UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CIENCIAS

CONSTRUCCION DE UN METABUSCADOR DE DOCUMENTOS EN INTERNET (AMOXCALLI)

T E S I S

Que para obtener el título de

M A T E M A T I C O

P r e s e n t a;

LUIS HERNANDEZ ORTEGA

DIRECTOR DE TESIS: M. en C. JOSE DE JESUS GALAVIZ CASAS

FACULTAD DE CIENCIAS
UNAM

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

UNIVERSIDAD NACIONAL
AUTÓNOMA DE
MÉXICO

M. EN C. ELENA DE OTEYZA DE OTEYZA
Jefa de la División de Estudios Profesionales de la
Facultad de Ciencias
Presente

Comunicamos a usted que hemos revisado el trabajo escrito:
"Construcción de un metabuscador de documentos en Internet (Amoxcalli)"

realizado por Luis Hernández Ortega

con número de cuenta 9109855-9 , quién cubrió los créditos de la carrera de Matemáticas

Dicho trabajo cuenta con nuestro voto aprobatorio.

Atentamente

Director de Tesis Propietario	M.en C. José de Jesús Galaviz Casas
Propietario	Mat. Mónica Leñero Padierna
Propietario	M. en C. Ricardo Paramont Hernández García
Suplente	M. en C. Juan Jesús Gutiérrez García
Suplente	Mat. Carlos Rivera Ortega

Consejo Departamental de Matemáticas

M. en C. Alejandro Bravo Mojica

Para mi madre Ninfa
y mis hermanos Ricardo y Rodrigo.

Universidad Nacional Autónoma de México.
Facultad de Ciencias

Título de tesis: "Construcción de un metabuscador de documentos en Internet (Amoxcalli)"

Alumno: Luis Hernández Ortega.

ÍNDICE.

INTRODUCCIÓN

Introducción.

La World Wide Web está revolucionando la manera en que la gente accede a la información y ha abierto nuevas posibilidades en áreas tales como bibliotecas digitales, diseminación y recuperación de información científica y general, educación, comercio, entretenimiento, gobierno y salud. La cantidad de información publicada en la Web está incrementándose rápidamente, con lo que se ha estimulado la investigación y desarrollo en el tratamiento y recuperación de información, y ha promovido diversos motores de búsqueda[1].

Las limitaciones de los servicios de búsqueda han conducido a la introducción de motores de metabúsqueda. Un motor de metabúsqueda navega en la Web haciendo consultas a diversos buscadores tales como Lycos, Excite, AltaVista, EuroSeek o HotBot. Las principales ventajas de los actuales metabuscadores son la habilidad para combinar los resultados de múltiples buscadores y la habilidad para proporcionar una interfaz de usuario consistente para hacer consulta sobre estos motores.

La idea de interrogar y cotejar resultados de múltiples bases de datos no es nueva, compañías como PLS, Lexis-Nexis, Dialog y Verity han creado productos que integran resultados de múltiples bases de datos heterogéneas. Existen otros servicios de metabúsqueda de la Web tal como el popular y útil servicio MetaCrawler. Servicios similares a MetaCrawler incluyen SavvySearch e InfoseekExpress.

Los motores de metabúsqueda pueden introducir sus propias deficiencias, por ejemplo: pueden tener dificultad para clasificar la lista de resultados. Si un motor regresa muchos documentos poco relevantes, esto puede disminuir la eficacia del metabuscador para proporcionar los resultados. Muchos de los motores de metabúsqueda también limitan el número de resultados que pueden obtenerse y típicamente no soportan todas las características de consulta de cada buscador.

Ante este panorama general de los servicios de búsqueda y en particular de los metabuscadores surge la necesidad de entender detalladamente el entorno en que estas aplicaciones trabajan y como es que lo hacen, como principio fundamental para el desarrollo de aplicaciones posteriores en las que se desee hacer algún tratamiento de la información del Web. En este trabajo se describirá el entorno de estas aplicaciones y sugerirá una tecnología para el desarrollo de ellas, esto será llevado a cabo mediante el diseño e implantación de un metabuscador.

A lo largo de este trabajo se hará una breve descripción de los principales conceptos involucrados en el medio en que un metabuscador trabaja, luego se procederá a justificar la tecnología que se decidió utilizar para la construcción del metabuscador (**Amoxcalli**), y finalmente se explicará detalladamente cada uno de los pasos que se llevaron a cabo para la construcción de esta aplicación.

[1] **Motor de búsqueda**. Programa especializado en la búsqueda de información en Internet.

Capítulo 1

Panorama de Internet.

Red Local (LAN: Local Area Network). Es una red dentro de un mismo edificio, como por ejemplo las redes de alumnos o de profesores de la Facultad de Ciencias.

Red de campus (CAN: Campus Area Network). Es una red que une distintos edificios dentro de una zona geográfica limitada, por ejemplo el campus de una universidad. De hecho todos los cables por los que circula la información son privados.

Red de ciudad (MAN: Metropolitan Area Network). Se trata de una red que une distintos edificios dentro de un área urbana. En la transmisión de la información interviene una empresa de telecomunicaciones, que podría ser de ámbito local o regional.

Red de área extensa (WAN: Wide Area Network). En este caso la red puede unir centros dispersos en una zona geográfica muy amplia, en ocasiones por todo el mundo. Es la red típica de las empresas multinacionales. En la transmisión de la información deberán intervenir múltiples empresas de telecomunicaciones. Internet puede ser considerada como la WAN más conocida y extensa que existe en la actualidad.

1.3 Protocolo TCP/IP.

Lo que permite que computadoras remotas con procesadores y sistemas operativos diferentes se entiendan y en definitiva que Internet funcione como lo hace en la actualidad, es un conjunto de instrucciones o reglas conocidas con el nombre de protocolo. La Internet utiliza varios protocolos, pero los que están en la base de todos los demás son el ***Transport Control Protocol*** (**TCP**) y el ***Internet Protocol (IP)***, o en definitiva ***TCP/IP*** para abreviar. Se trata de una serie de reglas para transmitir de una computadora a otra los datos electrónicos descompuestos en paquetes, asegurándose de que todos los paquetes llegan y son ensamblados correctamente en su destino. Todas las computadoras en Internet utilizan el protocolo TCP/IP, y gracias a ello se consigue eliminar al menos en parte la barrera de la heterogeneidad de las computadoras, sistemas operativos y resolver los problemas de direccionamiento.

1.4 Servicios.

Sobre la infraestructura de transporte de datos que proporciona el protocolo TCP/IP se han construido protocolos específicos que permiten por ejemplo enviar correo electrónico ***(SMTP)***, establecer conexiones y ejecutar instrucciones en máquinas remotas ***(TELNET)***, acceder a foros de discusión o *news* ***(NNTP)***, transmitir archivos ***(FTP)***, conectarse con un servidor web ***(HTTP)***, etc. A estas capacidades de Internet se les llama servicios. A continuación se revisan los más conocidos.

1.4.3 Transferencia de archivos (Ftp)

El servicio proporcionado por ***FTP (File Transfer Protocol)*** es una parte importante de Internet. FTP permite transferir bidireccionalmente cualquier tipo de archivos con cualquier computadora que cuente con un servidor FTP. Se pueden transferir archivos ejecutables, de gráficos, sonido, video o cualquier otro tipo. Al igual que Telnet, FTP establece conexiones síncronas y permanentes. Para utilizar el servicio FTP suele ser necesario proporcionar un login y un password aunque es muy frecuente encontrar servidores FTP abiertos a todo el mundo y que permiten sólo lectura de archivos. Muchas empresas como **Microsoft**, **Sun**, **Netscape**, etc. utilizan este sistema para distribuir software de forma gratuita. En ocasiones, para conectarse a este tipo de servicio hay que dar como nombre de usuario la palabra "anonymous", y como password la propia dirección de correo electrónico.

1.4.4 Grupos de discusión (News).

Los news groups o grupos de discusión son foros globales para la discusión de temas específicos. Son utilizados con el fin de discutir e intercambiar información, que versa sobre gran variedad de temas. Estas discusiones suelen ser públicas, es decir, accesibles a personas de todo el mundo interesadas en el tema. Las discusiones pueden ser libres (cada usuario que desea intervenir lo hace sin limitación alguna) o moderadas (un moderador decide si las intervenciones se incluyen o no).

1.4.5 World Wide Web.

La World Wide Web, o simplemente Web, es el sistema de información más completo y actual, que une tanto elementos multimedios como hipertexto. De hecho, tomando el todo por la parte, con mucha frecuencia la Web se utiliza como sinónimo de Internet. La World Wide Web (WWW) es el resultado de cuatro ideas o factores:

1. La idea de Internet y los protocolos de transporte de información en que está basada.
2. El concepto de Ted Nelson de un sistema de hipertexto, extendido a la red.
3. La idea de programas cliente que interaccionan con programas servidores capaces de enviar la información en ellos almacenada. En la Web, esto se hace mediante el protocolo HTTP (HyperText Transfer Protocol).
4. El concepto de lenguaje marcado *(Markup language)* y más en concreto del lenguaje ***HTML (HyperText Markup Language)***, el lector interesado puede consultar [20].

HTML es una herramienta fundamental de Internet. Gracias al hipertexto, desde una página Web se puede acceder a cualquier otra página Web almacenada en un servidor HTTP situado en cualquier parte del mundo. Todo este tipo de operaciones se hacen mediante un programa llamado browser o navegador, que básicamente es un programa que reconoce el lenguaje HTML, lo procesa y lo representa en pantalla con el formato más adecuado posible.

En resumen, un URL es una manera conveniente y sucinta de referirse a un archivo o a cualquier otro recurso electrónico.

La sintaxis genérica de los URLs es la que se muestra a continuación:
`método://servidor.dominio/ruta-completa-del-archivo` donde método es una de las palabras que describen el servicio: http, ftp, news, etc. Enseguida se verá la sintaxis específica de cada método, pero antes conviene añadir unas breves observaciones:

En ocasiones el URL empleado tiene una sintaxis como la mostrada, pero acabada con una diagonal (/). Esto quiere decir que no se apunta a un archivo, sino a un directorio. Según como esté configurado, el servidor devolverá un listado de archivos y subdirectorios de ese directorio para poder acceder al que se desee, un archivo por omisión que el servidor busca automáticamente en el directorio (comúnmente llamado *Index.htm* o *Index.html*) o quizás impida el acceso si no se conoce exactamente el nombre del archivo al que se quiere acceder (como medida de seguridad).

