High Performance Large Scale Web Spider Architecture

Kasom Koht-arsa and Surasak Sanguanpong

Department of Computer Engineering
Kasetsart University
Thailand
g4265077@ku.ac.th

Department of Computer Engineering
Kasetsart University
Thailand
nguan@ku.ac.th

Abstract

This paper describes a cluster-based high-performance web spider architecture. Its architecture has been designed for handling a very large number of web pages with both URLs contents compression. The method we used to fetch URLs has been designed for achieving maximum performance with respect to well-known spider's considerations. In experiments, our spider achieves an average download rate of 618 URLs/sec and 6 MBytes/sec.

1. Introduction

A web spider, some might called a crawler or a robot, plays an important role as an essential infrastructure of every search engines. It automatically discovers and collects resources, especially the web pages, from the Internet. As the rapidly growth of the Internet, a typical design of web spider may not cope with the overwhelming number of web pages.

Large-scale web spiders are facing many challenge problems such as dealing with a large number of URLs. The web spider must keep track of all the URLs it already visited. This is not a trivial task when billion of URLs must be tracked because it may not be able to keep them all in the memory. On the other hand, keeping them on disks is not a good option because it may result in poor performance. Dealing with web pages itself is, therefore, needed a special treatment because several billion of web pages imply that terabytes of storage space is required to well manage.

It is more challenge when the politeness of the web spider is taken into account. The performance of a web spider relies heavily on the parallelism of the fetching method, but fetching many URLs simultaneously from the same web server is said to be rude [6]. A web spider must have some mechanism to ensure that, while keeping a good parallelism, every machine in the cluster should not fetch too many pages from the same web server in a short period of time. This caution could be found in robot exclusion standard [5,6].

This paper describes a cluster-based high-performance web spider architecture that is designed to handle a large number of web pages. Although our ultimate goal is to develop a very high performance spider, but the way to design the polite spider is the first priority to be considered.

In the next Section, we describe the design of our design methodology. In Section 3, we describe our Spider's architecture. Finally, we present its performance measurements and conclude the paper.

2. Design

We design the web spider based on Beowulf cluster[4]. Some PCs are connected together using a high-speed network interconnection, and all of them are connected to the Internet through the campus network. The overall architecture is shown in Figure 1.

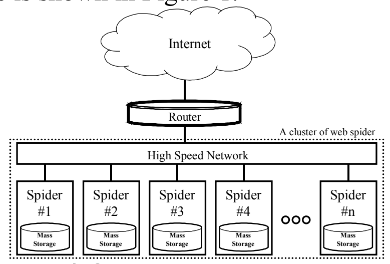

Figure 1 Overall system architecture

Fetching data from web servers in the Internet usually take a long time. The network might be congested, the server might be busy, and the network path might be very long so the latency is high.

Fetching data in parallel, either in the same machine or in the cluster, can minimize some of these problems. A Parallel of the pages fetching could yields more performance, a special care must be taken to ensure that the spider will not overload the network or target servers.

Like any parallel program, every node (Spider #1, spider #2…) in the cluster needs to communicate and synchronize to each other. Every node has to exchange the URLs. Our web spider has no central node, so every node does the same work like every other node, but they work in a different data set. The way how to distribute the data set are describes in section 2.1. The web spider also needs to make sure that every node will not fetch the data from the same web server at the same time, to solve this problem, we use a technique called 'phase swapping' as describes in section 2.2. To keep track of the URLs, we have design a compression algorithm describes in section 2.3.

