Informe Técnico - Technical Report
DPTOIA-IT-2006-02
Marzo, 2006

Diseño de *spiders*

Carlos G. Figuerola
José Luis Alonso Berrocal
Ángel F. Zazo Rodríguez
Emilio Rodríguez Vázquez de Aldana

Departamento de Informática y Automática
Universidad de Salamanca

Revisado por

Dr. D. Luis Alonso Romero
Departamento de Informática y Automática
Universidad de Salamanca

Dra. Da. María N. Moreno García
Departamento de Informática y Automática
Universidad de Salamanca

Aprobado en el Consejo de Departamento de 24 de Marzo de 2006

Información de los autores:

Dr. Carlos G. Figuerola
Dr. José Luis Alonso Berrocal
Dr. Ángel F. Zazo Rodríguez
Emilio Rodríguez Vázquez de Aldana

Todos ellos son integrantes del
Grupo de Recuperación Automatizada de la Información (REINA)
Área de Lenguajes y Sistemas Informáticos.
Departamento de Informática y Automática.
Facultad de Traducción y Documentación. Universidad de Salamanca
C/ Francisco Vitoria, 6-16. 37008 - Salamanca
`reina@usal.es`
`http://reina.usal.es`

Este documento puede ser libremente distribuido.
©2006 Departamento de Informática y Automática - Universidad de Salamanca.

Resumen

Este trabajo pretende recoger la experiencia de sus autores en el diseño de programas que recorren automáticamente eso que se ha dado en llamar *ciberespacio*.

Se muestra la estructura básica de un *spider*, así como algunas de las herramientas básicas de programación para la construcción de éstos. Se examinan diversas opciones de rendimiento de estos programas y convenciones utilizadas en la exploración automática del *web*. Se muestran también los problemas más frecuentes en el funcionamiento de estos programas, y diversas opciones para abordar dichos problemas.

Abstract

This paper aims to show the authors's experience in developping software that automatically explore cyber-space.

The basic structure of a spider is showed, as well as the major tools to build this software. Several performance issues of this kind of software are revised, besides the standards and conventions used in automatic exploration of the web. Also are showed the more frequent troubles found in the work of this software, and several issues to resolve them.

Índice

1. Introducción

El *web* es una colección de billones de documentos escritos de tal forma que pueden ser citados usando hiperenlaces y conformando el denominado hipertexto. Estos documentos, o páginas *web*, tienen unos pocos cientos de caracteres escritos en infinidad de idiomas y que cubren esencialmente todas las materias del saber humano. Estas páginas *web* se encuentran instaladas en un servidor *web* y son servidas ante las peticiones del cliente empleando el protocolo `http` y presentadas al usuario por los visores *web*.

Para poder analizar esta enorme cantidad de páginas es necesario elaborar programas automáticos que permitan analizar los documentos hipertexto recorriendo toda la red a través de los hiperenlaces que los conectan.

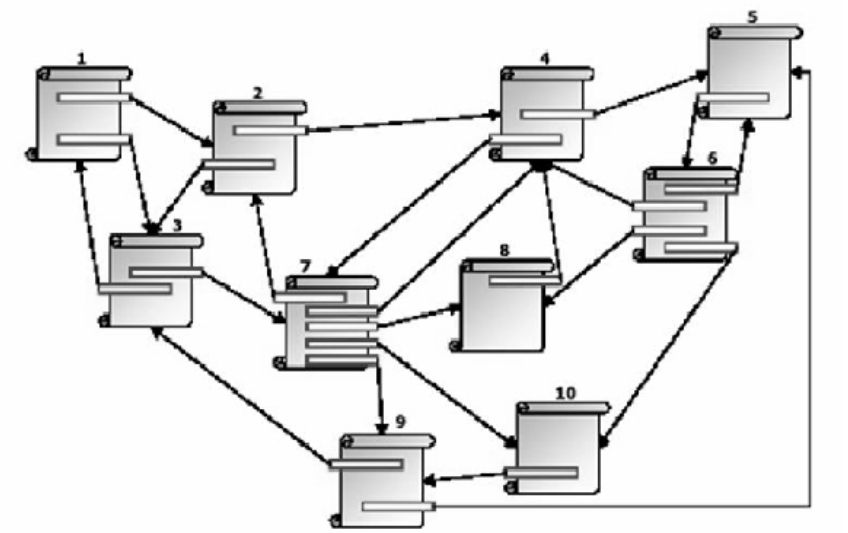

Figura 1: Esquema básico del *web*.

La bibliografía existente sobre este particular es extensa y variada destacando los trabajos de [11], [1], [5], [4] que dan una idea de los mecanismos necesarios para el trabajo con este tipo de herramientas.

Hay muchas razones para explorar de forma automática la red. La más inmediata deriva de su tamaño, que hace implanteable una exploración manual, salvo para fracciones muy pequeñas. La estructura de grafo hace factible la utilización de programas de navegación automática; tales programas se conocen con diversos nombres: *robots*, *spiders*, *wanderers*, o incluso gusanos (*worms*).