A continuación se muestran las distintas formas de construir los URLs según los distintos servicios de Internet.

URLs del protocolo HTTP.

Como ya se ha dicho, HTTP es el protocolo específicamente diseñado para la World Wide Web. Su sintaxis es la siguiente:
http://host<:puerto>/<ruta>

donde host es la dirección del servidor WWW, el puerto indica a través de que "entrada" el servidor atiende los requerimientos HTTP (puede ser omitido, en cuyo caso se utiliza el valor por omisión 80) y la ruta indica al servidor la trayectoria al archivo que se desea cargar (el camino es relativo a un directorio raíz indicado en el servidor HTTP).

Así, por ejemplo, *http://www.msn.com/index/prev/welcome.htm* accede a la Web de Microsoft Network, en concreto al archivo welcome.htm (cuya ruta de acceso es index/prev).

URLs del protocolo Telnet.

El URL necesario para crear una sesión *Telnet* en un servidor remoto en Internet, se define mediante el protocolo Telnet. Su sintaxis es la siguiente:

`telnet://<usuario><password>@<host><:puerto>/` donde los parámetros usuario y password pueden ser omitidos si el servidor no los requiere, host identifica la computadora remota con la que se va a realizar la conexión y port define el puerto por el que dicho servidor atiende los requerimientos de servicios Telnet (puede ser igualmente omitido, siendo su valor por defecto 23).

Nombres específicos de archivos.

Es posible acceder también directamente a un archivo concreto mediante el método file, esto suele confundirse con el servicio ftp; la diferencia radica en que ftp es un servicio específico para la transmisión de archivos, mientras que file deja la forma de traer los archivos a la elección del cliente, que en algunas circunstancias podría ser el método ftp. La sintaxis para el método file es la que se muestra a continuación:
`file://<host>/<ruta-de-acceso>`

Como siempre, host es la dirección del servidor y ruta-de-acceso es el camino jerárquico para acceder al documento (con una estructura directorio/directorio/.../nombre del archivo). Si el parámetro host se deja en blanco, se supondrá por defecto localhost, es decir, que se van a traer archivos de la misma computadora.

Capítulo 2

Infraestructura y tecnologías para el desarrollo de aplicaciones sobre Internet.

En el tercer nivel la lógica de los procesos se divide entre los distintos componentes del cliente y del servidor. El diseñador de la aplicación debe definir los servicios y las interfaces del sistema de información de forma que los papeles de cliente y servidor sean intercambiables, excepto en el control de los datos que es responsabilidad exclusiva del servidor. En este tipo de situaciones se dice que hay un proceso distribuido o cooperativo.

En el cuarto nivel el cliente realiza tanto las funciones de presentación como los procesos. Por su parte, el servidor almacena y gestiona los datos que permanecen en una base de datos centralizada. En esta situación se dice que hay una gestión de datos remota.

En el quinto y último nivel, el reparto de tareas es como en el anterior y además el gestor de base de datos divide sus componentes entre el cliente y el servidor. Las interfaces de ambos están dentro de las funciones del gestor de datos y, por lo tanto, no tienen impacto en el desarrollo de las aplicaciones. En este nivel se da lo que se conoce como bases de datos distribuidas.

2.1.2 Características físicas

La figura 2 da una idea de la estructura física de conexión entre las distintas partes que componen una arquitectura cliente / servidor. La idea principal consiste en aprovechar la potencia de las computadoras personales para realizar sobre todo los servicios de presentación y, según el nivel, algunos procesos o incluso algún acceso a datos locales. De esta forma se descarga al servidor de ciertas tareas para que pueda realizar otras más rápidamente.

También existe una plataforma de servidores que sustituye a la computadora central tradicional y que da servicio a los clientes autorizados. Incluso a veces la antigua computadora central se integra en dicha plataforma como un servidor más. Estos servidores suelen estar especializados por funciones (seguridad, cálculo, bases de datos, comunicaciones, etc.), aunque dependiendo de las dimensiones de la instalación se pueden reunir en un servidor una o varias de estas funciones.

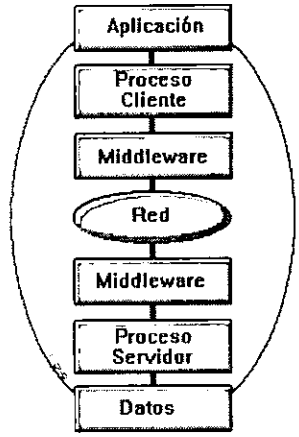

Figura 2. Estructura física de la arquitectura cliente/servidor.

- Mediante la integración de las aplicaciones cliente/servidor con las aplicaciones personales de uso habitual, los usuarios pueden construir soluciones particularizadas que se ajusten a sus necesidades cambiantes.
- Una interfaz gráfica de usuario consistente reduce el tiempo de aprendizaje de las aplicaciones.

- Menor costo de operación:

  - Permite un mejor aprovechamiento de los sistemas existentes, protegiendo la inversión. Por ejemplo, compartir servidores (habitualmente caros) y dispositivos periféricos (como impresoras) entre máquinas clientes permite un mejor rendimiento del conjunto.
  - Proporciona un mejor acceso a los datos. La interfaz de usuario ofrece una forma homogénea de ver el sistema, independientemente de los cambios o actualizaciones que se produzcan en él y de la ubicación de la información.
  - El movimiento de funciones desde una computadora central hacia servidores o clientes locales origina el desplazamiento de los costes de ese proceso hacia máquinas más pequeñas y por tanto, más baratas.

- Mejora en el rendimiento de la red:

  - Las arquitecturas cliente/servidor eliminan la necesidad de mover grandes bloques de información por la red hacia las computadoras personales o estaciones de trabajo para su proceso. Los servidores controlan los datos, procesan peticiones y después transfieren sólo los datos requeridos a la máquina cliente. Entonces, la máquina cliente presenta los datos al usuario mediante interfaces amigables. Todo esto reduce el tráfico de la red, lo que facilita que pueda soportar un mayor número de usuarios.
  - Tanto el cliente como el servidor pueden escalarse para ajustarse a las necesidades de las aplicaciones. Los CPUs utilizados en los respectivos equipos pueden dimensionarse a partir de las aplicaciones y el tiempo de respuesta que se requiera.
  - La existencia de varios CPUs proporciona una red más fiable: un fallo en uno de los equipos no significa necesariamente que el sistema deje de funcionar.

Simple

Al ser C y C++ los lenguajes mas difundidos, Java se diseñó para ser parecido a ellos y así facilitar un rápido y fácil aprendizaje.

Una característica agregada que simplifica la programación en este lenguaje es la asignación y liberación automática de memoria llevada a cabo por el recolector de basura (garbage collector). El recolector periódicamente libera memoria que no es referenciada y como es un hilo[4] (thread) de baja prioridad, cuando entra en acción permite liberar bloques de memoria muy grandes, lo que reduce la fragmentación de la memoria.

Java reduce en un 50% los errores más comunes de programación con lenguajes como C y C++ al eliminar muchas de las características de éstos, entre las que destacan:

aritmética de apuntadores
registros (struct)
definición de tipos (typedef)
macros (#define)
necesidad de liberar memoria (free)

Otro de los aspectos que le conceden simplicidad al lenguaje es el hecho de ser pequeño. El intérprete[5] de Java que hay en este momento es muy pequeño, solamente ocupa 215 Kb de RAM.

Orientado a objetos.

Las necesidades de los sistemas distribuídos basados en la arquitectura cliente/servidor coinciden con los paradigmas de encapsulación y entrega de mensajes de la tecnología orientada a objetos. Para poder abarcar la creciente complejidad de los entornos basados en redes, surge la necesidad de adoptar los conceptos de la orientación a objetos.

Java está completamente orientado a objetos, proporcionando los mecanismos para que el programador use todas las técnicas de diseño y programación orientados a objetos, así como el acceso a bibliotecas de clases para obtener los tipos de datos básicos, procedimientos de entrada/salida, comunicaciones a través de red, protocolos de Internet (TCP/IP, HTTP y FTP) y funciones para desarrollar interfaces de usuario. Todas estas bibliotecas pueden ser extendidas para modificar su comportamiento y adaptarlo a las necesidades del programador.

Otra de las funciones inexistentes en C++, que incluye Java es la resolución de métodos[6] en forma dinámica. Esta característica deriva del lenguaje **Objective C**, propietario del

[4] **Hilo** (thread). Un hilo es un flujo secuencial de control dentro de un programa.
[5] **Intérprete.** Programa que lee instrucción por instrucción de un programa escrito en un determinado lenguaje y las ejecuta directamente.
[6] **Resolución de métodos.** Es el proceso de obtener una referencia al procedimiento que implementa un método particular de un objeto particular en tiempo de ejecución.

- El acceso a los campos de un objeto se sabe que es legal: publico, privado y protegido (public, private, protected).
- No hay ningún intento de violar las reglas de acceso y seguridad establecidas.

El cargador de clases de Java también ayuda a mantener su seguridad separando el espacio de nombres del sistema de archivos local, del de los recursos procedentes de la red. Esto limita cualquier aplicación del tipo Caballo de Troya[7], ya que las clases se buscan primero entre las locales y luego entre las procedentes del exterior.

Las clases importadas de la red se almacenan en un espacio de nombres privado, asociado con el origen. Cuando una clase del espacio de nombres privado accede a otra clase, primero se busca en las clases predefinidas (del sistema local) y luego en el espacio de nombres de la clase que hace la referencia. Esto imposibilita que una clase suplante a una predefinida.

En resumen, las aplicaciones de Java resultan extremadamente seguras, ya que no acceden a zonas delicadas de memoria o de sistema, con lo cual evitan la operación de programas maliciosos. Java no posee una semántica específica para modificar la pila de programa, la memoria libre o utilizar objetos y métodos de un programa sin los privilegios del núcleo[8] *(kernel)* del sistema operativo.

Para el caso de **Applets**[9], Java imposibilita para ellos abrir archivos de la máquina donde son ejecutados (siempre que se realizan operaciones con archivos, éstas trabajan sobre el disco duro de la máquina de donde partió el applet), no permite ejecutar ninguna aplicación nativa de otra plataforma e impide que se utilicen otras computadoras como puente, es decir, nadie puede utilizar nuestra maquina para hacer peticiones o realizar operaciones con otra.

Portable.

