

World Wide Web Crawler

Toshiyuki Takahashi
JST / University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, Japan
tosiyuki@is.s.u-tokyo.ac.jp

Hong Soonsang
University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, Japan
sshong@is.s.u-tokyo.ac.jp

Kenjiro Taura
University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, Japan
tau@logos.t.u-tokyo.ac.jp

Akinori Yonezawa
University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, Japan
yonezawa@is.s.u-tokyo.ac.jp

ABSTRACT

We describe our ongoing work on world wide web crawling, a scalable web crawler architecture that can use resources distributed world-wide. The architecture allows us to use loosely managed compute nodes (PCs connected to the Internet), and may save network bandwidth significantly. In this poster, we discuss why such architecture is necessary, point out difficulties in designing such architecture, and describe our design in progress. We also report on our experimental results that support the potential of world wide web crawling.

Keywords

Crawling, Distributed Computing, P2P

1. INTRODUCTION

Today's search engines move web data from the world to one place. They use a server-client architecture where the central server manages all the status information (URLs visited and to visit) [2,3]. This architecture requires a significant amount of compute and network resources at the crawling site, so only a small number of organizations can afford to operate them. Even with today's state-of-the-art search engines [1], a single round of crawling and indexing takes a month or more to cover a significant portion of web data. As a consequence, they cannot provide up-to-date version of frequently updated pages. To catch up frequent updates without putting a large burden on contents provider, we believe *retrieving and processing data near the data source is inevitable*.

With the distributed nature of web data, it is natural to crawl the web with ordinary PCs already distributed world wide. This is more economical, provided PC users have an incentive to donate their cycles for crawling. More importantly, this may reduce network traffic significantly, provided work is shared

among crawlers according to their locations. However, there are several technical challenges to make this architecture a real alternative to the server-client architecture. This poster discusses issues around this approach and describe solutions we perceive. We also report on our experimental results that suggest the potential of such widely-distributed architecture.

2. OUR WORLD WIDE WEB CRAWLER DESIGN

Technical challenges involved in crawling with world-wide distributed crawlers include:

Scalable management of status data: First and foremost, simply distributing crawlers with the traditional client-server architecture wouldn't improve overall performance at all, because status data (most importantly, the list of visited URLs) would still be on the central server, and it must be consulted for every potentially new URLs. We must design architecture in which the list of visited URLs are also managed in a scalable manner.

Exploiting dynamic and somewhat unreliable resources: We can no longer assume resources are managed by a single organization, nor they are available exclusively for crawling purpose. Those resources may disconnect, shutdown, or even leave the computation permanently. Software must tolerate such dynamic configuration changes, and still distribute work among participating resources in a scalable manner.

We are trying to address these issues by designing a self-organizing network in which work is distributed among nodes that participate at that time [5]. The basic architecture we are currently prototyping is as follows.

URLs are partitioned by their hash values. The range of possible hash values is very large, say $[0, 2^{32})$ or even $[0, 2^{64})$, and this entire range is partitioned among participating crawlers. When a crawler finds a URL in a page, it calculates the hash value of the page and sends it to the node that assumes the value at that time (the home node). The home node checks if the URL has been visited, and if not, it schedules to visit the URL in future. Crawlers that run out of URLs steal ones from other crawlers. This is done by generating a random number within the hash range and sending a request to the crawler that assumes the generated value. When a new crawler joins a crawling session, a participating node splits its assuming hash ranges into two and gives one to the new node. If a crawler permanently leaves a session, it gives its hash range to another node.

The main technical challenge we are working on is the design of an autonomous network that routes messages with a given hash key to the node assuming it. It dynamically adapts to any network configuration (firewall, DHCP, etc.) and changes to node membership and assignments of hash values to nodes.

The design we have described so far addresses the scalability issue, but does not take proximity into account. We envision this can be achieved by occasionally re-arranging the assignment of hash values according to proximity. To be more specific, after crawling for a while, each crawler has a list of hostnames that have been found, and knows its distance (e.g., round-trip time of ping). Two nodes can exchange their hash values so as to improve the average distance to URLs to visit.

3. EXPERIMENTS

3.1 Experimental Server-Client Crawler

We have implemented an experimental crawler based on a traditional client-server system. The purpose is to experiment with network/node performance and to compare our distributed design with the traditional approach. Figure 1 shows the structure of the experimental crawler.