Este trabajo pretende recoger la experiencia de sus autores en el diseño de programas que recorren automáticamente eso que se ha dado en llamar *ciberespacio*, en la confianza de que pudiera resultar útil a quienes tengan que enfrentarse con la tarea de escribir uno de estos programas; y, también, simplemente, de quienes quieran entender algo mejor su funcionamiento y los problemas con los que han de enfrentarse.

La organización de este trabajo es como sigue: en la siguiente sección se describe la estructura básica de un *spider*. A continuación se mencionan algunas de herramientas de programación útiles en este campo, para pasar, después, a considerar los problemas de rendimiento (básicamente, velocidad de exploración) y formas de abordarlos. Luego se describen los Estándares de Exclusión de Robots, utilizados por aquellos propietarios o administradores de sitios de red que no quieren ser explorados (o que desean poner algún tipo de limitación). Se describe también el tratamiento de algunos errores frecuentes en la negociación con servidores.

Son diversas las razones por las que uno puede necesitar un *spider* o *robot* que recorra

automáticamente el ciberespacio. Algunas de las más frecuentes pueden ser [7]:

- Análisis estadístico de la red. Desde la obtención de datos simples (tamaño de sedes *web*, por ejemplo) hasta trabajos mucho más complejos de lo que ha dado en llamarse *cibermetría* requieren de herramientas automáticas que recolecten, recuenten y calculen coeficientes.
- Labores de mantenimiento: detección de enlaces muertos o erróneos, verificación de sintaxis `HTML`, etc.
- Tareas de copiado o *mirroring* con diversos propósitos.
- Recuperación de Información, una de las tareas más obvias.

Hay, como puede suponerse, muchos *robots* pero no todos están realmente disponibles para uso abierto. Ciertamente, hay, por ejemplo, un montón de servicios de búsqueda en Internet, cada uno de los cuales tiene su propio *spider* o *robot*; pero debemos contentarnos con los datos que éstos nos facilitan, lo cual muchas veces no resuelve nuestros problemas. Por ejemplo, muchos buscadores nos pueden dar de una forma aproximada el número de páginas que hay en un dominio dado, pero, aparte de la mayor o menor precisión de esa cifra, será difícil obtener más detalles, por la simple razón de que no tenemos control sobre el *spider* de ese buscador.

De otro lado, el valor económico de muchos de estos servicios hace que, en ocasiones, la literatura sea bastante evasiva acerca de detalles concretos sobre el funcionamiento de muchos de ellos. Así, no son muchos los *spiders* disponibles para uso abierto, es decir, con licencia `GNU` o parecida.

Algunos de los que podemos tener disponibles pueden ser:

- `WebBot`. Disponible en la dirección `http://www.w3.org/Robot/`, se trata de un proyecto del *World Wide Web Consortium* (W3C).
- `Harvest-NG`. Disponible en la dirección `http://webharvest.sourceforge.net/ng/`, se trata de un conjunto de utilidades para costruir *webcrawlers* y está escrito en lenguaje *perl*.
- `Webvac Spider`. Disponible en la dirección `http://dbpubs.stanford.edu:8091/~testbed/doc2/WebBase/webbase-pages.html`, es un proyecto de la Universidad de Stanford.
- `SocSciBot 3` y `SocSciBotTools`. Disponible en la dirección `http://webharvest.sourceforge.net/ng/http://socscibot.wlv.ac.uk/`, es una opción interesante con utilidades adicionales.
- `WIRE crawler`. Disponible en la dirección `http://www.cwr.cl/projects/WIRE/`, es el desarrollo del Centro de Investigación del *Web* (CWR) dirigido por el Dr. Ricardo Baeza-Yates.
- `SacarinoBot` y `EloisaBot Tools`. Se trata de un desarrollo nuestro y estará disponible proximamente.

2. Estructura de un *spider*

El procedimiento básico de un *robot* consiste en suministrar una URL inicial o un conjunto de ellas, obtener la página *web* correspondiente y a continuación extraer todos los enlaces existentes en dicha página. Con los enlaces obtenidos es necesario realizar una serie de operaciones previas de normalización entre las que podemos indicar las siguientes:

- convertir URL a minúscula
- eliminar anclas
- adecuar el sistema de codificación
- emplear la heurística para la determinación de la página por omisión
- resolver los URL relativos

A continuación será necesario comprobar los URL que se habían seguido previamente y en caso de no haberlos recorrido introducirlos en una cola de URL a seguir. Después, normalmente, almacenamos la información, bien en bases de datos o en estructuras de ficheros con codificacion ASCII. Finalmente obtenemos el URL del siguiente enlace a seguir y comienza de nuevo el proceso.

Pero uno de los aspectos más interesantes es la gestión de la lista en la que se van almacenando los enlaces que nos vamos encontrando. El aspecto clave aquí es el orden en que vamos explorando las direcciones que guardamos en dicha lista; lista que, por otra parte, es dinámica, puesto que constantemente se están añadiendo nuevas direcciones. Suelen distinguirse, en función de esto, tres formas básicas de exploración:

1. Recorrido en anchura (*breadth-first*).
2. Recorrido en profundidad (*depth-first*).
3. El mejor posible (*best-first*).