Más allá de la portabilidad básica por ser de arquitectura neutral, Java implanta otros estándares de portabilidad para facilitar el desarrollo. Los enteros son siempre enteros y además, enteros de 32 bits en complemento a 2. Además, Java construye las interfaces de usuario a través de un sistema abstracto de ventanas que pueden ser implantadas en entornos Unix, PC o Mac.

[7] **Caballo de Troya.** Es un programa que tiene una apariencia amigable, sin embargo su propósito es malicioso.
[8] **Núcleo del sistema operativo.** Es el encargado de que el software y el hardware de una computadora puedan trabajar juntos.
[9] **Applet.** Es un programa Java que corre en un navegador.

Interpretado

Para que Java pueda ser un lenguaje independiente de la plataforma es tanto interpretado como compilado. Y en esto no hay contradicción, el código fuente escrito con cualquier editor (extensión .java) se compila generando el byte-code (extensión .class). Este código intermedio es de muy bajo nivel, pero sin alcanzar las instrucciones máquina propias de cada plataforma. El byte-code corresponde al 80% de las instrucciones de la aplicación. Ese mismo código es el que se puede ejecutar sobre cualquier plataforma, para la cual ha sido montado el run-time (implementación de la JVM) que sí es completamente dependiente de la máquina y del sistema operativo; éste interpreta dinámicamente el byte-code y añade el 20% de instrucciones que faltaban para su ejecución, consiguiendo con ello la independencia de plataforma. La figura 3 ejemplifica este comportamiento.

Soporte para conexión a red.

Java se ha construido con extensas capacidades de interconexión TCP/IP. Existen bibliotecas de rutinas para acceder e interactuar con protocolos como HTTP y FTP. Esto permite crear aplicaciones que accedan a información de la red con tanta facilidad como a los archivos locales.

Multihilos.

Hay muchas cosas en el mundo real que ocurren al mismo tiempo. Multihilos (Multithreading) es una forma de construir aplicaciones que realizan muchas actividades simultáneas.

Java tiene un conjunto sofisticado de primitivas de sincronización que están basadas en el paradigma de monitores y variables de condición introducido por C.A.R. Hoare**[11]**. Integrando estos conceptos en el lenguaje (más que en las clases) los hilos en Java resultan más fácil de usar y más robustos. Parte de este estilo de integración fue tomado del sistema Cedar/Mesa de Xerox.

El hecho de ser multihilos deriva en un mejor rendimiento interactivo y mejor comportamiento en tiempo real. Aunque el comportamiento en tiempo real está limitado a las capacidades del sistema operativo subyacente (Unix, Windows, etc.), aún supera a los entornos de flujo único de programa (single-threaded) tanto en facilidad de desarrollo como en rendimiento, como ocurre en C y C++.

Cualquiera que haya utilizado la tecnología de navegación concurrente, sabe lo frustrante que puede ser esperar por una gran imagen que se está trayendo. En Java, las imágenes se pueden ir trayendo en un hilo independiente, permitiendo que el usuario pueda acceder a la información en la página sin tener que esperar por el navegador.

Dinámico

Java se beneficia todo lo posible de la tecnología orientada a objetos. Java no intenta conectar todos los módulos que comprenden una aplicación hasta el tiempo de ejecución.

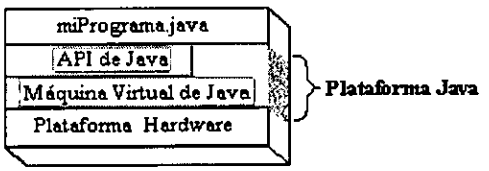

Figura 4. Capas que dan soporte a un programa Java.

2.4 Tendencias actuales para las aplicaciones en Internet.

En la actualidad, la mayoría de aplicaciones que se utilizan en entornos empresariales están construidas sobre una arquitectura cliente/servidor, en la cual una o varias computadoras (generalmente de una potencia considerable) son los servidores, que proporcionan servicios a un número mucho más grande de clientes conectados a través de la red. Los clientes suelen ser PCs de propósito general, menos potentes y más orientados al usuario final. A veces los servidores son intermediarios para los clientes y otros servidores más especializados (por ejemplo los grandes servidores de bases de datos corporativos basados en mainframes y/o sistemas Unix. En este caso se habla de *aplicaciones de varias capas*).

Con el auge de Internet la arquitectura cliente/servidor ha adquirido una mayor relevancia, ya que la misma es el principio básico de funcionamiento de la World Wide Web: un usuario que mediante un navegador (cliente) solicita un servicio (páginas HTML, etc.) a una computadora que hace las veces de servidor. En su concepción más tradicional, los servidores HTTP se limitaban a enviar una página HTML cuando el usuario la solicitaba directamente o al hacer clic sobre un enlace. La interactividad de este proceso era mínima, ya que el usuario podía pedir archivos, pero no enviar sus datos personales de modo que fueran almacenados en el servidor u obtuviera una respuesta personalizada. La Figura 5 representa gráficamente este concepto.

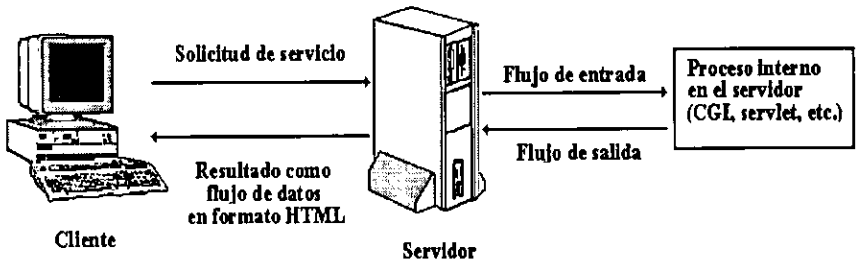

Figura 5. Arquitectura Cliente/Servidor

Esta concepción tradicional ha ido evolucionando en dos direcciones complementarias:

de enviar esta página HTML al cliente es a través de la salida estándar *(stduot* o *System.out*), que de ordinario suele estar asociada a la pantalla. La página HTML tiene que ser construida elemento a elemento, de acuerdo con las reglas de este lenguaje. No basta enviar el contenido: hay que enviar también todas y cada una de las etiquetas.

En principio, los programas CGI pueden estar escritos en cualquier lenguaje de programación, aunque en la práctica se han utilizado principalmente los lenguajes Perl y C/C++. Un claro ejemplo de un programa CGI sería el de un formulario en el que el usuario introdujera sus datos personales para registrarse en un sitio web. El programa CGI recibiría los datos del usuario, introduciéndolos en la base de datos correspondiente y devolviendo al usuario una página HTML donde se le informaría que sus datos han sido registrados.

Es importante resaltar que estos procesos tienen lugar en el servidor. Esto a su vez puede resultar un problema, ya que al tener múltiples clientes conectados al servidor, el programa CGI puede estar siendo llamado simultáneamente por varios clientes, con el riesgo de que el servidor se llegue a saturar. Téngase en cuenta que cada vez que se recibe un requerimiento se arranca una nueva copia del programa CGI.

Java ofrece una alternativa a los programas CGI: los servlets, que son a los servidores lo que los applets a los navegadores. Se podría definir un servlet como un programa escrito en Java que se ejecuta en el marco de un servicio de red, (un servidor HTTP, por ejemplo), y que recibe y responde a las peticiones de uno o más clientes.

La tecnología Servlet proporciona las mismas ventajas del lenguaje Java en cuanto a portabilidad ("*write once, run anywhere*") y seguridad, ya que un servlet es una clase de Java igual que cualquier otra, y por tanto tiene en ese sentido todas las características del lenguaje. Esto es algo de lo que carecen los programas CGI, ya que hay que compilarlos para el sistema operativo del servidor y no disponen en muchos casos de técnicas de comprobación dinámica de errores en tiempo de ejecución.

Otra de las principales ventajas de los servlets con respecto a los programas CGI es el rendimiento, y esto a pesar de que Java no es un lenguaje particularmente rápido. Mientras que con los otros es necesario cargar los programas CGI tantas veces como peticiones de servicio existan por parte de los clientes, los servlets, una vez que son llamados por primera vez, quedan activos en la memoria del servidor hasta que el programa que controla el servidor los desactiva. De esta manera se minimiza en gran medida el tiempo de respuesta.

Además, los servlets se benefician de la gran capacidad de Java para ejecutar métodos en computadoras remotas, para conectarse con bases de datos, para la seguridad en la información, etc. Se podría decir que las clases estándar de Java resuelven muchos problemas que con otros lenguajes el programador tiene que resolver.

Además de las características indicadas en el apartado anterior, estos tienen las siguientes:

1. Los servlets pueden llamar a otros servlets, e incluso a métodos concretos de otros.

Capítulo 3

Las herramientas de desarrollo.

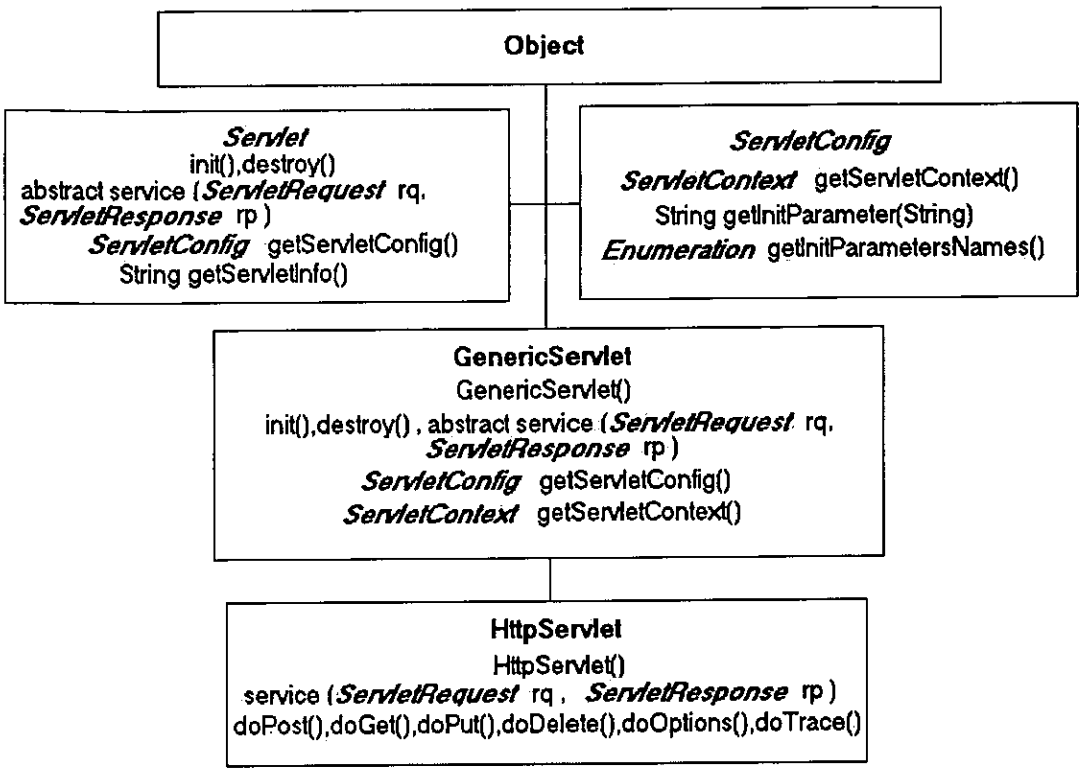

Figura 6. Jerarquía de clases para la clase HttpServlet.

