevar a malgastar recursos de comunicación.

Los mecanismos de *proxy-cache* son útiles para que una comunidad de usuarios que comparten una conexión a Internet pueda ahorrar ancho de banda, y reducir transferencias repetitivas de objetos. Sin embargo, ésta es sólo una parte del problema.

El problema en conjunto se conoce humorísticamente como el **World-Wide Wait**. Varios agentes participan en el rendimiento de una transferencia web.

- Proveedor de contenidos (servidor web): debe planificar su capacidad para poder dar servicio en horas punta y aguantar posibles avalanchas de peticiones y, de esta manera, tener cierta garantía de que sus clientes dispondrán de buen servicio con cierta independencia de la demanda.
- Cliente: podría usar un servidor *proxy-cache* para “economizar” recursos de la red. Cuando un contenido está en más de un lugar, debería elegir el mejor servidor en cada momento: el original o una réplica “rápida” (o traer por trozos el objeto desde varios a la vez).
- Réplica: si un servidor guarda una réplica de algún contenido probablemente es porque desea ofrecerlo y reducir la carga del servidor original. Por tanto, debe ser “conocido” por sus clientes, pero además el contenido tiene que ser consistente con el contenido original.
- Proveedor de red: debería elegir el mejor camino para una petición, vía direccionamiento IP, vía resolución DNS (resolver nombres en la dirección IP más próxima al cliente que consulte, aunque no es lo más habitual), vía servidores *proxy-cache* HTTP y evitar zonas congestionadas (*hot spots*) o *flash crowd* (congestión en el servidor y en la red próxima debida a una avalancha de demandas).

Por tanto, los *proxy-caché* sólo son parte de la solución. Las redes de distribución de contenidos son la solución de otra parte del “W-W-Wait”.

¿Una réplica (*mirror*)?

En enero de 2007, el programa HTTPD de Apache, el servidor web más utilizado en Internet, podía bajarse unas **300 réplicas** del sitio web original. La lista de réplicas está en la dirección siguiente:
<http://www.apache.org/mirrors/>
y la información del servidor en la dirección:
<http://httpd.apache.org>.

4. Contenidos distribuidos

Otra mejora que ayuda a combatir el mal del “W-W-Wait” son los sistemas de distribución de documentos: ¿basta un único servidor para cualquier “audiencia”? La respuesta es que no, si se desea ofrecer una calidad adecuada para demandas tanto muy pequeñas como bastante grandes, para contenidos que pueden llegar a ser muy populares en ciertos momentos, que pueden visitarse desde cualquier lugar del mundo o que pueden tener una audiencia potencial enorme.

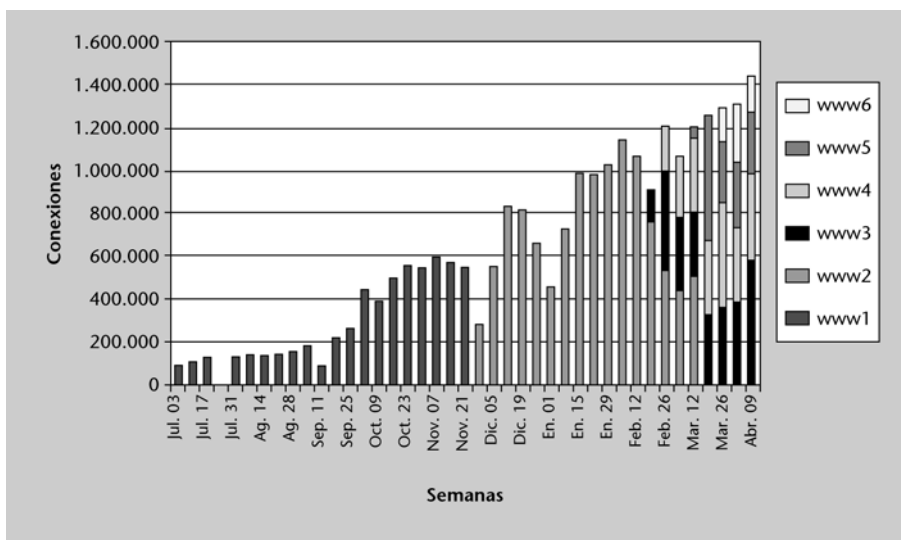
Las aplicaciones que ofrecen contenidos en Internet se enfrentan al reto de la escala: un solo servidor ante eventualmente millones de personas que pueden pedirle sus servicios todos a la vez: el proveedor de información debe poner tantos recursos como audiencia pueda tener.

En consecuencia, debe pagar más quien tiene algo más interesante que contar o quiere llegar a más. La web no funciona como una antena de radio: la potencia de emisión determina la cobertura, pero el número de receptores no la afecta; es más similar al teléfono: la capacidad se mide en número de líneas y esto determina el número de personas que pueden atenderse a la vez.

Por este motivo, puede ser necesario disponer de varios servidores para poder repartir la carga entre los mismos.