Las dos primeras son bien conocidas y dependen de que dicha lista se organize como una pila o como una cola. En el primer caso, el entorno inmediato y más cercano a una página es explorado primero; resulta útil cuando se buscan exploraciones exhaustivas de un dominio, aún cuando éstas se limiten a una determinada profundidad solamente.

Si la lista se organiza como una cola, la exploración se aleja inmediatamente del punto de partida, y hasta que cada posible camino no ha sido recorrido no se vuelve a éste. Dado el tamaño de la red, si no se impone algún tipo de limitación (circunscribirse a un dominio dado o a un número de nodos de alejamiento), esto puede suponer en la práctica, que no se vuelve nunca al entorno del punto de partida. Cuando se explora la red con ciertos propósitos, esto puede resultar útil, pues ayuda a descubrir información poco obvia, o difícil de obtener mediante navegación manual.

La figura 2 muestra un esquema de cómo serían los dos primeros recorridos y lo resultados que se obtendrían.

Ordenar la lista de direcciones a explorar por el mejor candidato significa aplicar el criterio que se estime más adecuado, en función del uso que se quiera hacer del *robot*. Un ejemplo simple sería priorizar o explorar antes aquellos elementos de la lista más citados, es decir, los enlaces que se repiten más veces, en la suposición de que deben ser los más prestigiosos o importantes. Es posible aplicar otro tipo de coeficientes más sofisticados, naturalmente.

Otro ejemplo, cuando lo que se busca es que el *robot* obtenga páginas o direcciones sobre un tópico determinado, es priorizar aquellos enlaces que presumiblemente lleven a nodos más relacionados con ese tópico. Por ejemplo, porque tales enlaces hayan sido obtenidos en nodos o páginas similares o relacionados con el tópico en cuestión; o porque el texto del enlace contenga determinadas palabras relacionadas con dicho tópico [3].

Naturalmente, hay algunos detalles que deben ser resueltos: la negociación con los servidores de los cuales hay que obtener páginas o lo que sea; determinar límites en la exploración, dado

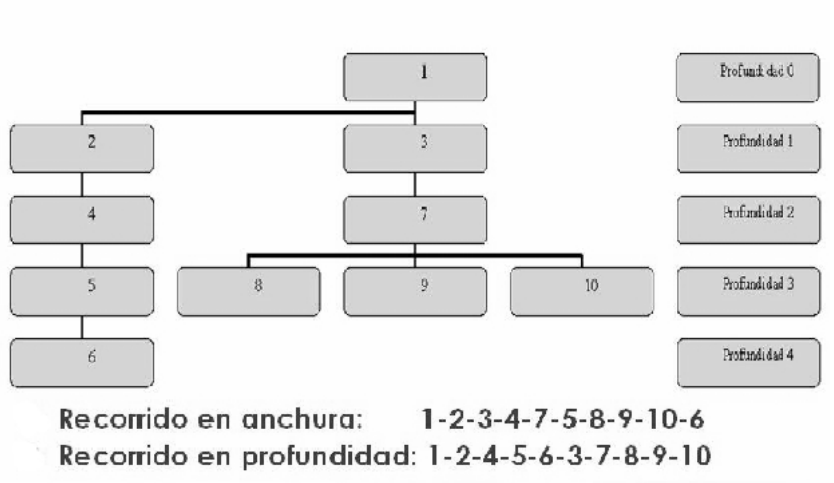

Figura 2: Resultados según el tipo de recorrido.