Cualquier clase que derive de ***GenericServlet*** deberá definir el método ***service()***. Es muy interesante observar los dos argumentos que recibe este método, correspondientes a las interfaces ***ServletRequest*** y ***ServletResponse***. La primera de ellas referencia a un objeto que describe por completo la solicitud de servicio que se le envía al servlet. Si la solicitud de servicio viene de un formulario HTML, por medio de ese objeto se puede acceder a los nombres de los campos y a los valores introducidos por el usuario; puede también obtenerse cierta información sobre el cliente (computadora y browser). El segundo argumento es un objeto con una referencia de la interfaz ***ServletResponse***, que constituye el camino mediante el cual el método ***service()*** se conecta de nuevo con el cliente y le comunica el resultado de su solicitud. Además, dicho método deberá realizar cuantas operaciones sean necesarias para desempeñar su cometido: escribir y/o leer datos de un archivo, comunicarse con una base de datos, etc. El método ***service()*** es realmente el corazón del servlet.

En la práctica, salvo para desarrollos muy especializados, todos los servlets deberán construirse a partir de la clase ***HttpServlet***, sub-clase de ***GenericServlet***.

La clase ***HttpServlet*** ya no es ***abstract*** y dispone de una implantación o definición del método ***service()***. Dicha implantación detecta el tipo de servicio o método ***HTTP*** que ha sido solicitado desde el navegador y llama al método adecuado de esa misma clase ***(doPost()***, ***doGet()***, etc.). Cuando el programador crea una sub-clase de ***HttpServlet***, por lo general no tiene que redefinir el método ***service()***, sino uno de los métodos más

3.2.1 El ciclo de vida de un servlet: Clase GenericServlet.

La clase ***GenericServlet*** es una clase abstracta porque declara el método ***service()*** como ***abstract***.

Aunque los ***servlets*** desarrollados suelen derivar de la clase ***HttpServlet***, puede ser útil estudiar el ciclo de vida de un ***servlet*** en relación con los métodos de la clase ***GenericServlet***.

Además la clase ***HttpServlet*** hereda los métodos de ***GenericServlet*** y define el método ***service()***.

Los servlets se ejecutan en el servidor HTTP como parte integrante del propio proceso del servidor. Por este motivo el servidor HTTP es el responsable de la iniciación, llamada y destrucción de cada objeto de un ***servlet***, tal y como puede observarse en la Figura 7.

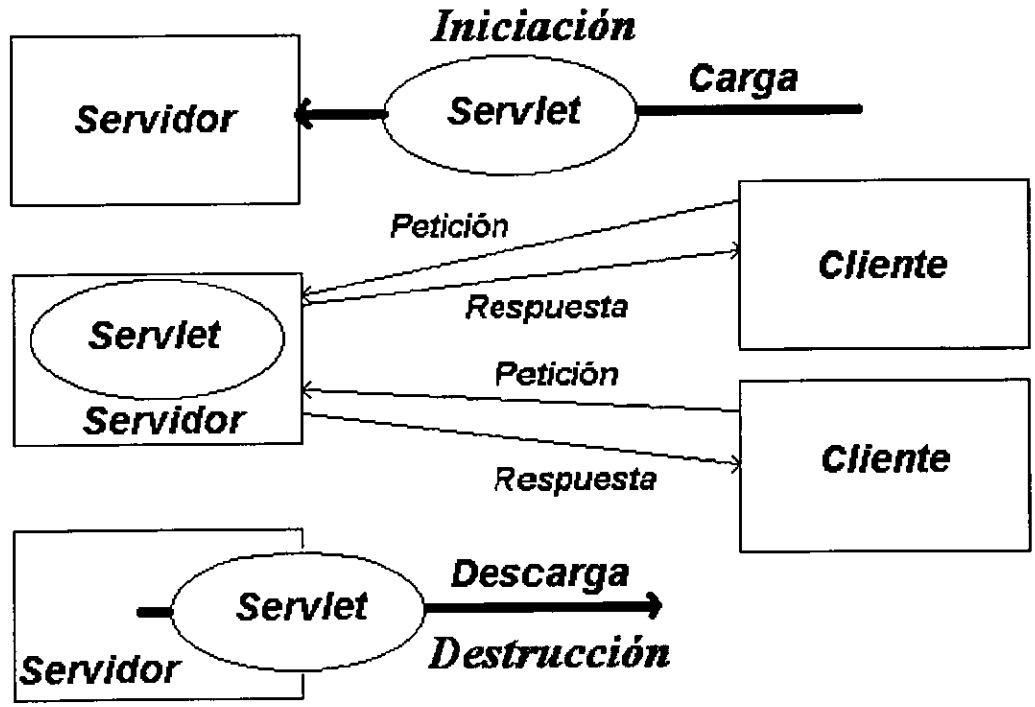

Figura 7. Ciclo de vida de un servlet.

Un servidor web se comunica con un servlet mediante los métodos de la interfaz ***javax.servlet.Servlet***. Esta interfaz está constituida básicamente por tres métodos principales:

Métodos de ServletRequest	Comentarios
public abstract int **getContentLength**()	Devuelve el tamaño de la petición del cliente o –1 si es desconocido.
public abstract String **getContentType**()	Devuelve el tipo de contenido **MIME** de la petición o **null** si éste es desconocido.
public abstract String **getProtocol**()	Devuelve el protocolo y la versión de la petición como un **String** en la forma <protocolo>/<versión principal>.<versión secundaria>, por ejemplo: HTTP/1.1 para servlets HTTP.
public abstract String **getScheme**()	Devuelve el nombre del esquema usado para hacer esta solicitud, por ejemplo: http, https, ftp,...
public abstract String **getServerName**()	Devuelve el nombre del host del servidor que recibió la petición.
public abstract int **getServerPort**()	Devuelve el número del puerto en el que fue recibida la petición.
public abstract String **getRemoteAddr**()	Devuelve la dirección IP de la computadora que realizó la petición.
public abstract String **getRemoteHost**()	Devuelve el nombre completo de la computadora que realizó la petición.
public abstract ServletInputStream **getInputStream**() throws IOException	Devuelve un **InputStream** para leer los datos binarios que vienen dentro del cuerpo de la petición.
public abstract String **getParameter**(String)	Devuelve un **String** que contiene el valor del parámetro especificado, o null si dicho parámetro no existe. Solo debe emplearse cuando se está seguro de que el parámetro tiene un único valor
public abstract String[] **getParameterValues**(String)	Devuelve los valores del parámetro especificado en forma de un arreglo de Strings, o null si el parámetro no existe. Útil cuando un parámetro tiene más de un valor.
public abstract Enumeration **getParameterNames**()	Devuelve una enumeración de objetos **String** que contiene los nombres de los parámetros de esta solicitud. Si la solicitud no tiene parámetros el método devuelve una enumeración vacía.
public abstract BufferedReader **getReader**() throws IOException	Devuelve un **BufferedReader** que permite leer el texto contenido en el cuerpo de la petición.
public abstract String **getCharacterEncoding**()	Devuelve el tipo de codificación de los caracteres empleados en la petición.

Tabla 1. Métodos de la interfaz ***ServletRequest***.

Esto hace que haya que ser especialmente cuidadoso con los hilos, para evitar por ejemplo que diversos objetos de un servlet se encuentren escribiendo simultáneamente en el mismo campo de una base de datos.

A pesar de la importancia del método ***service()***, en general no es aconsejable su definición (no queda más remedio que hacerlo si la clase del servlet deriva de ***GenericServlet***, pero lo lógico es que el programador derive las clases de sus servlets de ***HttpServlet***). El motivo es simple; la clase ***HttpServlet*** define ***service()*** de una forma más que adecuada, llamando a otros métodos ***(doPost())***, ***doGet()***, etc.) que son los que tiene que redefinir el programador.

El método destroy() en la clase GenericServlet: forma de terminar ordenadamente.

Una buena implantación de este método debe permitir que el servlet concluya sus tareas de forma ordenada.

De esta forma es posible liberar recursos (archivos abiertos, conexiones con bases de datos, etc.) de una forma limpia y segura. Cuando esto no es necesario o importante, no hará falta redefinir el método ***destroy()***.

Puede suceder que al llamar al método ***destroy()*** haya peticiones de servicio que estén todavía siendo ejecutadas por el método ***service()***, lo que podría provocar un fallo general del sistema.

Por este motivo es conveniente escribir el método ***destroy()*** de forma que se retrase la liberación de recursos hasta que no hayan concluido todas las llamadas al método ***service()***.

A continuación se presenta una forma de lograr una correcta descarga del servlet:

En primer lugar es preciso saber si existe alguna llamada al método ***service()*** pendiente de ejecución, para lo cual se debe llevar un contador con las llamadas activas a dicho método.

Aunque en general es poco recomendable redefinir ***service()*** (en caso de tratarse de un servlet que derive de ***HttpServlet***) en este caso sí resulta conveniente su redefinición, para poder saber cuándo ha sido llamado. El método redefinido deberá llamar al método ***service()*** de su super-clase para mantener íntegra la funcionalidad del servlet.

Los métodos de actualización del contador deben estar ***sincronizados***, para evitar que dicho valor sea accedido simultáneamente por dos o más hilos, lo que podría hacer que su valor fuera erróneo.

Además, no basta con que el servlet espere a que todos los métodos ***service()*** hayan acabado. Es preciso indicarle a dicho método que el servidor se dispone a apagarse. De otra forma, el servlet podría quedar esperando indefinidamente a que los métodos ***service()*** acabaran. Esto se consigue utilizando una variable ***boolean*** que establezca esta condición.