2.1 Data distribution

All data from the web are distributed all over the nodes in the cluster. On any given URL, there is only one node that is responsible to fetch and keep that URL's data.
Every node is assigned with a unique Node-ID, starting from zero. A simple hash function based on the summation of every character in the URL, as in Figure 2, is then used to distribute the URLs among the nodes in the cluster. When a node found a new URL, it will hash the URL using the hash function to decide which machine in the cluster has to be responsible to that URL.

$$hash(url) = \sum_{i=0}^{i<len(url)} url[i] \bmod Nnode$$

Figure 2 An URL Hash function

The effect of the hash function is that every node has a disjoint set of the URLs, as shown in Figure 3. Supposed that 5 machines are used in the cluster. Every column belongs to a node in the cluster, S0 belongs to node 0, S1 belongs to node 1, and so on. Another effect of the hash function is that every node need not to keep track about the URLs of the other nodes. The communication between nodes is also minimized.

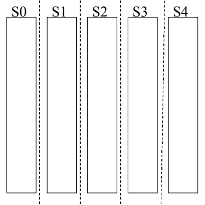

Figure 3 Disjoint set of the URLs (5 nodes)

2.2 Phase swapping

In order to prevent a web spider from fetching data from the same web server at the same time, we use a technique called 'phase swapping'. The swapping is not only solve simultaneous fetching problem, but also help a lot in reducing the number of URLs the spiders have to manage in any given time.
We hash the set of URLs (from the Figure 3) again by using another simple hash function based on the summation of every character in the host name part of the URL, as shown in Figure 4.

$$hash(url) = \sum_{i=0}^{i<len(host(url))} host(url)[i] \bmod Nphase$$

Figure 4 A Host Hash function

This hash function splits every set of the URLs further, as shown in Figure 5. Every set of URLs in the same row belongs to the same phase. At any given time, only one set (hilighted box) is active on any given node (row), as shown in Figure 6. This means that at any given time there will never be any node working (fetching) in the same set of web servers. Note that the number of phases must equal to or greater than the number of nodes.

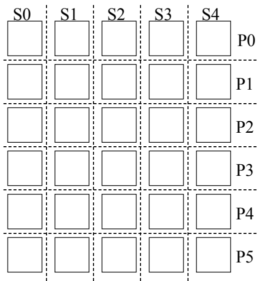

Figure 5 Disjoint set of the URLs further split by the second hash function (6 phases)

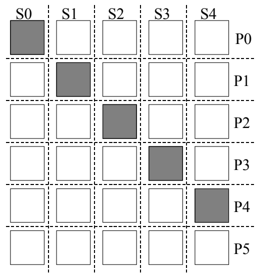

Figure 6 The active set of URLs

The active sets are changed by the system's wall clock, as shown in Figure 7. All nodes need to synchronize their wall clock using the network time protocol (NTP) to ensure that they will swap their working set (phase swapping) at the same time. There is no implicit synchronization between nodes for the phase swapping.
When the memory is tight, the inactive set can be stored on disk and loaded back when those sets become active again.

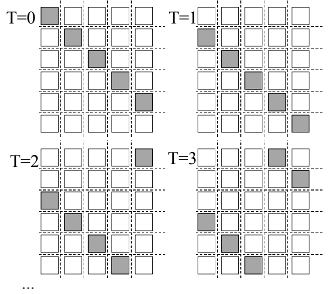

Figure 7 The active set of URLs are changing as time changed

2.3 URL compression

The web spider needs to keep track of the URLs. We have design a special compression algorithm [3] for keeping the URLs. It can compress the URLs very well, and the speed of storing, searching, and retrieving the URL back is very fast.

The algorithm we use to compress the URLs is similar to the Connectivity Server[1]; whose store the URLs by sorting them lexicographically and then store them as a delta-encoded text file. Their algorithm required that the URLs must be sorted before compressing them and that after compressed, no URLs can be added further. Furthermore, the algorithm also lacks the ability to randomly retrieve the URL back and the ability to search for the existence of any given URL; in order to find or to retrieve any single URL, the URLs must be scanned from the beginning.

Our method compresses the URLs by only keeping the differences of URLs tails. To find the minimum different (which yield the maximum common prefix), the sorting of URLs is required. But instead of sorting all URLs, we incrementally construct an AVL tree of URLs.