La primera página web pública, ahora ya fuera de funcionamiento, fue <http://info.cern.ch>. Sin embargo, la primera web con una demanda importante fue <http://www.ncsa.uiuc.edu>. Esta web tuvo que usar cuatro servidores replicados para satisfacer la demanda. En la siguiente gráfica puede verse la evolución del número de peticiones entre julio de 1993 (91.000 peticiones/semana) y abril de 1994 (1.500.000 peticiones/semana).

Figura 19



Crecimiento del número de peticiones semanales durante dos años. Cada tono de gris se corresponde con una máquina distinta. A mediados de febrero de 1994, diferentes máquinas empezaron a servir peticiones al mismo tiempo hasta llegar a cuatro.

Existen varios “trucos” para repartir peticiones entre diferentes máquinas:

- *Mirrors* con un programa que redirige la petición HTTP a la mejor réplica (podéis ver el ejemplo siguiente de Apache).
- Hacer que el servicio de nombres DNS devuelva diferentes direcciones IP (el método *round robin*). De esta manera, los clientes pueden hacer peticiones HTTP cada vez a una dirección IP distinta.
- Redirección en el nivel de transporte (conmutador de nivel 4, *L4 switch*): un encaminador mira los paquetes IP de conexiones TCP hacia un servidor web (puerto80) y las redirige a la máquina interna menos cargada.
- Redirección en el nivel de aplicación (conmutador de nivel 7, *L7 switch*): un encaminador que mira las conexiones HTTP y puede decidir a qué réplica contactar en función del URL solicitado. Muy complejo.
- Mandar todas las peticiones a un *proxy* inverso que responda o con contenido guardado en la memoria o que pase la petición a uno o varios servidores internos.

Si, además, las réplicas se sitúan cerca de los clientes, el rendimiento será mejor y más predecible. El inconveniente es que es difícil y caro montar un servicio web distribuido.

La Fundación Apache distribuye el servidor HTTPD Apache con la colaboración de más de doscientas réplicas distribuidas en todo el mundo. Aunque las páginas web se pueden consultar en el servidor origen, a la hora de bajar el programa hay un programa que calcula y redirige al visitante al servidor más próximo. De esta manera, las réplicas ayudan a repartir la carga a la vez que ofrecen un mejor servicio al cliente. Aquí el contenido se actualiza periódicamente.

4.1. Redes de distribución de contenidos

Han surgido empresas que se han dedicado a instalar máquinas en muchos lugares del mundo y algoritmos para decidir qué máquina es la más adecuada para atender peticiones según la ubicación del cliente y la carga de la red. Estas empresas venden este “servidor web distribuido” a varios clientes que pagan por poder disponer de un sistema de servicio web de gran capacidad que puede responder a demandas extraordinarias de contenidos.

Estos sistemas se denominan **redes de distribución de contenidos** (*content delivery networks* o CDN), y se pueden encontrar varias empresas que ofrecen este servicio: Akamai es la más importante.

www.google.com

Si se pregunta al DNS por www.google.com, la respuesta contiene varias direcciones IP: `nslookup www.google.com`.

¿Cómo ser *mirror* de Apache?

Apache pide que al menos se actualice el contenido dos veces por semana.

Las herramientas para hacerlo son las siguientes.

1) `rsync`: un programa que compara dos lugares remotos y transfiere los bloques o ficheros que hayan cambiado.

Ver:

<http://rsync.samba.org>.

2) `cvs`: un programa pensado para desarrollo de código a distancia, y que permite detectar e intercambiar diferencias.

Ver:

<http://www.cvshome.org>.

3) Apache como *proxy-cache*: configurar un servidor Apache para pasar y hacer *cache* de las peticiones. Orden: `ProxyPass / http://www.apache.org/CacheDefaultExpire 24`.

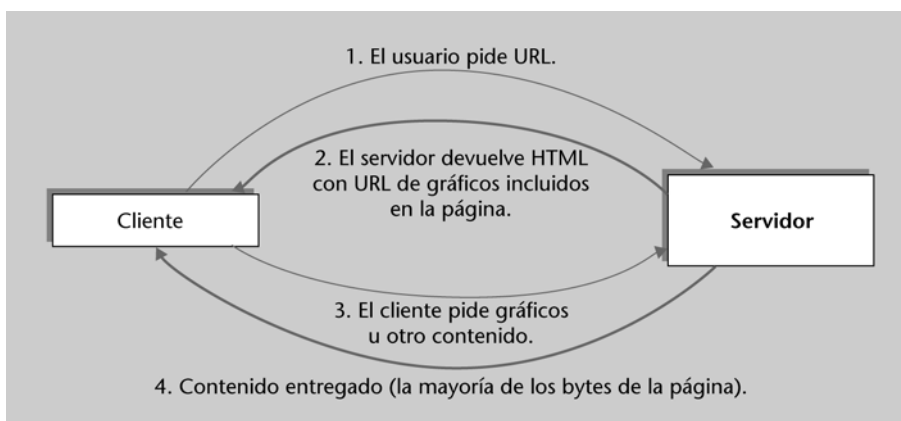
Una CDN es un *software* intermediario (*middleware*) entre **proveedores de contenidos y clientes web**. La CDN sirve contenido desde varios servidores repartidos por todo planeta. La infraestructura de la CDN está compartida por varios clientes de la CDN, y las peticiones tienen en cuenta la situación de la red para hacer que cada cliente web obtenga el contenido solicitado del servidor de la CDN más eficiente (habitualmente, una combinación del más próximo, el menos cargado y el que tiene un camino de mayor ancho de banda con el cliente).