3.2.2 El contexto del servlet (Servlet context).

Un servlet vive y muere dentro de los límites del proceso del servidor. Por este motivo, puede ser interesante en un determinado momento obtener información acerca del entorno en el que se está ejecutando el servlet.

Esta información incluye la disponible en el momento de iniciación del servlet, la referente al propio servidor o la información contextual específica que puede contener cada petición de servicio.

Información durante la iniciación del servlet.

Esta información es suministrada al servlet mediante el argumento ***ServletConfig*** del método ***init()***.

Cada servidor HTTP tiene su propia forma de pasar información al servlet. En cualquier caso, para acceder a dicha información habría que emplear un código similar al siguiente:

```
String valorParametro;
public void init(ServletConfig config) {
valorParametro = config.getInitParameter(nombreParametro);
}
```

Como puede observarse, se ha empleado el método ***getInitParameter()*** de la interfaz ***ServletConfig*** (implantada por ***GenericServlet***) para obtener el valor del parámetro. Asimismo, puede obtenerse una ***enumeración*** de todos los nombres de parámetros mediante el método ***getInitParameterNames()*** de la misma interfaz.

Información contextual acerca del servidor.

La información acerca del servidor está disponible en todo momento a través de un objeto de la interfaz ***ServletContext***. Un servlet puede obtener dicho objeto mediante el método ***getServletContext()*** aplicable a un objeto ***ServletConfig***.

La interfaz ***ServletContext*** define los métodos descritos en la Tabla 3.

Métodos de ServletContext	**Comentarios**
public abstract Object **getAttribute(String)**	Devuelve información acerca de determinados atributos del tipo clave/valor del servidor. Es propio de cada servidor.
public abstract Enumeration **getAttributeNames()**	Devuelve una enumeración con los nombres de atributos disponibles en el servidor. Solo disponible en la versión 2.1.
public abstract String **getMimeType(String)**	Devuelve el tipo **MIME** de un determinado archivo.

- El ***Servlet API*** incluye dos clases de excepciones[14]:

1.- La excepción ***javax.servlet.ServletException*** puede ser empleada cuando ocurre un fallo general en el servlet. Esto hace saber al servidor que hay un problema.

2.- La excepción ***javax.servlet.UnavailableException*** indica que un servlet no se encuentra disponible. Los servlets pueden notificar esta excepción en cualquier momento.

Existen dos tipos de indisponibilidades:

a) ***Permanente*:** El servlet no podrá seguir funcionando hasta que el administrador del servidor haga algo. En este estado, el servlet debería escribir en el archivo de *log* una descripción del problema, y posibles soluciones.

b) ***Temporal***: El servlet se ha encontrado con un problema que es potencialmente temporal, como pueda ser un disco lleno, un servidor que ha fallado, etc. El problema puede arreglarse con el tiempo o puede requerir la intervención del administrador.

3.2.4 Clase HttpServlet: Soporte específico para el protocolo HTTP.

Los servlets que utilizan el protocolo ***HTTP*** son los más comunes. Por este motivo, ***Sun*** ha incluido un paquete específico para estos servlets en su ***JSDK***: ***javax.servlet.http***.

Antes de estudiar dicho paquete en profundidad, se va a hacer una pequeña referencia al protocolo HTTP.

HTTP son las siglas de ***HyperText Transfer Protocol***, que es un protocolo mediante el cual los navegadores y los servidores puedan comunicarse entre sí, haciendo uso de una serie de métodos:

GET, HEAD, POST, PUT, DELETE, TRACE, CONNECT y OPTIONS.

Para la mayoría de las aplicaciones, bastará con conocer los tres primeros.

[14] **Excepción.** Es un evento que ocurre durante la ejecución del programa que interrumpe el flujo normal de las sentencias.

Sin embargo, hay que tener cuidado con algunos caracteres al incluirlos en un URL, en particular con aquellos caracteres no pertenecientes al código ASCII y con aquellos que tienen un significado concreto para el protocolo HTTP. Ejemplificando citaremos algunos caracteres especiales y su secuencia o código equivalente:

```
" (%22), # (%23), % (%25), & (%26), + (%2B), , (%2C), / (%2F),
: (%3A), < (%3C), = (%3D), > (%3E), ? (%3F) y @ (%40).
```

Adicionalmente, Java proporciona la posibilidad de codificar un URL de forma que cumpla con las anteriores restricciones. Para ello, se puede utilizar la clase ***URLEncoder***, que se encuentra incluida en el paquete ***java.net***, que es un paquete estándar de Java. Dicha clase tiene un único método, ***String encode(String)***, que se encarga de codificar el ***String*** que recibe como argumento, devolviendo otro ***String*** con el URL debidamente codificado. Así, considérese el siguiente ejemplo:

```
import java.net.*;
public class Codificar {
public static void main(String argv[]) {
String URLcodificada=URLEncoder.encode("/servlet/MiServlet?nombre=Antonio"+
"&Apellido=López de Romera");
System.out.println(URLcodificada);
}
}
```

que cuando es ejecutado tiene como resultado la siguiente secuencia de caracteres:

```
%2Fservlet%2FMiServlet%3Fnombre%3DAntonio%26Apellido%3DL%A2pez+de+Romera
```

Obsérvese además que, cuando sea necesario escribir una comilla dentro del String de ***out.println(String)***, hay que precederla por el carácter ***escape*** (\). Así, la sentencia:

```
out.println("<A HREF="http://www.yahoo.com">Yahoo</A>"); // INCORRECTA
```

es incorrecta y produce errores de compilación. Deberá ser sustituida por:

```
out.println("<A HREF=\"http://www.yahoo.com\'>Yahoo</A>");
```

Las peticiones HTTP GET tienen una limitación importante (recuérdese que transmiten la información a través de las variables de entorno del sistema operativo) debido a la cantidad de caracteres que se pueden aceptar en el URL. Si se envían los datos de un formulario muy extenso mediante HTTP GET pueden producirse errores por este motivo, por lo que habría que utilizar el método HTTP POST.

Se suele decir que el método ***GET*** es ***seguro*** e ***idempotente***:

- ***Seguro***, porque no tiene ningún efecto secundario del cual pueda considerarse al usuario responsable (una llamada al método GET no debe ser capaz, en teoría, de alterar una base de datos). El método GET debería servir únicamente para obtener información.

Nótese la existencia de una línea en blanco entre el encabezado *(header)* y el comienzo de la información extendida. Esta línea en blanco indica el final del encabezado.

A diferencia de los anteriores métodos, POST no es ni seguro ni idempotente, y por tanto es conveniente su utilización en aquellas aplicaciones que requieran operaciones más complejas que las de sólo-lectura, como por ejemplo modificar bases de datos.

3.2.5 Clases de soporte HTTP.

Una vez que se han presentado los conceptos básicos del protocolo HTTP, resulta más sencillo entender las funciones del paquete ***javax.servlet.http***, que facilitan la creación de servlets que empleen dicho protocolo.

La clase abstracta ***javax.servlet.http.HttpServlet*** implanta la interfaz ***java.servlet.Servlet*** e incluye un número importante de funciones adicionales. La forma más sencilla de escribir un ***servlet HTTP*** es heredando de ***HttpServlet*** como puede observarse en la Figura 8.

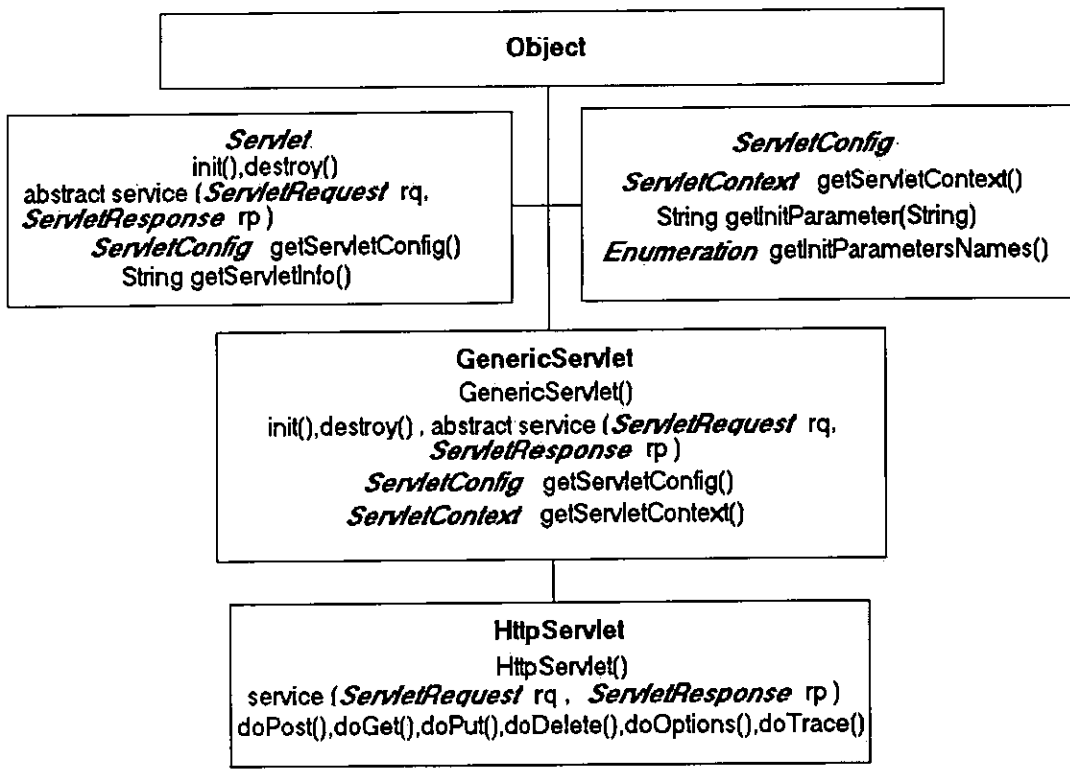

Figura 8. Jerarquía de clases en servlets.

La clase ***HttpServlet*** es también una clase abstracta (***abstract***), de modo que es necesario definir una clase que derive de ella y redefinir en la clase derivada al menos uno de sus métodos, tales como ***doGet()***, ***doPost()***, etc.

algunos de los métodos de ***ServletRequest*** y ***ServletResponse*** se encuentran en la Tabla 1 y Tabla 2 respectivamente.