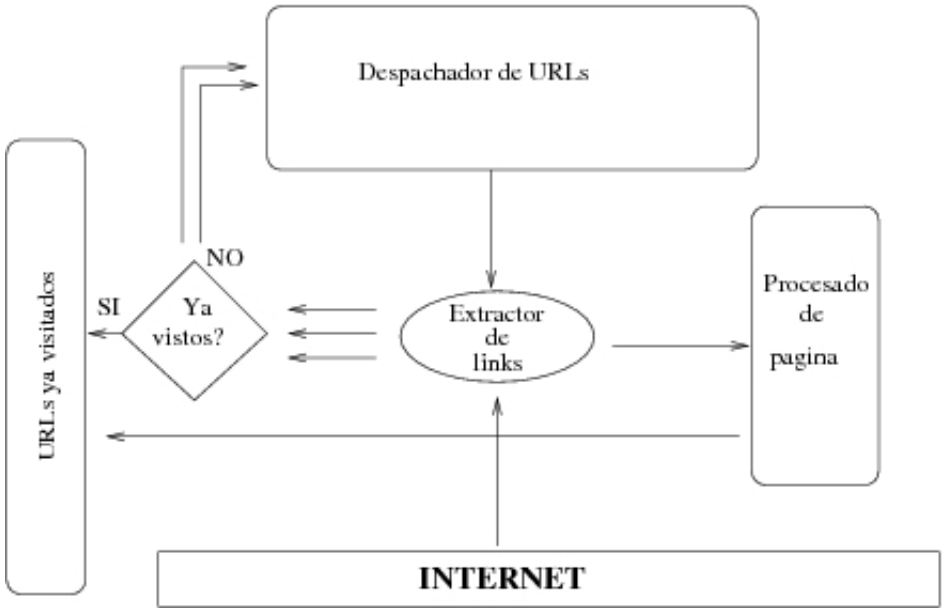

Figura 3: Esquema básico de un *robot*

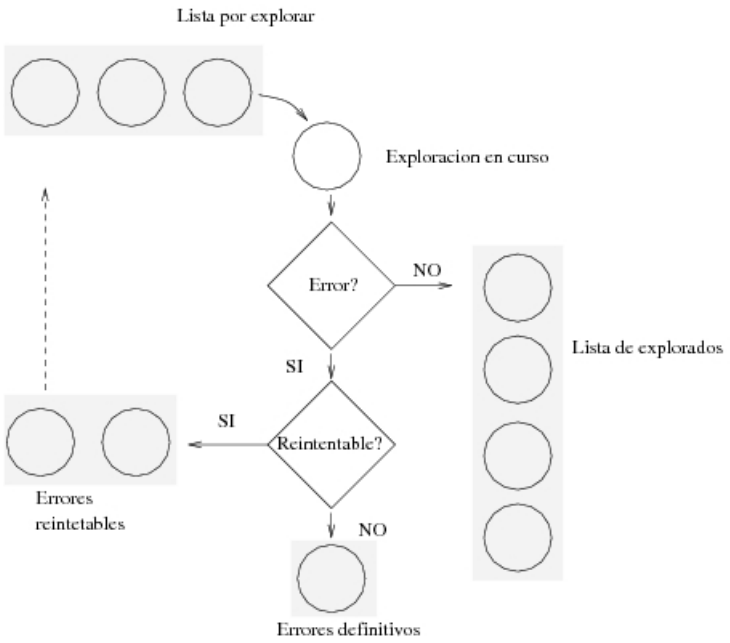

Figura 4: Proceso seguido en la exploración de un nodo

el tamaño y volatilidad de la red (a determinados dominios, hasta un determinado número de páginas, etc.); evitar exploraciones repetidas del mismo recurso y referencias circulares ...

Algunos de estos detalles suelen abordarse mediante la habilitación de listas adicionales; por ejemplo, para guardar los nodos o páginas ya explorados, o para guardar las que no pudieron explorarse por haberse obtenido algún tipo de error reintentable. Si, como se comentará, aprovechando la multitarea, se realizan exploraciones paralelas, es preciso retener un nodo que está siendo explorado, hasta que esa exploración haya finalizado, satisfactoriamente o no. El esquema de la figura 4 intenta representar el proceso seguido en la exploración de un nodo.

Un aspecto interesante que deben incorporar los *robots* es la capacidad para realizar recogidas incrementales. Esta idea consiste en guardar información adicional para la recogida realizada, de forma que se pueda implementar una nueva recogida posterior basada en seguir solamente los cambios realizados desde la recogida anterior. De esta forma podemos controlar de forma más eficaz las nuevas creaciones, actualizaciones y eliminaciones realizadas, permitiendo elaborar una política de recogidas posteriores que mejoren el rendimiento[2].

3. Herramientas

La construcción de un *spider* puede abordarse desde cualquier lenguaje de programación, y es claro que cada uno tiene sus ventajas y sus inconvenientes. Pero, independientemente de esto, un *spider* tiene que resolver una serie de tareas, algunas de las cuales puede ser más fácil o más eficiente resolverlas con el uso de determinadas bibliotecas, determinadas estructuras de datos o determinados métodos (por ejemplo: *threads*, bloqueos de registros, etc.), no siempre disponibles o fáciles de implementar en todos los lenguajes.

En este sentido, las operaciones básicas a resolver son:

- acceso a funciones de red y negociación con servidores
- análisis de páginas o recursos obtenidos para la obtención o extracción de los hiperenlaces
- la gestión eficaz de la lista de nodos a explorar

La herramienta o utilidad por excelencia es la propia librería *WWW* del *W3 Consortium* [http://www.w3.org/Library/]. La librería es de uso abierto mediante una licencia compatible GPL [`http://www.gnu.org/philosophy/license-list.html`], y tiene la ventaja de ajustarse al protocolo estándar `HTTP 1/1`, entre otros. Evita, naturalmente, tener que operar con la red a bajo nivel, y está pensada para, entre otras cosas, soportar todo el peso de la negociación con los servidores de los cuales hay que obtener algún recurso.

Soporta `HTML 4` (estándar), por lo que facilita la obtención de hiperenlaces. Bien documentada, entre las aplicaciones de demostración que incluye hay, precisamente, un pequeño *robot*.

Su uso, sin embargo, puede resultar un tanto farragoso. Afortunadamente existe un interfaz de esta librería para *Perl*: *LWP* [`http://lwp.linpro.no/lwp/`]. Bien documentado, su uso simplifica la programación y el código notablemente. Provee acceso a recursos `http`, `https`, `ftp`, `gopher`, `news`, `file` y `mailto`; soporta mecanismos de autentificación `basic` y `digest`, redirección, *proxies*, *cookies*; naturalmente, es capaz de analizar y descomponer objetos `HTML` y soporta el Estándar para la Exclusión de Robots, del que se hablará luego.