Es como un servicio de “logística”: la CDN se encarga de mantener copias cerca de los clientes en sus propios almacenes (no en un punto central, sino en los extremos de la red). Para esto, debe disponer de multitud de puntos de servicio en multitud de proveedores de Internet (servidores *surrogate*: funcionalidad entre servidor *proxy-cache* y réplica). Por ejemplo, Akamai ofrecía en enero del 2007 unos 20.000 puntos de servicio en todo el mundo.

Algunas CDN, además, se sirven de enlaces vía satélite de alta capacidad (y retardo) para trasladar contenidos de un lugar a otro evitando la posible congestión de Internet. Aunque puedan no ser ideales para objetos pequeños, puede funcionar bien para objetos grandes (por ejemplo, *software*, audio, vídeo). Algunos ejemplos son las empresas Amazon 33, Castify y Real Networks.

Mientras que la transferencia de una página web normal funciona como sigue:

Figura 20



Fases de una petición web.

Una petición de una página web (documento HTML + algunos gráficos) hace distintas operaciones: 0.1) resolución DNS del nombre del servidor, 1.1) conexión TCP con el servidor, 1.2) solicitud del URL del documento, 2) el servidor devuelve el documento HTML que contiene referencias a gráficos incluidos en la página, 3) resolución DNS del nombre del servidor donde están los gráficos, que acostumbra a ser el mismo servidor que para la página HTML, 3.1) solicitud de los gráficos necesarios (puede repetirse distintas veces), 4) contenido servido y página visualizada por el cliente (navegador).

Una propuesta difícil de rechazar

La oferta de una CDN como Akamai a un proveedor de acceso a Internet (ISP) podría ser: “nos dejás en tu sala de máquinas el espacio equivalente para un frigorífico casero, nosotros te mandamos unas máquinas y tú las enchufas a la red eléctrica y a tu red (Internet). Nosotros las administramos”.

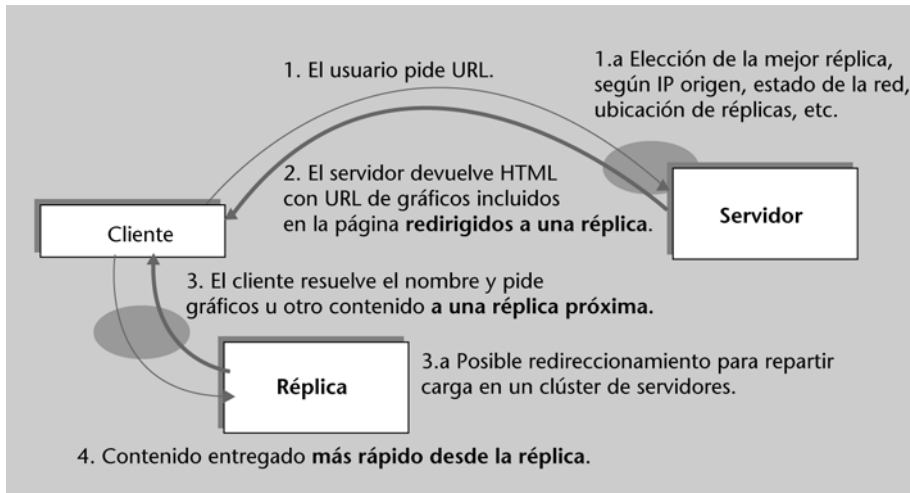
¿Qué ocurrirá? Todas las peticiones web a varios miles de empresas serán servidas por nuestras máquinas.

El ISP ahorra gasto de ancho de banda con Internet, pues sus usuarios no necesitarán ir a Internet para buscar algunos contenidos, que serán servidos por las máquinas de Akamai en el ISP.

Akamai cobra del proveedor de contenidos por servir el contenido más rápido y mejor que nadie.

Una petición a una CDN como Akamai aprovecha para tomar decisiones en cada paso de la petición:

Figura 21



Fases de una petición web usando una CDN.

La CDN puede actuar sobre lo siguiente.

- El DNS: cuando se resuelve un nombre, el servidor DNS responde según la ubicación de la dirección IP del cliente.
- El servidor web: cuando se pide una página web, se reescriben los URL internos según la ubicación de la dirección IP del cliente.
- Cuando se pide un objeto a la dirección IP de un servidor de la CDN, el conmutador (*switch*) de acceso a un conjunto de distintos servidores *proxy-cache*, o réplicas, puede redirigir la conexión al servidor menos cargado del grupo (todo el conjunto responde bajo una misma dirección IP).

Los pasos de la petición de una página web con gráficos podrían ser los siguientes:

0. El usuario escribe un URL (por ejemplo, `http://www.adobe.com`) en su navegador, y su navegador **resuelve** el nombre en DNS (`192.150.14.120`).