En resumen el método ***doGet()*** debería :

1. Leer los datos de la solicitud, tales como los nombres de los parámetros y sus valores

2. Establecer el encabezado *(header)* de la respuesta (longitud, tipo y codificación).

3. Escribir la respuesta en formato HTML para enviarla al cliente.

Teniendo en cuenta que la implantación de este método debe ser segura e idempotente.

El método ***doPost()*** por su parte, debería realizar las siguientes funciones:

1. Obtener input stream del cliente y leer los parámetros de la solicitud.

2. Realizar aquello para lo que está diseñado (actualización de bases de datos, etc.).

3. Informar al cliente de la finalización de dicha tarea o de posibles imprevistos. Para ello hay que establecer primero el tipo de la respuesta, obtener luego un PrintWriter y enviar a través suyo el mensaje HTML.

3.2.6 Formas de seguir la trayectoria de los usuarios (clientes).

Los servlets permiten seguir la trayectoria de un cliente, es decir, obtener y mantener una determinada información acerca del cliente. De esta forma se puede tener identificado a un cliente (usuario que está utilizando un navegador) durante un determinado tiempo.

Esto es muy importante si se quiere disponer de aplicaciones que impliquen la ejecución de varios servlets o la ejecución repetida de un mismo servlet.

Un claro ejemplo de aplicación de esta técnica es el de los comercios vía Internet que permiten llevar un carrito de la compra en el que se van guardando aquellos productos solicitados por el cliente.

El cliente puede ir navegando por las distintas secciones del comercio virtual, es decir realizando distintas conexiones HTTP y ejecutando diversos servlets, y a pesar de ello no se pierde la información contenida en el carrito de la compra y se sabe en todo momento que es un mismo cliente quien está haciendo esas conexiones diferentes.
El mantener información sobre un cliente a lo largo de un proceso que implica múltiples conexiones se puede realizar de tres formas distintas:

La forma de implantar todo esto es relativamente simple gracias a la clase ***Cookie*** incluida en el ***Servlet API***. Para enviar una cookie es preciso:

1. Crear un objeto ***Cookie***
2. Establecer sus atributos
3. Enviar la ***Cookie***

Por otra parte, para obtener información de una cookie, es necesario:

1. Recoger todas las cookies de la petición del cliente
2. Encontrar la cookie precisa
3. Obtener el valor almacenado en la misma

3.2.6.2 Seguimiento de sesiones (Session tracking).

Una ***sesión*** es una conexión continuada de un mismo navegador a un servidor durante un lapso prefijado de tiempo. Este tiempo depende habitualmente del servidor, aunque a partir de la versión *2.1* del ***Servlet API*** puede establecerse mediante el método ***setMaxInactiveInterval(int)*** de la interfaz ***HttpSession***. Esta interfaz es la que proporciona los métodos necesarios para mantener sesiones.

Al igual que las cookies, las sesiones son compartidas por todos los servlets de un mismo servidor.

Por ello, si el navegador no acepta cookies, habrá que emplear las sesiones en conjunción con la reescritura de URLs.

La forma de obtener una sesión es mediante el método ***getSession(boolean)*** de un objeto ***HttpServletRequest***. Si este boolean es ***true***, se crea una sesión nueva si es necesario mientras que si es ***false***, el método devolverá la sesión actual. Por ejemplo:

```
...
HttpSession miSesion = req.getSession(true);
...
```

crea una nueva sesión con el nombre ***miSesion***.

Una vez que se tiene un objeto ***HttpSession***, es posible mantener una colección de pares ***nombre_de_dato/valor_de_dato***, de forma que pueda almacenarse todo tipo de información sobre la sesión. Este valor puede ser cualquier objeto de la clase ***Object*** que se desee. La forma de añadir valores a la sesión es mediante el método ***putValue(String ,Object)*** de la clase ***HttpSession*** y la de obtenerlos es mediante el método ***getValue(String , Object)*** del mismo objeto. Esto puede verse en el siguiente ejemplo:

De todos los anteriores métodos conviene comentar dos en especial: *invalidate()* y *isNew()*.

El método ***invalidate()*** invalida la sesión en curso. Tal y como se ha mencionado con anterioridad, una sesión puede ser invalidada por el propio servidor si en el transcurso de un intervalo prefijado de tiempo no ha recibido peticiones de un cliente. ***Invalidar*** quiere decir eliminar el objeto ***HttpSession*** y los valores asociados con él del sistema.

El método ***isNew()*** sirve para conocer si una sesión es *"nueva"*. El servidor considera que una sesión es nueva hasta que el cliente se una a la sesión. Hasta ese momento ***isNew()*** devuelve ***true***.

Un valor de retorno ***true*** puede darse en las siguientes circunstancias:

1. El cliente todavía no sabe acerca de la sesión.
2. La sesión todavía no ha comenzado.
3. El cliente no quiere unirse a la sesión. Ocurre cuando el navegador tiene la aceptación de cookies desactivada.

3.2.6.3 Reescritura de urls.

A pesar de que la mayoría de los navegadores más extendidos soportan las cookies en la actualidad, para poder emplear sesiones con clientes que o bien no soportan cookies o bien las rechazan, debe utilizarse la reescritura de URLs.

No todos los servidores soportan la reescritura de URLs (por ejemplo el ***servletrunner*** que acompaña el ***JSDK***).

Para emplear esta técnica se incluye el código que identifica la sesión (***sessionId***) en el URL de la petición. Los métodos que se encargan de rescribir el URL si fuera necesario son:

HttpServletResponse.encodeUrl() y
HttpServletResponse.encodeRedirectUrl()

(sustituidas en el *API 2.1* por ***encodeURL()*** y ***encodeRedirectURL()*** respectivamente).

El primero de ellos lee un ***String*** que representa un URL y si fuera necesario la rescribe añadiendo el identificador de la sesión, dejándolo inalterado en caso contrario. El segundo realiza lo mismo sólo que con URLs de redirección, es decir, permite reenviar la petición del cliente a otro URL .

Véase el siguiente ejemplo:

Capítulo

Creación de un servlet.

El código correspondiente a la *página* HTML que contiene este formulario es el siguiente:

(archivo Ejemplo.html):

```
<HTML>
<HEAD>
<TITLE>Envíe su opinión</TITLE>
</HEAD>
<BODY>
<H2>Servlet ejemplo</H2>
<FORM ACTION="http://localhost:8080/servlet/Ejemplo" METHOD="POST">
Nombre: <INPUT TYPE="TEXT" NAME="nombre" SIZE=15><BR>
Apellidos: <INPUT TYPE="TEXT" NAME="apellidos" SIZE=30><P>
Opinión que le ha merecido este sitio web<BR>
<INPUT TYPE="RADIO" CHECKED NAME="opinion" VALUE="Buena">Buena<BR>
<INPUT TYPE="RADIO" NAME="opinion" VALUE="Regular">Regular<BR>
<INPUT TYPE="RADIO" NAME="opinion" VALUE="Mala">Mala<P>
Comentarios <BR>
<TEXTAREA NAME="comentarios" ROWS=6 COLS=40>
</TEXTAREA><P>
<INPUT TYPE="SUBMIT" NAME="botonEnviar" VALUE="Enviar">
<INPUT TYPE="RESET" NAME="botonLimpiar" VALUE="Limpiar">
</FORM>
</BODY>
</HTML>
```

En el código anterior hay algunas cosas que merecen ser comentadas.

En primer lugar es necesario asignar un identificador único (es decir, un valor de la propiedad *NAME*) a cada uno de los *campos* del formulario, ya que la información que reciba el *servlet* estará organizada en forma de *pares de valores*, donde uno de los elementos será una cadena que contendrá el *nombre del campo*.

Así, por ejemplo, si se introdujera como nombre del visitante *"Jorge"*, el *servlet* recibiría del browser el par *nombre=Jorge*, que permitirá acceder de una forma sencilla al nombre introducido mediante el método *getParameter()*, tal y como se explicará posteriormente al analizar el *servlet* del ejemplo introductorio.

Por este motivo es importante no utilizar nombres duplicados en los elementos de los formularios.

Por otra parte puede observarse que en el *tag* <FORM> se han utilizado dos propiedades, *ACTION* y *METHOD*.

El método (*METHOD*) utilizado para la transmisión de datos es el método HTTP POST.

El código fuente de la clase Ejemplo es el siguiente:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class Ejemplo extends HttpServlet {

// Declaración de variables miembro correspondientes a
// los campos del formulario
private String nombre=null;
private String apellidos=null;
private String opinion=null;
private String comentarios=null;


// Método llamado mediante un HTTP POST. Este método se llama
// automáticamente al ejecutar un formulario HTML
public void doPost (HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {

// Adquisición de los valores del formulario a través del objeto req
nombre=req.getParameter("nombre");
apellidos=req.getParameter("apellidos");
opinion=req.getParameter("opinion");
comentarios=req.getParameter("comentarios");

// Devolver al usuario una página HTML con los valores adquiridos
devolverPaginaHTML(resp);

} // fin del método doPost()

public void devolverPaginaHTML(HttpServletResponse resp) {

// En primer lugar se establece el tipo de contenido MIME de la respuesta
resp.setContentType("text/html");

// Se obtiene un PrintWriter donde escribir (sólo para mandar texto)
PrintWriter out = null;

//Tratamiento de la excepción que puede generar el método getWriter()
try {
out=resp.getWriter();
} catch (IOException io) {
System.out.println("Se ha producido una excepcion");
}

// Se genera el contenido de la página HTML
```

2. También desde una página *HTML* puede llamarse a un *servlet*. Para ello habrá de emplearse el *tag* adecuado. En el caso de que se trate simplemente de un enlace:

Ejemplo de un servlet

Si se trata de un formulario, habrá que indicar el *URL* del *servlet* en la propiedad *ACTION* de la *tag* <FORM> y especificar el método *HTTP* (*GET*, *POST*, etc.) en la propiedad *METHOD* también en la misma *tag*.

3. Al tratarse de clases *Java* como las demás, pueden crearse objetos de dicha clase (ser *instanciadas*), aunque siempre con el debido cuidado de llamar a aquellos métodos de la clase instanciada que sean necesarios.

   En ocasiones es muy útil escribir *servlets* que realicen una determinada función y que sólo puedan ser llamados por otros *servlets*. En dicho caso, será preciso redefinir su método *service()* de la siguiente forma:

```
public void service(ServletRequest req, ServletResponse resp)
throws ServletException, IOException
{
   throw new UnavailableException(this,
  "Este servlet no acepta llamadas de clientes, solamente de otros servlets");
} // fin del método service()
```

En este caso el programador debe llamar explícitamente desde otro *servlet* los métodos del servlet que quiere ejecutar. Recuérdese que de ordinario los métodos ***service()***, ***doPost()***, ***doGet()***, etc. son llamados automáticamente cuando el servidor HTTP recibe una solicitud de servicio de un formulario introducido en una página HTML.

Con el fin de ofrecer una mayor simplicidad en esta primera aproximación a los servlets, se ha evitado tratar de conseguir un código más sólido, que debería realizar las comprobaciones pertinentes (verificar que los *String* no son *null* después de leer el parámetro, excepciones que se pudieran dar, etc.) e informar al usuario acerca de posibles errores si fuera necesario.

En cualquier caso, puede observarse que el aspecto del código del servlet es muy similar al de cualquier otra clase de Java.

y después escribirlo en el ***stream*** o flujo de salida. Sin embargo, uno de los parámetros a tener en cuenta en los servlets es el tiempo de respuesta, que tiene que ser el mínimo posible.

En este sentido, la creación de un *String* mediante concatenación es bastante costosa, pues cada vez que se concatenan dos *Strings* mediante el signo + se están convirtiendo a ***StringBuffers*** y a su vez creando un nuevo *String*, lo que utilizado profusamente requiere más recursos que lo que se ha hecho en el ejemplo, donde se ha escrito directamente mediante el método ***println()***.

El esquema mencionado en este ejemplo se repite en la mayoría de los servlets y es el fundamento de esta tecnología.

Capítulo 5

Creación del metabuscador Amoxcalli.

Amoxcalli no hace un manejo detallado e inteligente sobre los resultados a proporcionar ya que el objetivo de este trabajo es comprender el desarrollo de aplicaciones sobre Internet, en particular la de un metabuscador, así la eficiencia para esta aplicación solo se mide por la relevancia de las páginas que son presentadas y la tecnología usada para la creación de la aplicación.

Los primeros resultados que proporciona cada uno de los buscadores base de la metaconsulta, son los resultados más relevantes de cada uno de ellos, de esta forma Amoxcalli desplegara una lista de los primeros diez resultados que proporciona cada metabuscador. En cuanto a la tecnología utilizada para el desarrollo de esta aplicación quedo justificada en el capitulo 2 al mencionar las ventajas del lenguaje de programación Java y la tecnología Servlet sobre tecnologías similares.

5.2 Diseño.

El desarrollo de esta aplicación estará basada en la arquitectura cliente/servidor, la figura 10 ilustra a grandes rasgos el comportamiento entre el cliente y el servidor para la solicitud del servicio de metaconsulta.

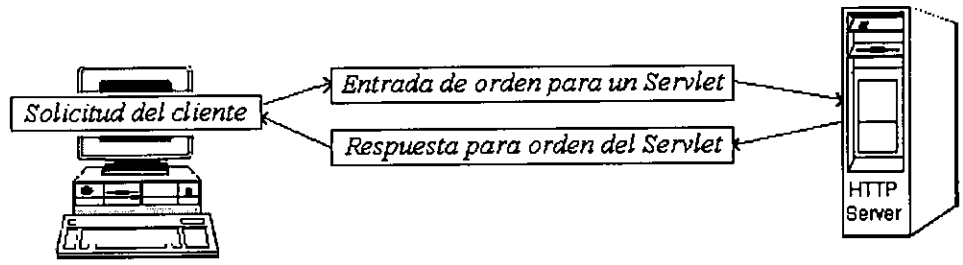

Figura 10. Comportamiento entre el cliente y servidor.

Del lado del usuario se tiene:

- Invocación de la solicitud del servicio en un navegador mediante un URL.
- Solicitud de servicio, la cual es una página HTML con los elementos necesarios para especificar la solicitud del usuario y para la invocación del servlet que efectuará el servicio de metaconsulta.

Del lado del servidor se cuenta con un contenedor de servlets el cual llamará al servlet que efectuará la metaconsulta, el comportamiento de esto se ejemplifica en la figura 11.

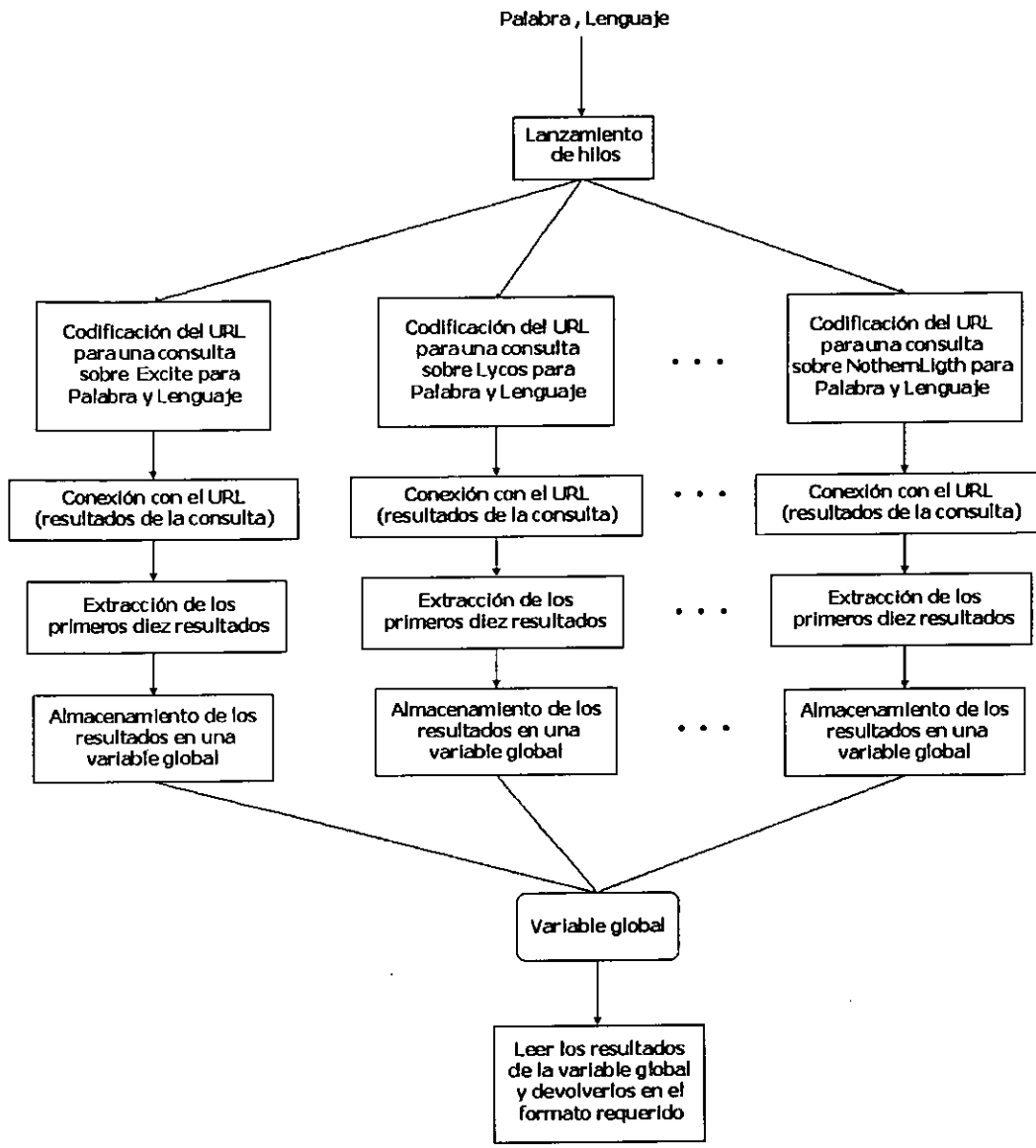

Figura 13. Funcionamiento de **MiServlet.**

A continuación se hará una descripción de las clases necesarias para implantar el servlet que efectúe la metaconsulta.

Parámetros

url – Especifica la localidad de la aplicación.

prop – Es una lista de parámetros con valores con los cuales será construido el URL

---

public class **ConEx**
extends

Esta clase proporciona dos métodos, uno para la creación de un URL que define una consulta en Excite sobre los atributos Palabra y Lenguaje y otro método para la recuperación de los primeros diez resultados de la consulta definida por este URL.

Resumen de Atributos	
public String	**Palabra** Palabra a consultar
public String	**Lenguaje** Lenguaje en el que se desea que se presenten los recursos

Resumen **de Constructores**
public ConEx(String **palabra**, String **lenguaje**) Inicia los atributos **Palabra** y **Lenguaje.**

Resumen de Métodos	
public URL	**daURL**() Devuelve el URL que define una consulta con Excite sobre los atributos **Palabra** y **Lenguaje.**
public Vector	**Busca**() Devuelve un vector con los primeros diez resultados de una consulta a Excite sobre **Palabra** y **Lenguaje.**

Resumen de Atributos	
public static Vector	**Resultados** Vector en el que son almacenados los resultados de cada una de las consultas para **Palabra** y **Lenguaje** sobre los buscadores base de la metaconsulta.
public static String	**Palabra** Palabra a consultar
public static String	**Lenguaje** Lenguaje en el que se desea que sean desplegados los recursos
public int	**Consulta** Entero que identifica sobre qué buscador se hará la consulta, 1 identifica el buscador EuroSeek, 2 identifica el buscador Lycos y así cada uno de los buscadores base tiene asociado un número entero.

Resumen de Constructores
Crea_Threads(String **pal**,String **len**) Inicia **Palabra** y **Lenguaje** con **pal** y **len** respectivamente.
Crea_Threads(ConSeek c) Inicia el valor de Consulta con 1
Crea_Threads(ConLyc c) Inicia el valor de Consulta con 2
Crea_Threads(ConGo c) Inicia el valor de Consulta con 3
Crea_Threads(ConVista c) Inicia el valor de Consulta con 4
Crea_Threads(ConNor c) Inica el valor de Consulta con 5
Crea_Threads(ConEx c) Inicia el valor de Consulta con 6

Resumen de Constructores
Por defecto

Resumen de Métodos	
public void	**doGet(HttpServletRequest req, HttpServletResponse res)** Método que efectúa la metaconsulta sobre la palabra y lenguaje que el usuario proporcione, devolviendo los resultados de esta consulta en una forma que contiene los campos de palabra y lenguaje en caso de que se desee efectuar otra consulta.
private void	**getParameters(HttpServletRequest req)** Mediante este método son capturados los valores que son introducidos en los campos palabra y lenguaje al efectuar la solicitud de metaconsulta.

5.3 Creación de un servlet que efectúa una consulta.

Una consulta consiste en solicitar a través de la interfaz de un buscador los recursos (documentos, imágenes, video, etc) que hagan referencia a la palabra que se solicitó.

En la consulta se pueden especificar ciertos parámetros como: lenguaje en el que estén escritas las páginas desplegadas, el número de recursos (enlaces) a desplegar, el país donde se quiere que se busquen los recursos, el tipo de recurso que se desea desplegar (html, video, imagen, pdf.), etc. Estos parámetros varían entre buscador, unos pueden tener lo que otros no.

Finalmente toda consulta en cualquier buscador que lleve a cabo la invocación del servicio mediante el método GET tendrá el siguiente formato:

http://servidor.dominio/ruta-de-acceso-del-buscador?
key1=value1&key2=value2&key3=value3&...keyn=valuen.

Donde servidor.dominio/ruta-de-acceso-del-buscador especifica la ubicación del programa que efectúa una búsqueda de recursos con base en los parámetros proporcionados.

El código del servlet consistirá de una clase que herede de HttpServlet, la cual estructuralmente tiene la siguiente forma:

- Capturar las entradas que el usuario proporciona en la forma.
- Con los parámetros que el usuario proporcionó crear el URL de consulta.