Para *Java*, hay que mencionar *WebSPHINX*, una librería de clases que permite construir *crawlers* especializados [`http://www.cs.cmu.edu/~rcm/websphinx/`]. Más sencilla es la clase `spider` de *ACME Labs (Purveyors of fine freeware since 1972. On the net since 1991)* [`http://www.acme.com/`].

4. Exploración educada

La expresión *educada* probablemente no es la más afortunada, puesto que no es un mero asunto de buenas formas; si un servidor *web* se ve perjudicado por un *spider*, tarde o temprano procurará evitar ser explorado por éste. La idea básica es efectuar la exploración consumiendo los menos recursos posibles de los servidores, de manera que el trabajo de éstos no se vea afectado; y, al mismo tiempo, observar las posibles restricciones impuestas por los administradores de dichos servidores.

4.1. Estándar de Exclusión de *Robots*

En muchas ocasiones los administradores no desean que ciertas partes (o, incluso la totalidad) de un sitio *web* sea explorado por *spiders*; páginas en construcción, directorios que no contienen documentos HTML, son casos típicos, aunque no los únicos. Muchos administradores no desean soportar un tráfico que no consideran útil.

Existen diversas formas para intentar evitar *spiders* no deseados; básicamente, el uso de determinadas etiquetas META, y lo que se conoce como *Estándar para la Exclusión de Robots*. Por lo que se refiere a las etiquetas META, en algunas páginas HTML podemos encontrar una de estas etiquetas con el atributo `NAME` como *robots*, y `CONTENT` con los valores *noindex* o *nofollow* [8].

El Estándar para la Exclusión de Robots o `SRE` (*A Standard for Robots Exclusion*) no es, en realidad, un estándar oficial, sino un simple acuerdo o consenso entre los participantes en diversas listas de correo sobre *robots* y otros temas relacionados con la tecnología *WWW*; este acuerdo se remonta a 1994, y desde entonces se ha consolidado de forma importante, de forma que puede considerarse un estándar *de facto* [6].

Básicamente lo que hace el `SRE` es proveer un medio para que los administradores de sitios *web* expresen qué partes no desean que sean exploradas, así como qué *spiders* están autorizados y cuáles no.

El mecanismo es sencillo: consiste en colocar un fichero en el `URL` local `/robots.txt`. Este fichero es opcional, y se entiende que, cuando no existe en un determinado *web* no hay restriccio-

nes en su exploración. En su interior, mediante unas sencillas declaraciones, el administrador del *web* puede indicar qué partes está permitido y cuáles no; también, si se desea, se puede permitir o denegar la exploración a determinados *robots*.

Es importante comprender que el `SRE` es un pacto simplemente, no una barrera real; esto significa que un *spider* puede hacer caso omiso de él y explorar lo que le venga en gana. Más aún, el *spider* debe estar expresamente programado para buscar el fichero `/robots.txt`, leer y seguir sus reglas.

4.2. *Rapid Fire*

Cuando se explora un servidor determinado, y, especialmente cuando sólo se pretende explorar ese servidor, y nuestro *spider* se concentra en él, nos encontramos conque, desde una misma máquina (la nuestra) se producen numerosos y seguidos intentos de conexión al mismo objetivo, aunque sea solicitando páginas diferentes. El servidor explorado puede interpretar esta situación como un ataque de denegación de servicio (DoS), y responder con un bloqueo de dichos intentos de conexión.

La solución obvia es implementar un retardo entre demanda y demanda a un mismo servidor. La duración de este retardo es controvertida, pero hay quien recomienda entre 30 y 60 segundos [10]. Sea como fuere, si nuestra exploración se centra en un solo servidor y tenemos que esperar varios segundos entre cada página a explorar, es claro que los tiempos de exploración pueden ser poco útiles.

Si se van a explorar varios servidores podemos aprovechar la capacidad multitarea del sistema operativo e ir intercalando servidores entre retardo y retardo.

Cuando se opera con un número pequeño de servidores, nuestra experiencia es que puede uno permitirse el lujo de un cierto atrevimiento, con algunas pruebas previas. En realidad, muchos servidores (tanto las máquinas como los programas) actuales pueden soportar perfectamente un número alto de demandas sin necesidad de retardos importantes. Solamente con algunos servidores que soportan mucho -muchísimo- tráfico hemos encontrado problemas en este tema. De manera que se pueden hacer pruebas parciales con retardos cortos, para ver hasta dónde se puede llegar con un determinado servidor.

4.3. Páginas con contraseña

Numerosas páginas y recursos están protegidas mediante contraseña y, cuando ésta nos es desconocida, nuestro *spider* obtendrá un error, típicamente el 401 (RFC 2616). Una gestión cuidadosa de los errores devueltos debería evitar reintentos de conexión; los administradores de algunos servidores podrían interpretar los intentos repetidos como un ataque a la contraseña por fuerza bruta, y tomar medidas contra nuestro *spider* (y, probablemente, también contra nosotros).