1. El usuario pide el URL a la dirección IP (`192.150.14.120`).

1.a El servidor web elige la mejor réplica, según IP origen, estado de la red y ubicación de las réplicas, o al menos clasifica al cliente en una zona geográfica determinada (por ejemplo, con Akamai decide que estamos en la zona del mundo o región “g”).

2. El servidor devuelve el HTML con el URL de los gráficos incluidos en la página (marcas ``), que se han **redirigido a una réplica**. De esta manera, los gráficos que constituyen el mayor número de bytes de una página web serán transferidos por la CDN. Por ejemplo,

```

```

Akamai

Es interesante visitar la web de Akamai:

<http://www.akamai.com>

3. El cliente **resuelve** el nombre y pide gráficos u otro contenido **a la réplica más cercana**.

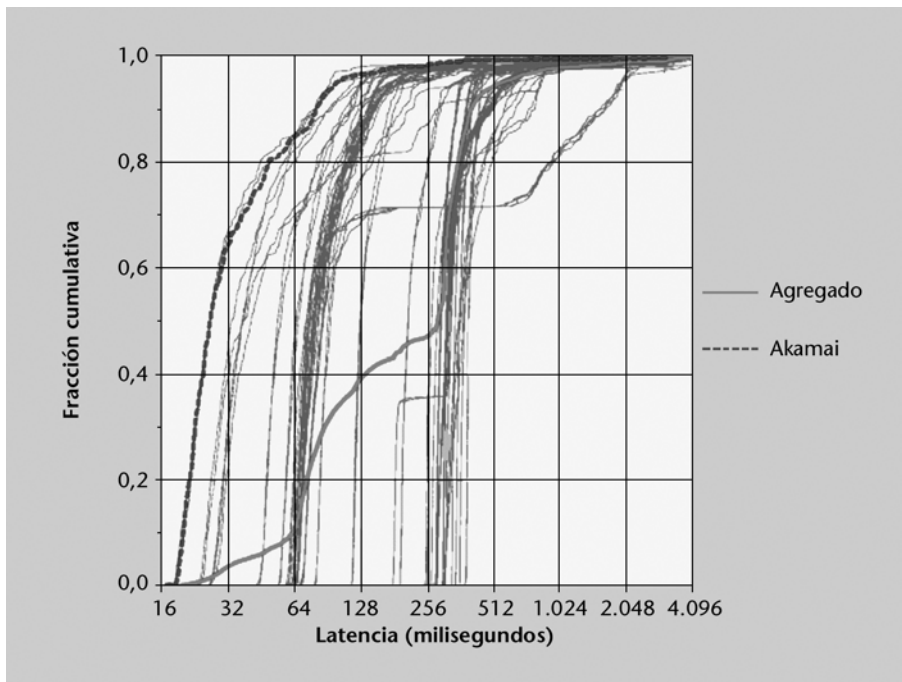
3.a Posible redirección con el DNS o por redirección de la conexión TCP (en el nivel del TCP o el HTTP: un conmutador de nivel 4 o 7), **reparto de la carga en un grupo de servidores**.

4. Contenido servido **más rápido desde una réplica**.

La ventaja de todo este montaje es que el usuario sigue viendo en su navegador un URL normal (el de la página HTML), el servidor de la empresa tiene *logs* con visitas a páginas HTML y el departamento de marketing puede hacer estadísticas, pero la mayoría del tráfico (los gráficos) lo sirve Akamai desde la proximidad del cliente.

Akamai mantiene información del estado de la red, de los *clusters*, de sus “servidores”. No siempre elige el “mejor posible”, pero sí uno de los mejores, y en cambio evita los peores servidores en cada momento.

Figura 22



Distribución acumulada de tiempo de respuesta (latencia) de distintos servidores Akamai.

La gráfica anterior muestra en líneas finas el tiempo que se tarda en obtener desde un lugar determinado un objeto de 4Kb pedido a muchos servidores de Akamai.

El eje *x*, en escala logarítmica, mide el tiempo (ms). El eje *y* mide la fracción acumulada de casos en los que el tiempo de descarga es inferior a un cierto valor. La línea gruesa continua (*agregado*) mide la media de todos los servidores, y la línea gruesa discontinua (*akamai*) muestra el tiempo de respuesta del servidor que selecciona Akamai en cada momento (prácticamente siempre selecciona el mejor).

La línea gruesa continua indica que si se hiciese una elección aleatoria, para el 20% (0,2) de los casos el tiempo de servicio sería inferior a 64 ms, pero para el 80% de los casos el tiempo sería inferior a 400 ms. Con la elección de Akamai, el 80% de las peticiones se sirven en menos de 50 ms. Si eligiéramos uno de los peores servidores, sólo podríamos decir que el 80% de las peticiones se sirven en menos de 1.000 ms.

Consecuencia: la decisión de Akamai es casi ideal, mucho mejor que la decisión aleatoria, y por supuesto que el peor caso (Ley de Murphy).

5. Computación orientada a servicios

Las aplicaciones web devuelven resultados que pueden corresponder al contenido de un archivo (contenidos estáticos) o ser el resultado de la ejecución de un programa (contenidos dinámicos). Las aplicaciones que generan contenidos dinámicos pueden requerir pequeñas o grandes dosis de computación y almacenamiento. Algunos ejemplos son los buscadores web, las tiendas en Internet, los sitios de subastas, los servicios de mapas o imágenes de satélite, que pueden tener que efectuar complejos cálculos, búsquedas, o servir enormes volúmenes de información.