La clase **URLQuery** se encargará de crear este URL, lo que llevará a cabo una vez que se le han proporcionado los parámetros: palabra a buscar y lenguaje, el buscador sobre el que se desea efectuar la consulta es especificado dentro del código de la misma clase. Así si se proporciona la palabra "galois" y lenguaje "inglés" esta clase nos debe de retornar el URL que representa la consulta sobre un buscador; si este buscador fuese Excite el resultado sería el siguiente:

http://search.excite.com/search.gw?search=galois&tsug=-1&csug=10&lang=en

- Establecer la conexión con el URL que representa una consulta a un buscador sobre una palabra y lenguaje determinados para poder acceder al contenido de este recurso; que a fin de cuentas son los resultados de la misma, para así tratarlos y desplegarlos de la forma deseada.

  La clase **ConEx** define el método **Busca()**, el cual se encarga de establecer la conexión con el URL de consulta y recuperar los resultados una vez establecida la conexión.

- Desplegar estos resultados en el formato que se desee.

La figura 15 muestra los resultados de un servlet que efectúa la consulta sobre el buscador Excite, la palabra Galois y el lenguaje Alemán.

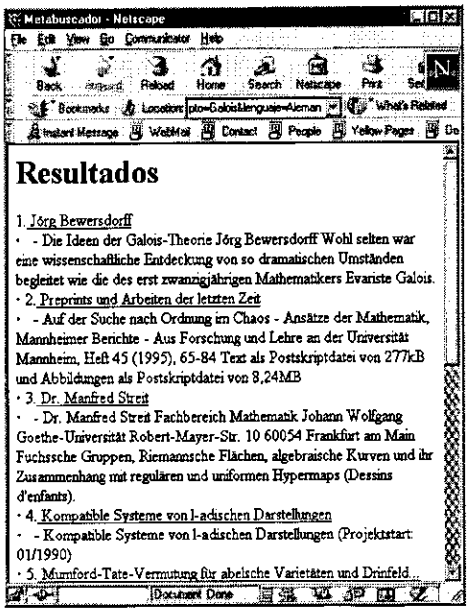

Figura 15. Resultados de la consulta sobre Excite.

Una vez finalizado el servlet de la metaconsulta, en el cual se hicieron las invocaciones de búsqueda de manera sucesiva sobre los diferentes buscadores, se encontró el inconveniente de que el tiempo que se llevaba el servlet para mostrar los resultados era demasiado, y por tal razón se optó por agregar una clase que creara hilos de ejecución; cada hilo efectuaría una de las consultas que se tienen sobre los diferentes buscadores y almacenaría los resultados de la consulta en una variable global y pública. (para una explicación mas detallada de cómo se llevo a cabo lo antes mencionado, se puede consultar el Apéndice B).

Una vez resuelto el problema sobre el tiempo para mostrar los resultados, solo restaba desplegar los resultados de tal forma que se mostraran los primeros diez enlaces y debajo de ellos un listado de números donde al hacer clic en el número *i* se mostraría el *i*-ésimo bloque de diez resultados, como cualquier buscador convencional; también se deseaba que una vez que se mostraran los resultados de una consulta particular el usuario tuviese la oportunidad de hacer ahí mismo otra consulta, es decir que se le presentase la forma de solicitud de consulta al mismo tiempo que se le mostraban los resultados. Para desplegar los resultados como se ha explicado se siguieron los pasos:

1.- Capturar los parámetros que el usuario proporciona en la forma de consulta; hay un parámetro implícito que nos dice si el usuario está invocando una nueva consulta; este parámetro es una variable que estará vacía si es la primera vez que se hace una consulta o si dentro de una consulta ya hecha se desea hacer otra; en otro caso el parámetro contendrá un número, lo que indica que el usuario hizo clic en alguno de los números listados abajo de los diez primeros resultados una vez que ya se ha hecho una consulta.

2.- Verificar el parámetro que nos indica si el usuario está haciendo una nueva consulta o si desea navegar sobre un bloque específico de diez resultados.

Si el usuario desea una nueva consulta se invocara la ejecución de la metaconsulta, en otro caso el usuario ya ha hecho la consulta y desea que se le despliegue el i-ésimo bloque de diez resultados; para este caso se accederá a la variable global donde se tienen los resultados de la metaconsulta (existen porque la consulta ya ha sido hecha), y se despliegan los resultados correspondientes al valor del parámetro. Las figuras 16 y 17 muestran la forma y los resultados de la metaconsulta.

Conclusiones.

La construcción de Amoxcalli nos condujo a un claro entendimiento del entorno de trabajo de un metabuscador; en general de cualquier aplicación que haga algún tratamiento para la recuperación y manejo de la información que se encuentra en la Web.

Dentro de este entorno quedaron claramente comprendidos los conceptos y tecnologías así como las relaciones entre ellos. Las ventajas, desventajas y nuevas tendencias dentro de este ámbito se descubrieron a lo largo de la investigación, diseño y construcción de Amoxcalli. En este trabajo se alcanzaron los objetivos planteados y se adquirió una visión más crítica y profunda con la cual se detectaron algunas de las deficiencias actuales de los sistemas en cuestión.

El conocimiento adquirido permitió proponer algunas ideas como trabajos posteriores a éste, las cuales pretenden mejorar la eficiencia y eficacia en la recuperación y tratamiento de información, partiendo de herramientas reutilizables las cuales conforman esta aplicación. Tales ideas surgieron de la necesidad de resolver los problemas encontrados a lo largo del desarrollo de Amoxcalli.

Los problemas detectados en el desarrollo de este trabajo y que no han sido resueltos por ninguno de los buscadores y metabuscadores actuales son:

Ambigüedad.

Los resultados presentan ambigüedad; por ejemplo si tecleamos en el campo de palabra "*mango*", los recursos que despliega la aplicación muestran información referente a la fruta mango, información referente a los mangos de las pistolas, información de un recurso que tiene como encabezado mango o quizás información sobre una compañía llamada mango que hace bolsas de plástico, lo que hace poco eficiente la búsqueda del usuario sobre la información que desea.

Resultados con poco valor informativo.

Se carece de resultados con valor informativo ya que los recursos que se muestran son en su mayoría recursos con muchos enlaces y poca información textual sobre la palabra que se tecleó, el recurso podría ser una imagen cuyo título es la palabra a buscar, y la imagen no tiene nada que ver con información referente a la palabra, toda esta información con poco valor textual le hace perder tiempo al usuario en encontrar información deseada.

Duplicidad.

La duplicidad se presenta cuando dos URLs distintos especifican el mismo recurso, lo cual disminuye la eficacia de los buscadores y metabuscadores.

Bibliografía.

[1] Eckel,B. **Thinking in Java.** Prentice-Hall 1998.
[2] Gosling,J., Steele,G. **The Java Language Specification.** Addison-Wesley 1996.
[3] Patzer,A., Ayers, D., Bergsten H. **Professional Java Server Programming.** Wrox Press 1999.
[4] Hunter, J.,Crawford. ,W. **Java Servlet Programming.** O´Reilly & Associates. 1998.
[5] Kernigan, B. **El lenguaje de programación C.**
[6] Held,G. **Internetworking LANs and WANs.** Wiley &Sons 1998.
[7] Kahia, B., Keller, J. **Coordinating the Internet** .The MIT Press. 1997.
[8] Hugues, L. **Internet e-mail protocols, standars, and implementation.** Artech House. 1998.
[9] Guzmán, A, Hernández, L. "**The CIC Digital Library**".
[10] Guzmán,A CLASITEX: A SEMANTIC ANALYZER FOR SPANISH ARTICLES
[11] Tanenbaum, A. S. ***Operating Systems Design Implemtations***. Prentice-Hall. (1987).

Recursos
[12] Java
http://java.sun.com/docs/white/langenv/index.html
[13] Artículo de Servlets
http://java.sun.com/products/servlet/whitepaper.html
[14] Tutorial de Servlets
http://java.sun.com/docs/books/tutorial/servlets/index.html
[15] Especificación para servlets
http://java.sun.com/products/servlet/2.1/servlet-2.1.pdf
[16] La documentación Javadoc de la API de servlets
http://java.sun.com/products/servlet/2.1/api/packages.html
[17] Lugar para descargar JSWDK1.0.1
http://java.sun.com/Download3
[18] Curso del Instituto MageLang sobre la funcionalidad de servlets
http://developer.java.sun.com/developer/onlineTraining/Servlets/Fundamentals/index.html
[19] Instalación de JSWDK
http://java.sun.com/products/servlet/README.html
[20] Lenguaje HTML
http://www.w3.org/TR/REC-html32.html
http://www.ncsa.uiuc.edu/General/Internet/WWW/HTMLPrimerAll.html
[21] ¿Qué es un buscador?
http://www.rionet.com.ar/rionet/buscar1.html
[22] Arquitectura cliente/servidor.
http://www.inei.gob.pe/cpi/bancopub/libfree/lib616/index.htm
[23] Internet
http://www.areas.net/servicio/funciona/conexion/tcpip.htm