In our method, all URLs are stored on each node of an AVL tree. A node structure is illustrated in Figure 8. Each node in the tree contains five fields as the following,: `RefID` is a URL identification used to reference to its predecessor, `CommonPrefix` is a number of common characters referenced to its predecessor, `diffURL` is the tail of the URL that does not common to its predecessor, `Lchild` and `Rchild` are the pointers to the node's left subtree and right subtree respectively.

RefID	*CommonPrefix*	*diffURL*	*Lchild*	*Rchild*

Figure 8 A node structure of the AVL tree

The first encountered URL, which is the root, is assigned with the RefID 0. The RefID is increment by one for every new encountered URLs. Please note that the value in the field RefID of the root is undefined. The common prefix is set to zero, and the full URL is stored. The next or any latter URL must be compared with every node on the path between the root node and its predecessor to find the maximum common prefix. The reference ID of a new node is then point to that node, and the common prefix is set to the number of common characters, and the remaining of the URL is then stored. This is the way the compression is done.

Figure 9 shows an example of a construction of an AVL tree from 4 URLs: http://www.sun.com/, http://www.sgi.com/, http://www.sun.com/news/, and http://www.sun. com/news/archive/.

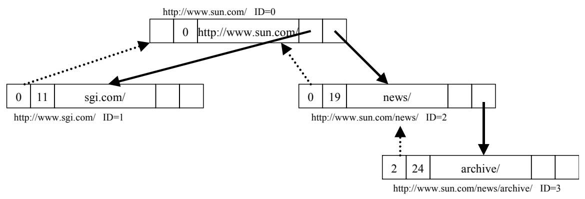

Figure 9 Compressed URL on an AVL tree

Retrieving any URL from the tree is very simple. The full URL can be re-constructed by following the path and concatenate all URL from the field `diffURL`. This approach can fulfil all the requirements of search engines and web spiders. The URLs can be added any time, the URLs can be searched, and the full URL can be retrieved.

3. Detailed architecture

Our web spider is implemented entirely in C++; it consists of five main objects as shown in Figure 10.

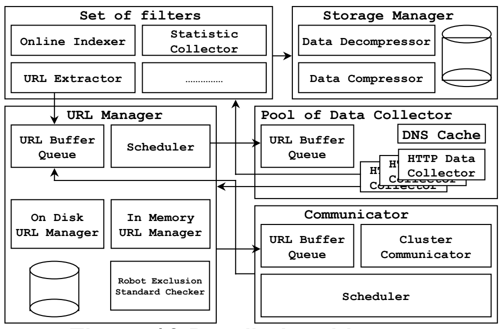

Figure 10 Detailed architecture

3.1 URL Manager

The URL Manager keeps track of all the URL set of that node. The incoming URLs are queue in the URL_Buffer_Queue, and then they are filtered using a regular expression and the robot exclusion standard. The scheduler on the URL_Manager selects and schedules the URLs to fetch by sending the list of URLs to the pool of data collector.

The scheduler chooses the URL to fetch from the active URL set. The URLs of the same web server are packed together in one list, but a list will have no more than 20 URLs. By packing the list of URLs, it enables the data collector to use HTTP1.1's persistent connection and pipelining to gain more performance [1].

3.2 Pool of data collector

The pool of data collector queue the URL list sent from the URL Manager. It has many collector threads to fetch the data from the web servers. Collector threads get a list of URL from the queue and send the request to the web server using HTTP/1.1 persistent connection/pipelining. The fetched data will be passed to the set of filters.

3.3 Set of filters

The set of filters sits between the pool of data collector and the storage manager. Any action that needs to be performing on the web data is taken there. The links in the web page are extracted, the statistics are collected, and the data are indexed by an online indexer which is a part of the search engine. After that the data are passed to the storage manager.

3.4 Storage manager

The storage manager's role is compression and decompressions, storing and retrieving the data. The data from the web pages are compressed and then pack together in a large file. We use LZO [4] as the compression library because it is very fast, while the compression ratio is acceptable.