Dentro de las organizaciones, también se utilizan sistemas complejos que procesan cantidades ingentes de datos y que presentan sus resultados mediante un navegador. Ejemplos son las aplicaciones financieras, científicas, de análisis de mercados, que tratan con enormes cantidades de datos que hay que recoger, simular, predecir, resumir, estimar, etc., para que unas personas puedan evaluar una situación y tomar decisiones.

Figura 23



Ejemplo de una aplicación de computación intensiva, en el que, a partir de unas fuentes de datos reales o simulados, se almacenan y procesan para la visualización de datos, análisis y toma de decisiones.

En estos sistemas, cada unidad autónoma de procesamiento puede agruparse en un servicio y la forma de interacción entre ellos suele estar basada en servicios *grid* o servicios web. En cuanto a la interacción con el usuario, todo ese conjunto de servicios se puede esconder tras un servidor web y una interfaz de usuario, que puede ser tradicional (basada en formularios html sencillos) o bien ser más interactiva y más parecida a una aplicación centralizada utilizando las capacidades avanzadas de los navegadores como AJAX.

En este modelo de aplicaciones o sistemas, el procesamiento está repartido entre el que ocurre en el navegador del usuario (relacionado con la presentación e interacción con el usuario), el servidor web (mediación entre una aplicación en el servidor y el navegador remoto) y otros servidores y servicios localizados potencialmente en otras máquinas, ubicaciones e incluso otras organizaciones siguiendo un modelo *grid*. Estos sistemas pueden tener una estructura fija o bien dinámica, en la que las herramientas y recursos se incorporan según la demanda (un modelo de uso de un *grid* llamado *utility computing*).

Sobre el AJAX, podéis ver el módulo 3, "Mecanismos de invocación".



5.1. Computación bajo demanda

Utility computing (el “servicio público” de computación) es un modelo técnico y de negocio, en el que los recursos informáticos se proporcionan bajo demanda y pagando según uso.

Difiere del modelo de computación convencional en que, bajo este modelo, no tienen que invertir en disponer de capacidad para el caso de máxima demanda, sino que pagan por el uso real de los recursos. La suma de varios clientes con diferentes demandas que usan ciertos recursos hace que se optimice su utilización. Este modelo de *grid* es comparable al de uso de la electricidad, gas, correos y muchos otros servicios públicos, así que se le podría llamar “servicio público de computación”. Atendiendo a su dependencia del uso, también se le llama “computación bajo demanda”.

La búsqueda en Internet

La búsqueda en Internet es una actividad intensiva en computación. El siguiente artículo muestra cómo se necesita un sistema masivamente replicado para dedicar a cada petición de búsqueda la máxima capacidad de computación necesaria para encontrar las referencias a cualquier cosa, en cualquier lugar de la web ordenado por relevancia.

Luiz Barroso; Jeffrey Dean; Urs Hoelzle (marzo, 2003). “Web Search for a Planet: The Google Cluster Architecture”. *IEEE Micro* (vol. 23, núm. 2, pág. 22-28).
<<http://labs.google.com/papers/googlecluster.html>>

Comparando con la red eléctrica, este modelo apuesta por no limitar la computación a la capacidad de una máquina aislada (con capacidad limitada de cálculo del procesador, almacenamiento de su disco duro, aplicaciones instaladas y licenciadas, electricidad de su batería), sino tomar computación, almacenamiento, aplicaciones y electricidad de los correspondientes servicios públicos.

Estas ideas no son nuevas, y en 1969 uno de los inspiradores de la red Internet dijo:

“As of now, computer networks are still in their infancy, but as they grow up and become sophisticated, we will probably see the spread of ‘computer utilities’ which, like present electric and telephone utilities, will service individual homes and offices across the country.”

Leonard Kleinrock (noviembre, 2005). “A vision for the Internet”. *ST Journal of Research* (vol. 2, núm. 1, pág. 4-5).

En este modelo, las aplicaciones web no quedan limitadas a la interacción navegador-servidor web, sino que se convierten en aplicaciones completamente distribuidas en términos de procesamiento, almacenamiento, donde múltiples componentes interactúan entre sí mediante el intercambio de datos, la invocación de servicios, y que requieren considerar en su diseño todos los conceptos presentados en esta asignatura: desde la estructura y organización de los componentes, la sincronización y los problemas de orden en la concurrencia, la fiabilidad y tolerancia a fallos, la réplica y el consenso entre las réplicas, el rendimiento, los mecanismos de invocación de servicios, etc.

Resumen

En este módulo se han presentado las formas con las que un servicio web o una aplicación asociada a un servidor web se puede organizar para atender la demanda que puede recibir de Internet.

Muchos indicadores de Internet continúan creciendo exponencialmente: el número de personas con acceso a Internet y el tráfico web, que hoy en día predomina sobre el resto de los protocolos. La popularidad de los contenidos sigue la ley de Zipf, la ley de la desigualdad extrema: muchas visitas para muy pocos lugares. Esto hace que la demanda que pueda recibir en un momento concreto un servidor web pueda ser exageradamente grande. Conocer las características principales de esta demanda ayuda a prever situaciones en el diseño de un servicio web.