5. Rendimiento

En el trabajo de muchos *spiders* un factor clave es la velocidad de exploración; ésta depende de diversos factores, pero el determinante, sin duda, es la limitación que nos impone la propia red. No solamente el ancho de banda, sino la capacidad de respuesta de aquellos servidores con los que contactamos. A esta capacidad de respuesta hay que añadir los retardos que introduzcamos por cortesía, tal como se indica más arriba.

Dicho de otra manera, en el proceso de solicitar una página de un servidor y recibirla, la mayor parte del tiempo un *spider* está haciendo, literalmente, nada. Y si introducimos retardos,

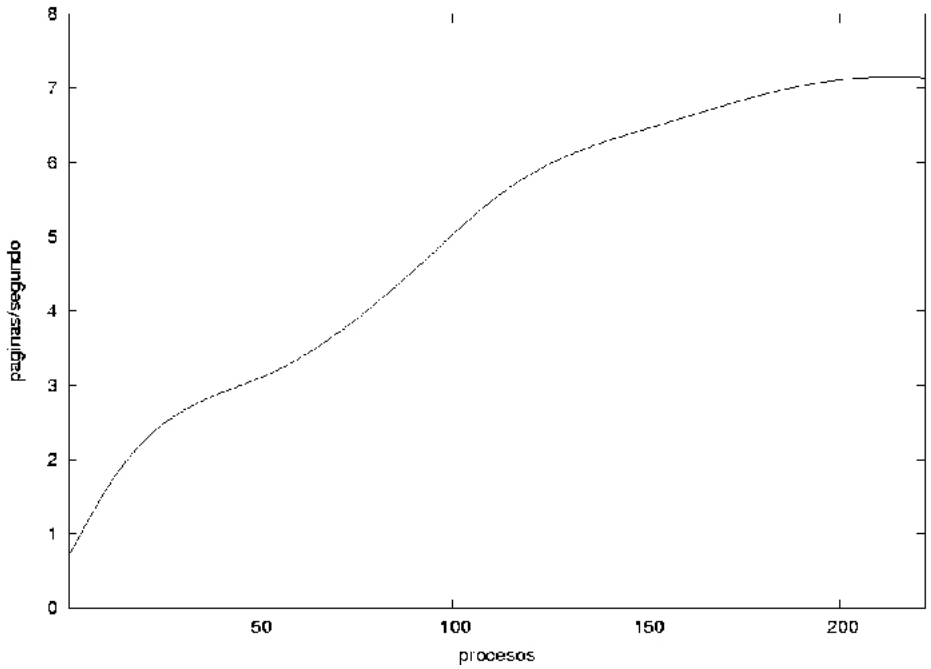

Figura 5: Número de procesos y páginas exploradas por segundo

más de lo mismo. Con un sistema operativo multitarea, la clave está en aprovechar esos tiempos muertos para obtener otras páginas; ocupar al máximo posible el ancho de banda disponible durante todo el tiempo.

Esto significa disponer de varios subprocesos o hebras (*threads*) simultáneos, explorando nodos diferentes. La diferencia teórica más importante entre subprocesos (*fork*) y *threads* es que los primeros son procesos independientes, cada uno con su propia zona de memoria y, por tanto, su copia independiente de código y datos. Las hebras o hilos, sin embargo, comparten memoria y, por tanto, acceden a los mismos datos y pueden modificarlos incontroladamente si no se toman precacuciones (uso de bloqueos, semáforos, etc.) [9].

Las hebras o hilos, por consiguiente, parecen, al menos en teoría, menos costosos en cosumo de memoria. Los datos compartidos, permiten, debidamente controlados, utilizar listas de nodos a explorar comunes a los distintos hilos. De otro modo, si cada hebra o subproceso mantiene sus propias listas, corremos el riesgo relativo de explorar con varios hilos o subprocesos la misma página, desde el momento en que ésta puede estar enlazada desde distintos nodos de distintos servidores.

La figura 5 ilustra el aumento de velocidad en función del número de *threads*. Éste tiene límites impuestos por el sistema operativo y la cantidad de memoria disponible, pero antes de que se alcancen éstos, probablemente habremos agotado nuestro ancho de banda.

Más aún, si todos los hilos envían requerimientos al mismo servidor, colapsaremos éste con facilidad. Esto significa que las hebras o los subprocesos dan mejores resultados cuando operan contra servidores distintos.

Los datos reflejados en el gráfico debe tomarse con cierta prevención, pues han sido obtenidos efectuando pruebas con una línea ADSL de medio Mega, y sólo con un par de muestras. Pero son lo suficientemente ilustrativos acerca de la eficacia de los *spiders* multihilo.

1XX	Código informativo, petición recibida
2XX	Exito, la petición ha sido recibida, comprendida y aceptada
3XX	Redirección, otra acción debe ser emprendida a continuación para completar la petición
4XX	Error del cliente, la petición contiene algún error
5XX	Error del servidor, éste no pudo satisfacer una petición aparentemente correcta

Cuadro 1: Códigos de Error HTTP

6. Problemas frecuentes y opciones de funcionamiento

6.1. Tratamiento de Errores

Típicamente el protocolo `HTTP` especifica una serie de códigos de estado que cualquier *spider* debería examinar (RFC 2616). Algunas de las situaciones posibles se han comentado ya, como es el caso de las páginas que requieren autentificación para acceder a ellas. Desde el punto de vista del diseño de un *spider*, merece la pena detenerse en aquellos errores que pueden considerarse temporales, o pasajeros. Un caso típico es la situación de *timeout*; el exceso de tiempo en la respuesta por parte de un servidor puede requerir diversos intentos, hasta que éste se encuentre accesible. Naturalmente, hay servidores que, por las razones que sea, nunca pasan a ser accesibles; una política razonable puede ser comenzar con un *timeout* corto, que puede ir incrementándose en los sucesivos intentos, hasta un límite razonable.