3.5 Communicator

The communicator send/receive new URLs found from the node that found it to the node that responsible to manage it. The URLs are pack together and send out as a UDP packet. The scheduler of the communicator takes care of timeouts and data retransmission.

4. Results and conclusion

Our spider is written entirely in C++, using GNU C++. The operating system is GNU/Linux RedHat 7.2, using Linux kernel version 2.4. The hardware is four sets of AMD Athlon XP 1500+, 786 MBytes of RAM, and six of 35 GB SCSI harddisks. Each node is equipped with Gigabit Ethernet card, and connected together with a Gigabit Ethernet switch. The gigabit switch is connected to the campus network's backbone using two fast Ethernet links.
In our test, about 400,000 URLs seed from ".th" domain were prepared. We have measured the gathering speed of one, two, three, and four machines, with 50 threads and 300 threads each. The results are shown in Figure 11.
The gathering speed increased when we use more threads or using more machines. There are some factors that limit the gathering speed in our test environment. One factor is that the spider's CPU utilization is easily saturated because every thread has to manage a fast stream of web data. Incoming stream of data has to be parsed and compressed, which both required much CPU resource. Actually, the CPU utilization already reaches 100% when the number of threads is closed to 300.

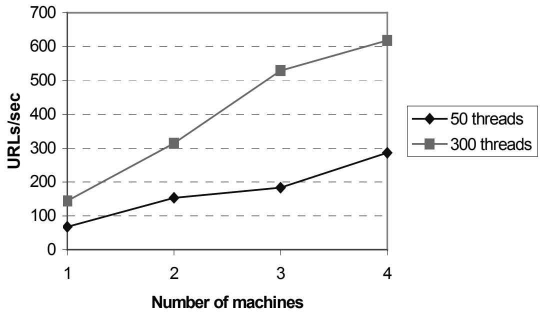

Figure 11 Gathering speed

Our URL compression algorithm yields about 50% reduction in size. The access time is fast for both storing and retrieving. It takes about 92 microsecond by average to compress and store a URL (Min = 10 µs, Max = 400 µ s). Retrieving is much faster. It takes only 6 microsecond by averaged to retrieve a full URL from any given URL's ID. (Min = 2 µs, Max = 188 µs)

5. Conclusion

In this paper we have describes a cluster-based high-performance web spider architecture that is designed to handle a large number of web pages. We have proposed an URL compression techinque, and a phase swapping technique. In our experiments, our spider achieves an average download rate of 618 URLs/sec and 6 MBytes/sec with four machines.

6. References

[1] H. F. Nielsen, J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. W. Lie, C. Lilley, "Network Performance Effects of HTTP/1.1, CSS1, and PNG", In Proc. of ACM SIGCOMM'97, 1997.
[2] K. Bharat, A. Broder, M. Henzinger, P. Kumar, and S. Venkatasubramanian, "The Connectivity Server: fast access to linkage information on the Web", In Proc. of the 7th International World Wide Web Conference, Brisbane, Australia, April 14-18, 1998.
[3] K. Koht-arsa, S. Sanguanpong, "In-memory URL Compression", National Computer Science and Engineering Conference, Chiang Mai, Thailand, November 7-9, 2001, pp. 425-428.
[4] M. F.X.J. Oberhumer, "LZO data compression library", 1996. Available at http://www.oberhumer.com/opensource/lzo/
[5] M. Koster, "A Standard for Robot Exclusion", 1993.
[6] M. Koster, "Guidelines for Robot Writes", 1993. Avaliable at http://www. robotstxt.org /wc/guidelines.html
[7] T. Sterling, D. J. Becker, D. Savarese, J. E. Dorband, U. A. Ranawake, and C. E. Packer, "Beowulf: A Parallel Workstation for Scientific Computation", In Proc. of International Conference on Parallel Processing 95, 1995.