La capacidad para servir peticiones es difícil de prever en un sistema tan complejo en el que influyen, entre muchos factores, la capacidad de la red y su carga, las características de toda la máquina, el sistema operativo con numerosos subsistemas o el servidor web. Es conveniente conocer la capacidad de servicio de nuestro servidor y observar su comportamiento con niveles de carga elevados usando herramientas de generación de tráfico y medida del comportamiento bajo una carga sintética, y también herramientas de visualización de estadísticas de demanda real a partir de diarios (*logs*), que nos permitan detectar sobrecargas, sintonizar o actualizar el conjunto adecuadamente.

Las aplicaciones en servidores web tienen dos componentes principales: el servidor web y el código adicional, que extiende el servidor para ofrecer servicios o contenidos “dinámicos”.

El servidor web es un sistema que se encarga de servir algunas de las muchas peticiones simultáneamente, por lo cual debe optimizarse la organización de toda esta actividad simultánea a partir de procesos, flujos y fibras.

Las aplicaciones web reciben peticiones del servidor y le devuelven un resultado para enviar al cliente. Los CGI son el modelo más sencillo y antiguo de interacción servidor-aplicación, en el que el servidor invoca un comando (ejecuta un proceso) para cada petición. *FastCGI* es una mejora de los CGI para que el código externo al servidor no deba ejecutarse con cada petición. Los *servlets*, la propuesta más reciente, proponen para Java un modelo muy completo y eficiente de gestión de concurrencia y duración de los procesos en el servidor.

Otro componente importante son los servidores intermedios entre navegadores y servidores web que guardan copias de los objetos que ven pasar desde un ser-

vidor hacia un navegador. Principalmente, permiten “acortar” peticiones sirviéndolas a medio camino del servidor, en la memoria, con objetos recibidos recientemente, hecho que reduce la congestión de Internet y la carga de los servidores.

Mientras que los servidores *proxy-cache* se suelen situar cerca de los lectores, la demanda global de ciertos contenidos o la tolerancia a fallos puede recomendar ofrecer un servicio web desde distintas ubicaciones. Hay diferentes maneras de montar un servicio distribuido: replicación o *mirroring*, DNS *round-robin*, redireccionamiento en ámbito de transporte o HTTP y servidores intermedios inversos.

Otra alternativa es contratar la provisión de servicio a una red de distribución de contenidos o CDN, que es una infraestructura comercial compartida por distintos clientes, con infinidad de puntos de presencia próximos a la mayoría de los usuarios de Internet, que se encargan de repartir y dirigir las peticiones web hacia los servidores menos cargados y más cercanos al origen de la petición.

La computación bajo demanda es un modelo más general que permite distribuir un sistema en varios componentes distribuidos, que se comunican a base de invocaciones a servicios, y el uso de recursos (computación, almacenamiento, servicios de aplicación), que se asignan dinámicamente según sea la necesidad o el volumen de demanda de sus usuarios.

Actividades

1. Averiguad si el proveedor de Internet que tiene la conexión a Internet disponible en vuestro PC tiene disponible algún servidor *proxy-cache* y, si es así, configurad vuestro navegador para utilizarlo. El uso del servidor ahorrará carga en algunos servidores que hayan servido antes el mismo contenido estático a otro usuario del servidor *proxy-cache*. Fijaos si se produce algún efecto apreciable con el uso del servidor *proxy-cache*.

Probad la memoria utilizando el comando telnet en el puerto del *proxy*. Se puede invocar el método GET de un objeto remoto:

```
GET http://www.apache.org http/1.1␣  
Host: www.apache.org␣  
␣
```

o el método *trace* del HTTP:

```
TRACE http://www.apache.org http/1.1␣  
Host: www.apache.org␣  
␣
```

y ver el resultado de la petición teniendo en cuenta los campos de cabecera de la respuesta que indican el camino seguido por la petición por medio de uno o varios servidores *proxy-cache*. Repetid la operación para ver si el contenido llega hasta el servidor o nos responde el servidor *proxy-cache* en lugar del servidor web.

2. Limpiad la memoria caché de vuestro navegador y configurad una memoria caché pequeña del navegador, visitad sitios con muchos gráficos y dejad la ventana abierta sobre la carpeta que contiene la caché en vuestro ordenador. Observad qué hace cuando está llena, qué objetos reemplaza e inferid una hipótesis sobre qué datos tiene en cuenta para gestionar la caché.

Ejercicios de autoevaluación

1. Considerando que una aplicación web eficiente se pueda construir utilizando *FastCGI* o *servlets*, indicad qué características podría tener una aplicación para que fuese preferible hacerla con *FastCGI*, y otra en la cual la mejor opción fuese *servlets*.

2. Qué diferencias pueden encontrarse al acceder por HTTP a un contenido web personalizado (distinto para cada persona, por ejemplo durante la compra de varios libros en una librería, donde para cada libro se muestra una ficha que incluye una foto y un capítulo de muestra en PDF) en el caso de:

a) conexión directa, b) conexión por medio de un *proxy-cache*, c) mediante una red de distribución de contenidos como Akamai.

