

SharpSpider: Spidering the Web through Web Services

Ken Moody and Marco Palomino
Computer Laboratory
University of Cambridge
Cambridge, CB3 0FD
{Ken.Moody, Marco.Palomino}@cl.cam.ac.uk

Abstract

Web search engines have become an indispensable utility for Internet users. In the near future, however, Web search engines will not only be expected to provide quality search results, but also to enable applications to search and exploit their index repositories directly. We present here SharpSpider, a distributed, C# spider designed to address the issues of scalability, decentralisation and continuity of a Web crawl. Fundamental to the design of SharpSpider is the publication of an API for use by other services on the network. Such an API grants access to a constantly refreshed index built after successive crawls of the Web.

1. Introduction

Many applications have lately emerged with an intrinsic need of searching for data on the Web. Search via non-HTML interfaces, automated market research and automated comparison shopping are just a few examples. Although these applications cannot be seen as an extension of a search engine, they all call for systems to seek, download and index Web pages on a massive scale.

In the near future, Web search engines will not only be expected to provide quality search results, but also to enable remote access to their repositories. Due to their open standards, platform independence and focus on collaboration, *Web services* [5] seem to be an ideal environment for the integration between Web search engines and applications whose main input is data collected from the Web.

Here, we report on *SharpSpider*, a distributed, C# spider that has been designed to address the issues of scalability, decentralisation and continuity of a Web crawl. Fundamental to the design of SharpSpider is the definition and publication of an API for use by other services on the network. It is through this API that other parties interested in our software can connect remotely and issue queries to our constantly refreshed lexicons and indices.

2. Related Work

A *spider* is a program that automatically downloads Web pages, parses them to collect information and uses that information to download more pages. Most of the recently developed spiders consist of cooperating processes that download Web pages, extract their links and sometimes send those links to other peer processes responsible for them [7, 8, 9, 11, 13, 15]. SharpSpider also comprises various communicating instances spread across different computers, and we coordinate their execution dynamically. In section 4, we elaborate on our distributed architecture.

At present, *Google* [2] seems to be the only search engine researching the possibility of providing access to the indices created by a spider. In April 2002, Google released a beta version of a *Web API Service* [3] intended to allow developers to build applications on top of the Google search engine. Although this represents a major innovation, Google was not originally designed for this service. Its optimised infrastructure is tuned for end users, but imposes limitations on automated access from other services.

3. SharpSpider Features

All the data structures included in SharpSpider are designed to keep operational costs as low as possible. For instance, to maintain a list of the URLs already downloaded, we have implemented our own data store, based on the *extendible hashing algorithm* [14], as an alternative to a relational database server such as *SQL Server* or *Oracle*. This design choice gives us the opportunity of customising the algorithms for our specific needs.

SharpSpider maintains a list of the URLs waiting to be downloaded. For this purpose, we have implemented an array of priority queues, where each entry in the array contains a queue corresponding to a specific host. Each queue includes all the URLs of its associated host found during a crawl. If we are executing more than one instance of SharpSpider in different locations, then any URL that belongs to a

host not covered by the local array is sent to the instance responsible for it. If no other instance has been assigned to that URL, SharpSpider will store it on disk to be handled later, during the crawl of a different part of the Web.

Even after being downloaded, URLs remain enqueued, with their status updated according to their downloading history (for example, whether a URL content has changed since its previous download). In this way, we continuously revisit pages and constantly refresh our generated indices. In order to store very large priority queues efficiently, we have employed a technique similar to that described in [6], which uses a *buffer tree* to model a priority queue.

Harnessing the inherent parallelism of the Internet provides an opportunity to improve the performance of a spider. Thus, SharpSpider creates a pool of *downloader threads* that immediately after their creation enter a loop; they first wait for another URL to be scanned, and then start its download as soon as it becomes available.