De otro lado, un *spider* bien hecho debería registrar adecuadamente los distintos códigos de estado, junto con otra información que ayude a desentrañar situaciones atípicas o infrecuentes. Hasta límites razonables, no hay porqué presuponer que las especificaciones del protocolo se siguen a rajatabla. Por ejemplo, no suele ser buena idea filtrar páginas basándose en el `Content-Type`; muchas páginas, simplemente no lo tienen, y otras tienen un tipo equivocado.

6.2. Redireccionamientos

Muchos enlaces son redireccionados de forma automática a otras páginas, por diversas razones. Depende de los objetivos del *spider* determinar si el redireccionamiento ha de seguirse o no, aunque lo más normal es que se siga, tal como los diseñadores de la página redireccionada han dispuesto.

Los navegadores siguen de forma automática los redireccionamietos, pero los *spiders* no, o al menos no necesariamente. Depende, en buena medida, de cómo estén programados; también, de cómo estén diseñadas las páginas redireccionadas. Básicamente, hay dos formas de hacer redirecciones: la primera es gobernada directamente por los servidores y produce un código de estado de redirección (código `3XX`); la segunda deja en manos de los navegadores seguir la redirección, y se efectúa mediante una etiqueta `META`:

```
<META HTTP-EQUIV="Refresh" URL=" ... ">
```

Nuestro *spider* debe estar preparado para reconocer este tipo de directiva.

6.3. Páginas dinámicas y páginas estáticas

En muchas ocasiones los usuarios de *spiders* no desean que éstos recorran páginas dinámicas; éstas son generadas al vuelo por terceros programas, generalmente (aunque no necesariamente)

en función de datos introducidos en formularios. Hay diferentes razones para no querer seguir este tipo de páginas, y dependen de los objetivos de los usuarios de los *spiders*.

El mecanismo fundamental para evitar páginas de este tipo es descartar los enlaces que contengan el símbolo `?` en su URL. Este símbolo se utiliza como separador estándar en la cadena que se envía a los programas que, en los servidores, generan las páginas dinámicas. Sin embargo, dichos programas pueden utilizar un separador diferente; esto no es frecuente, pero nosotros hemos encontrado casos en los que, intencionadamente, se ha hecho así. Por ejemplo, el servidor `http://sid.usal.es` utiliza en algunos de sus servicios de búsqueda el símbolo _; sabiamente combinado con enlaces como *página siguiente*, *página anterior* y similares, consigue que todos los *robots* rastreen al completo su amplia base de datos, aunque se trate de páginas dinámicas.

6.4. Recursos no deseados

Normalmente no deseamos explorar todos los recursos enlazados desde las páginas por las que va pasando el *spider*. Un situación típica se produce cuando un fichero de gran tamaño (por ejemplo, un ejecutable) es enlazado desde una página *web*. La forma más frecuente de abordar esto es descartar los enlaces a seguir en función de su tipo. Ya hemos comentado antes que no deberíamos fiarnos de las cabeceras `Content-Type`, así que podemos recurrir a las extensiones de los ficheros enlazados, y permitir que el *spider* siga sólo unas determinadas.

6.5. Páginas duplicadas, URLs duplicados

Como se ha comentado antes, una de las cosas básicas que un *spider* debe hacer es no visitar varias veces el mismo sitio, para lo cual, generalmente, lo que se hace es mantener una lista o registro de sitios visitados y, antes de ver uno nuevo, comprobar si no hemos estado ya allí. Sin embargo, hay URLs que, siendo cadenas de texto diferentes, en realidad apuntan al mismo sitio. Por ejemplo, la dirección `http://reina.usal.es` es la misma que `http://reina.usal.es/index.htm`.

Algo más complicado es el caso de los alias en los nombres de los servidores. Por ejemplo, el mencionado antes `reina.usal.es` no se llama realmente así, sino `reina.rec.usal.es`; en ocasiones, las redes se vuelven más complejas y se reestructuran, cambiando nombres y dominios. Otras veces, direcciones en las que se detectan errores parecen más fáciles de corregir poniendo un alias en el servidor de nombres con la dirección o nombre erróneo.

Una reacción obvia es convertir los nombres simbólicos a IPs. Sin embargo, aquí topamos con los servidores virtuales, de manera que varios de ellos, diferentes y con distinto nombre, pueden estar albergados en la misma dirección IP.