Comentad el efecto sobre el retardo (tiempo en recibir el primer byte), tiempo de servicio (recibir el último byte), el gasto de recursos de red y la carga del servidor origen del contenido.

3. HTTP1.1 tiene reservado el código de error 305 (*Use proxy*) para que los servidores de HTTP puedan no aceptar conexiones directas (que no hayan pasado antes por un *proxy*). Indicad las razones y una situación en la que este código de error pueda ser de utilidad. Describid los pasos que seguiría un cliente que intentara inicialmente acceder a un servidor que no acepta conexiones directas.

4. Hay distintas maneras de repartir la carga entre varios servidores de HTTP: a) redirección a un servidor HTTP que tiene una réplica del contenido, b) hacer que el servidor de DNS resuelva en cada momento a un servidor distinto (redirección DNS), c) la redirección a un *proxy* de la pregunta anterior. Haced una tabla que compare qué limitaciones tiene cada una, y proponed una situación óptima para cada caso.

Solucionario

1. Una aplicación con *FastCGI*: una aplicación no escrita en Java, o que no necesite interactuar excesivamente con el servidor, o que deba ser extremadamente eficiente o pequeña.
Una aplicación con *servlets*: una aplicación escrita en Java, o que necesite un modelo de proceso sofisticado, o que deba interactuar con el servidor web más estrechamente o que tenga que ser transportable con facilidad a otros servidores y/o otras máquinas.

2. Efecto sobre el retardo (tiempo en recibir el primer byte): $a \leq b$ aunque similares, excepto en los casos en los que el objeto se encuentre en algún servidor *proxy-cache* y se pueda servir inmediatamente sin verificar con el servidor (objeto estático, que cambia poco o nada y obtenido recientemente), en d ; si el contenido lo sirve la CDN, seguramente será muy rápido. Tiempo de servicio (recibir el último byte): a, b como mínimo igual, b puede ser mucho más rápido si el contenido se sirve del servidor intermediario. d probablemente será mucho más rápido, ya que se sirve de un lugar próximo al cliente, salvo que el cliente tenga limitaciones grandes de velocidad de conexión a Internet.

Gasto de recursos de red: b, c ahorrar recursos de red, ya que pueden llevar el contenido desde más cerca que a .

Carga del servidor origen del contenido: b ahorra peticiones repetitivas de contenido estático desde el grupo de clientes que usan los servidores *proxy-cache*. En d el servidor puede haber delegado prácticamente por completo el servicio de aquel objeto a la CDN.

3. Cuando un servidor está sobrecargado puede devolver este código de error para indicar al cliente que en lugar de conectarse directamente utilice un servidor intermediario. Esto podría ayudar a agrupar las peticiones y, por lo tanto, a reducir la carga del servidor. El cliente debería saber qué servidor intermediario tiene cerca que le pueda proporcionar servicio, y debería conectarse con él y repetir la petición vía servidor intermediario. El inconveniente es que el cliente deba tener un servidor intermediario accesible o que lo tenga que descubrir. Una alternativa útil sería que este código de error pudiese “orientar” al cliente dándole la información de un servidor intermediario que lo pudiese ayudar (un servidor intermediario inverso, por ejemplo, que aceptaría conexiones de cualquier cliente que pidiese páginas de aquel servidor).

4. La respuesta se podría expresar en una tabla como la siguiente:

	A (a réplica)	B (DNS)	C (redirección)
Limitaciones	Hay que tener preparada la réplica antes. Propagar y actualizar el contenido para que sea consistente. Es mayor el retardo, pues hay que redirigir (establecer una nueva conexión TCP).	Ha de replicarse un dominio completo (no sólo un conjunto de páginas). Un cliente puede cachear la resolución y cargar más un servidor que otro. Si se reduce el TTL de las respuestas, se genera más tráfico DNS. Hay que modificar todos los servidores de DNS de este dominio (no si sólo se usa <i>round robin</i>).	Implica establecer otra conexión TCP, con el retardo que esto supone. El cliente puede no saber tratar el código de error o no tener conocimiento de un <i>proxy</i> , con lo que no se puede obtener la página solicitada. Es necesario, además, asegurarse de que el contenido esté sincronizado.
Situación óptima	Contenido de alta demanda que cambia poco (<i>mirror</i> de Apache).	Dominio completo replicado en varios servidores.	Situación extraña. Este código de error no está bien especificado y, por tanto, no está implementado correctamente. Serviría para reducir la carga de un único servidor sin usar réplicas.

Glosario

CDN *f* Véase content delivery/distribution network.