Parsing of HTML files is programmed as an asynchronous, callback-based scheme. SharpSpider implements a callback interface that notifies the spider whenever certain tags are found in an HTML file. In this way, the spider gets URLs passed back to it as soon as they are found, giving it the possibility of downloading newly discovered links without waiting for the parser to finish.

SharpSpider adheres to the standard conventions for the *robots exclusion protocol* [12]. Given that the number of domains within a Web site is predictably small, and considering that each of our crawls is restricted to a limited number of sites, robot instructions can be held in main memory for the duration of a crawl.

If SharpSpider abruptly stops because of system failures, it can pick up where it left off and complete any unfinished processing without duplicating work. Periodically, a timer signals a callback to perform check pointing as a background task. The advantage of this approach is that the operating system or runtime platform can optimise the way in which these signals are managed and use fewer resources.

In the long run, we expect to transform part of the code written for SharpSpider into a suite of class libraries to support the implementation of other spiders and programs that browse and process Web pages automatically. This does not seem difficult to achieve, as we have programmed our spider through a component-based architecture. Every component of SharpSpider is specified by an abstract interface. As a result, multiple implementations can be provided for each component, or our own implementations can be extended through object-oriented sub-classing.

As of August 2003, we have performed repeated crawls of 50 Web sites of the University of Cambridge. At present, SharpSpider is capable of downloading 9,000 pages per minute, which is satisfactory but not yet ideal. From the first million URLs processed, HTML pages have accounted for a

substantial amount of the downloaded documents (91.07%). *JPEG* files are the second most retrieved type, but *PDF* and *Postscript* files are roughly as common as JPEGs. Our results seem to be influenced by the fact that we have crawled academic institutions only. Hence, multimedia files, in their various formats, are almost nonexistent.

4. A Distributed Crawl of the Web

Although executing multiple instances of a spider over a LAN increases crawling performance, we presume that major advantages can be obtained when different instances of the spider are launched simultaneously at distant geographical locations. At some stage, the different instances may need to communicate or transfer pages to a central location, but even in that case the network bandwidth consumption would be smaller, since the transfer could simply involve compressed or summarised information [10].

If two or more instances of the same spider run simultaneously, there is always the danger that they may be downloading the same Web pages, which, for various reasons, would be detrimental to the whole system. Therefore, the individual processes need to coordinate with each other to decide which pages to download next. We propose to set up this coordination dynamically.

Before starting a crawl, we divide the Web into logical partitions, and assign each partition to an instance of SharpSpider. We always partition the Web by site names, so that only pages at the same site belong to a given partition. After starting a crawl, an instance may find pages in its partition that have links to pages in a different partition. When this happens, that instance will not follow those links, but rather will transfer them to the instance that is responsible for them. Of course, each instance must have the knowledge of the mapping between other URL subsets and the location of the instances responsible for them, in much the same way that peer-to-peer nodes require a routing table. Each instance keeps its own database, which stores only URLs from the subset assigned to it. As a consequence, these databases are disjoint sets.

5. The SharpSpider Web Service API

Applications that need to run automated search engine queries may *pull down* results formatted in HTML and parse them to obtain what they are looking for (this technique is sometimes known as *Web-scraping*). However, if the site from which the HTML pages are being taken changes its format in any way, it is unlikely that the Web-scraping code will remain effective. In addition, some search engines, such as Google, specifically forbid Web-scraping [1], perhaps because they do not want others to parse their search results by program without their knowledge.

Offering regulated access to a search engine repository would eliminate the need for Web-scraping, and would benefit both users and providers of information. Web services could offer an ideal solution to this problem, as they are becoming a standard platform for application integration.

Publishing an API to regulate access to a repository would not impose any restriction on the programming language that other developers might wish to use. Once they have the proxy code to access our service, they could simply concentrate on their own algorithms.