6.6. URLs mal formados

Es relativamente frecuente encontrarse con enlaces mal formulados. La mayor parte de ellos, una vez convertidos a forma absoluta, dan lugar a direcciones erróneas que, obviamente, no pueden ser alcanzadas. Sin embargo, no siempre ocurre así; direcciones como `http://abubilla.usal.es/lema/../index.html` son resueltas exitosamente por los servidores (en este ejemplo, devolviéndonos directamente a `http://abubilla.usal.es`). `http://abubilla.usal.es/lema/../../../../../index.html` hará exactamente lo mismo. Pero si en esa página hay un enlace a sí misma (cosa frecuente si se trata de *home*), el URL se irá haciendo cada vez más largo, aunque remitiendo siempre a la misma página. Difícilmente saldremos de ahí, salvo con desbordamientos de memoria, dependiendo de cómo esté escrito nuestro *spider*.

De otro lado, algunos servidores están configurados para responder con una página generada

al vuelo en lugar de con el código de estado correspondiente, cuando una página requerida es inexistente. Esto puede deformar la visión que podemos hacernos de ese sitio *web*, en el sentido de que encontraremos menos enlaces erróneos de los que realmente hay y algunas páginas pueden aparecer recibiendo muchos más enlaces de los que realmente hay.

Pero hay casos algo más enrevesados. Por ejemplo, la página de búsquedas en el catálogo de la biblioteca de cierta universidad es generada sobre la marcha por el programa que gestiona esa biblioteca. Dicha página tiene un enlace para modificar el formato de salida de los resultados de una búsqueda; pero, inexplicablemente, ese enlace está activo y puede ser seguido sin necesidad de hacer una búsqueda primero. Esto es exactamente lo que hará un *spider*, ante lo cual el programa de la biblioteca genera al vuelo una página advirtiendo de un error en el programa y redirigiendo a una nueva página de búsqueda, también generada al vuelo. Como todas estas páginas generadas sobre la marcha tienen nombre diferente (por ejemplo: `axY3hh98MF.html`, `OQertWP44.html` y cosas así) nos encontraremos dando vueltas sobre el mismo sitio indefinidamente.

6.7. Caracteres extraños

Muchas páginas son generadas por programas automáticos. Esto ocurre no sólo con las páginas dinámicas, sino también con las estáticas; sucede que tales programas a veces manejan unos URLs que contienen lo que podríamos llamar *caracteres extraños*. Nada que objetar, si los servidores y navegadores son capaces de manejarlos, pero el código de un *spider* debe estar preparado para tratar con ello adecuadamente. Por ejemplo, debe prever URLs con comillas en su interior, con signos `+` o con aperturas y cierres de corchetes, incluso sin nada dentro. En especial, *spiders* que manejan expresiones regulares deben ser cuidadosos con este extremo.

7. Conclusiones

La construcción de un *spider* puede ser necesaria para cumplir diversos objetivos. Aunque el diseño final de un programa de este tipo depende, en muchos aspectos, de esos objetivos, hay una serie de aspectos que pueden considerarse comunes a cualquier tipo de *spider*. Así, se ha mostrado la estructura básica de un *spider*, así como las herramientas de programación más frecuentemente utilizadas en la cosntrucción de este tipo de programas. Igualmente, se han comentado los problemas más importantes de rendimiento que afectan a este tipo de programas, y se han mostrado algunas de opciones más importantes con las que el diseñador de un *spider* debe enfrentarse. Aunque estas opciones dependen en buena medida del objetivo y finalidad de un *spider* determinado , conocerlas de antemano puede resultar de utilidad para el diseñador de un *spider*.

Referencias

[1] J. L. Alonso Berrocal, C. G. Figuerola, Á. F. Zazo, and E. Rodríguez. Agentes inteligentes: Recuperación autónoma de información en la Web. *Revista Española de Documentación Científica*, 26(1):11–20, 2003.

[2] R. Baeza-Yates. The web of spain. *UPGRADE*, 3(3):82–84, 2003.

[3] S. Beitzel, E. Jensen, R. Cathey, L. Ma, D. Grossman, O. Frieder, A. Chowdury, G. Pass, and H. Vandermolen. Task classification and document structure for known-item search. In *The Twelfth Text REtrieval Conference (TREC 2003)*. NIST Special Publication 500-255, 2003.

[4] C. Castillo. *Effective Web Crawling*. PhD thesis, Departament of Computer Science. University of Chile., november 2004.

[5] S. Chakrabarti. *Mining the Web : discovering knowledge from hypertext data*. Morgan Kaufmann, San Francisco, CA, 2003. ISBN: 1558607544.

[6] M. Koster. A standard for robot exclusion, 1994.

[7] M. Koster. Robots in the web: threat or treat? *ConneXions*, 9(4), 1995.

[8] M. Mauldin. Spidering bof report. In *W3C Workshop on Distributed Indexing/Searching*, Cambridge, MA, 1996.

[9] J. Palmero Esteban. Sistemas operativos multiproceso y multithread. *Login*, (15):19–22, 1996.

[10] V. Shkapenyuk and T. Suel. Design and implementation of a high-performance distributed web crawler. In *IEEE International Conference on Data Engineering (ICDE)*, 2002.

[11] M. Thelwall. A web crawler design for data mining. *Journal of Information Science*, 27(5):319–325, 2001.