CGI *f* Véase interfaz común de pasarela.

common gateway interface *f* Véase interfaz común de pasarela.

computación grid *f* Modelo de computación distribuida o paralela en el que un sistema puede repartir sus funciones entre componentes que se comunican en red y utilizan recursos o servicios pertenecientes a diversas organizaciones.

content delivery/distribution network *f* Red de servidores que sirve páginas web según la ubicación de los usuarios, el origen de la página web y el estado de carga de la red. Las páginas que sirve una CDN están ubicadas (posiblemente en forma de caché) en distintos servidores repartidos por varias ubicaciones. Cuando un usuario pide una página que está alojada en una CDN, la CDN redirecciona la petición desde el sitio web original a un servidor de la CDN que sea prácticamente óptimo en aquel momento para este cliente, teniendo en

cuenta distintos factores: distancia, carga de la red, carga del servidor, presencia del contenido solicitado, etc. Es muy útil para sitios web con mucho tráfico e interés global. Cuanto más próximo geográficamente esté el servidor de la CDN del cliente, más rápido recibirá el contenido y menos influencia tendrá la carga de la red. La presencia de la CDN puede ser transparente (invisible) para el usuario.

sigla: CDN

interfaz común de pasarela *f* Interfaz que especifica cómo transferir información entre un servidor web y un programa. El programa recoge datos que le pasa el servidor web y que vienen de un navegador, y devuelve datos en forma de documento. El programa se puede escribir en cualquier lenguaje que se pueda ejecutar en la máquina del servidor web. Un problema es que para cada petición se debe cargar y ejecutar el programa asociado, hecho que significa un gran gasto de recursos del servidor.

en common gateway interface

sigla: CGI

proxy *m* Véase servidor intermediario.

réplica *f* Copia de una entidad (documento, conjunto de documentos, servidor) que se mantiene actualizada o sincronizada con otras réplicas de un conjunto. Los cambios realizados en una réplica se aplican y propagan al resto de las réplicas de manera automática y transparente para el usuario.

servidor intermediario *m* Servidor situado entre un cliente y un servidor real. Intercepta las peticiones del servidor real para ver si puede satisfacer la respuesta. Si no, reenvía la petición al servidor real. Los servidores intermediarios pueden tener dos propósitos: mejorar el rendimiento (reducir el camino hasta la información: servidores *proxy-cache*) o filtrar peticiones (prohibir el acceso a ciertos contenidos web, o transformar el formato).

en proxy

servlet *m* Miniaplicación Java (o *applet*) que se ejecuta en un servidor. El término se acostumbra a aplicar a servidores web. Son una alternativa a los programas CGI. Los *servlets* son persistentes y, una vez invocados, pueden quedarse cargados en la memoria y servir distintas peticiones, mientras que las CGI se cargan, ejecutan y desaparecen para atender una sola petición.

transparente *adj* Invisible en informática. Una acción es transparente si se produce sin efecto visible. La transparencia se suele considerar una buena característica de un sistema, porque aísla al usuario de la complejidad del sistema.

Bibliografía

Krishnamurthy, B.; Rexford, J. (2001). *Web Protocols and Practice: HTTP/1.1, Networking Protocols, Caching, and Traffic Measurement*. Cambridge (EE.UU.): Addison-Wesley. ISBN 0-201-71088-9. Un libro exhaustivo y detallado sobre el funcionamiento de la web más allá de la presentación en los navegadores. Estudia los detalles de las diferentes versiones del protocolo HTTP, su rendimiento, la interacción con los otros servicios involucrados: DNS, TCP, IP, la evolución histórica de los componentes de la web, *scripts*, buscadores, galletas (*cookies*), autenticación, técnicas para recoger y analizar tráfico web, *proxy-caches* web e interacción entre *proxy-caches*.

Luotonen, A. (1998). *Web Proxy Servers*. Londres: Addison-Wesley. ISBN 0-13-680612-0. Ari desarrolló el servidor *proxy-cache* del CERN, y también el de Netscape. Habla de los detalles de la estructura interna (procesos) de los servidores, cooperación entre servidores, mediación, filtrado, monitorización, control de acceso, seguridad, aspectos que afectan al rendimiento de un servidor intermediario, etc.

Luotonen, A.; Altis, K. (1994). *World-Wide Web Proxies*. Actas de la Conferencia WWW94. Es el artículo histórico sobre *proxy-cache* en web, basado en la experiencia con el servidor *proxy-cache* web del CERN.

Rabinovich, M.; Spatscheck, O. (2002). *Web Caching and Replication*. Boston (EE.UU.): Addison-Wesley. ISBN 0-201-61570-3.

Un libro exhaustivo y detallado sobre los avances recientes en cache y replicación para la web. Los capítulos 4: "Protocolos de aplicación para la web", 5: "Soporte HTTP para *proxy-cache* y replicación" y 6: "Comportamiento de la web" ofrecen una buena visión general del problema y los mecanismos que influyen en el mismo. La parte II analiza detalladamente los mecanismos de *proxy-cache* y la parte III, los mecanismos de replicación en la web.

Hay muchos libros específicos y detallados sobre cada tecnología explicada: sobre las especificaciones, sobre cómo utilizarla, cómo instalarla y administrarla, referencias breves, etc. Entre otras, la editorial O'Reilly tiene muchos libros bastante buenos y muy actualizados sobre diferentes temas de la asignatura. Son muy detallados, mucho más de lo que se pretende en la asignatura, pero en la vida profesional pueden ser de gran utilidad para profundizar en los aspectos prácticos de un tema (<http://www.ora.com>).