Our choice of C# as the implementation language for the whole project has allowed us to take advantage of the facilities offered by Microsoft's solution to the Web services initiative, namely, the *.NET Framework*. Communication via the *Simple Object Access Protocol (SOAP)*, as well as support for different SOAP methods, is seamlessly integrated by means of the *.NET Framework*. In addition, a number of programming environments can automatically generate a service description for our development, which conforms to the *Web Service Description Language (WSDL)*. Further, some of the available toolkits can generate proxy code directly from our WSDL description file.

Due to space limitations, we cannot include the complete details of our API in this report. Its main method is designed to support a search request; it receives a query string and supplies in return a set of search results. The signature of this method is given below.

Search(requester_credentials, query, maxResults): The *requester_credentials* are used for authentication. The *query* is a string made of words separated by commas, and our service returns only pages that contain all the words in the query.

As our project advances, we expect to provide a more elaborate query syntax, such as that offered by most search engines. This will permit the use of special query terms in order to support additional capabilities.

6. Conclusions and Future Work

We have presented SharpSpider, a distributed, C# spider. Different instances of SharpSpider are responsible for distinct URL subsets. Whenever an instance comes across a URL that does not belong to its own subset, it forwards it to the appropriate instance. We also described a proposal to provide a standard interface to a constantly refreshed index built during continuous crawling of the Web.

Our immediate goals fall into two areas: testing and optimisation. Although we have already implemented the basic communication infrastructure to allow us to disperse instances of our spider, we still have to experiment with a large distributed crawl of the Web. We are currently looking for an opportunity to install SharpSpider in different lo-

cations and run a synchronised crawl. *PlanetLab* [4] may be an option.

We should also test our API extensively. Our code is still subject to experiment and a further technical report is being prepared for publication.

Acknowledgments

We are very grateful to Arasnath Kimis for reading our manuscripts and providing insightful comments. Palomino wishes to thank the *National Council for Science and Technology of Mexico (CONACYT)*, which supported his participation on this project.

References

- [1] Google Terms of Service, http://www.google.com/terms_of_service.html.
- [2] Google, <http://www.google.com>.
- [3] Google Web APIs Reference, <http://www.google.com/apis/reference.html>.
- [4] PlanetLab, <http://www.planet-lab.org>.
- [5] Web Services Activity, <http://www.w3.org/2002/ws>.
- [6] L. Arge. The Buffer Tree: A New Technique for Optimal I/O-Algorithms. Technical Report RS-96-28, BRICS: Basic Research in Computer Science, University of Aarhus, August 1996.
- [7] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. Trovatore: Towards a Highly Scalable Distributed Web Crawler. In *Proceedings of the 10th International World Wide Web Conference*, Hong Kong, May 2001.
- [8] S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. In *Proceedings of the 7th International World Wide Web Conference*, Brisbane, Australia, April 1998.
- [9] M. Burner. Crawling towards Eternity: Building an Archive of the World Wide Web. *Web Techniques Magazine*, 2(5), May 1997.
- [10] J. Cho. *Crawling the Web: Discovery and Maintenance of Large-Scale Web Data*. PhD thesis, Stanford University, Stanford, CA 94305, November 2001.
- [11] J. Edwards, K. McCurley, and J. Tomlin. An Adaptive Model for Optimizing Performance of an Incremental Web Crawler. In *Proceedings of the 10th International World Wide Web Conference*, pages 106–113, Hong Kong, May 2001.
- [12] M. Koster. Robots Exclusion, <http://www.robotstxt.org/wc/exclusion.html>.
- [13] M. Najork and A. Heydon. High-Performance Web Crawling. Technical Report 173, Compaq Systems Research Center, Palo Alto, CA 94301, September 2001.
- [14] R. Sedgewick. *Algorithms in C++*. Addison-Wesley, Boston, MA 02116, 2nd edition, April 1992.
- [15] V. Shkapenyuk and T. Suel. Design and Implementation of a High-Performance Distributed Web Crawler. In *Proceedings of the 18th International Conference on Data Engineering*, San Jose, California, February 2002.