Server node is responsible for managing a database, *URL DB*, of URLs that have been visited and those to visit in future. The controller process on the server takes care of the URL DB. The process responds to the client message requesting a list of URLs to retrieve. The retriever process makes up many connections to web servers simultaneously and downloads contents. Each client has a cache, *Robots DB*, of Robots.txt files of web servers. Retrieved contents are stored in a local disk of the client. The retriever returns two lists, *retrieved URLs* and *found URLs*. The found URLs are the links which have been found in retrieved pages. The controller process in the server receives the lists and register them to the URL DB. It also extracts new URLs which have not been retrieved yet and enqueue them.

Both of the controller and the retriever are written in Python. We are using Berkeley DB [4] for implementing the URL DB and Robots DB.

Our testbed consists of seven workstations located at University of Tokyo and two laptop PCs at home of the first author. Table 1 shows the configuration. Using the testbed we have been running the test crawler for several days. Four retriever processes on each client, 32 in total participated in the crawling. The number of URLs found reached 40 million. 15 million of pages were retrieved, of which 9 million returned Ok status code.

3.2 Server performance limitation

First we tried to investigate how many retrievals can a single server sustain. In our experience, accessing the URL DB which has grown to 40 million entries costs 1.00 msec to query or register a URL. According to the result of the crawling, each page contains 7.00 anchors. Therefore each page on average requires 8.00 DB accesses, one to indicate the completion of this page, and 7.00 to check if each anchor has been visited. This means the single central server cannot sustain more than 125 retrieves per second.

3.3 Client performance

To figure out relationship between the performance and proximity, we have fixed top domain to crawl for 12 hours. First 6 hours of Figure 2 and 3 show the result crawling .jp domain only. Last 6 hours of the figures are that of .com domain. `Client@univ` is one of the results of a client at the university while `Client@home` is that at the home. `Total` is the aggregate of the 8 clients.

Average number of total retrieved pages per second when crawling .jp domain is 42.30 while that when crawling .com domain is 25.07. Even though there are so many extremely broad-band overseas lines now, crawling within the region is faster than that across the borders. This result shows that proximity of the network should be considered to achieve high performance crawling.

The performance just after changing the domain to .com is better than after a while. This is because of implementation of scheduler within the controller process. The scheduler enqueues '.com' hosts within Japan at that time.

It should be noted that '@home' is faster than '@univ'. Average performance of '@home' crawling .jp domain is 55% faster than '@univ', 69% faster when crawling .com. Though there is difference of hardware architecture, it is not related to the performance since 4 retrieve process does not run out machine resources.

The fact that crawling at home is faster may not be true every time. But the results shows there may be possible crawling using combination of PCs at homes would be faster than that using a big cluster at single site.

| | Machine specification | Internet connection |
|-----------------------------|----------------------------------------------------------------------------------------------------------|-----------------------------------------------------------|
| Server + Client@univ * 6 | Sun Blade 1000, (Ultra SPARC III 750MHz * 2, 1GB Memory, IDE HDD, 100Mbps Ether, Sun OS 5.8) | 100Mbps connected to the campus network |
| Client@home * 2 | IBM Thinkpad T23, (Mobile Pentium III 866MHz, 640MB Memory, IDE HDD, 100Mbps Ether, RedHat 7.2) | 1Mbps(upload)+8Mbps(download) ADSL connected to an ISP |

Table 1. Testbed configuration

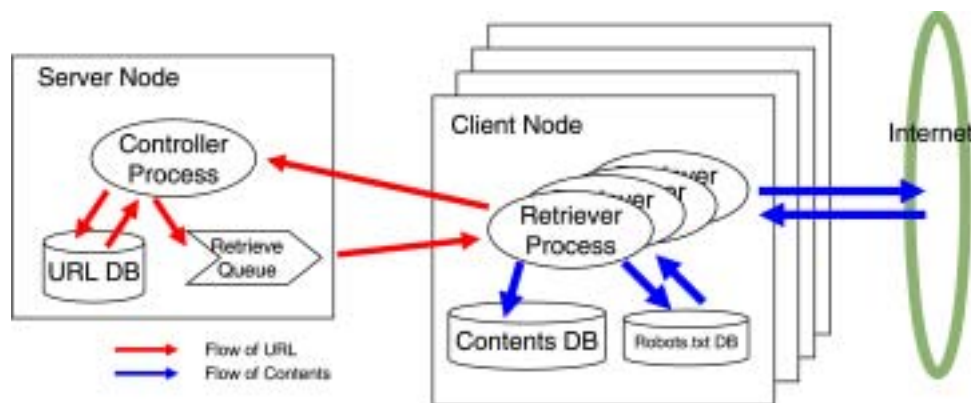


Figure 1. Structure of the test crawler

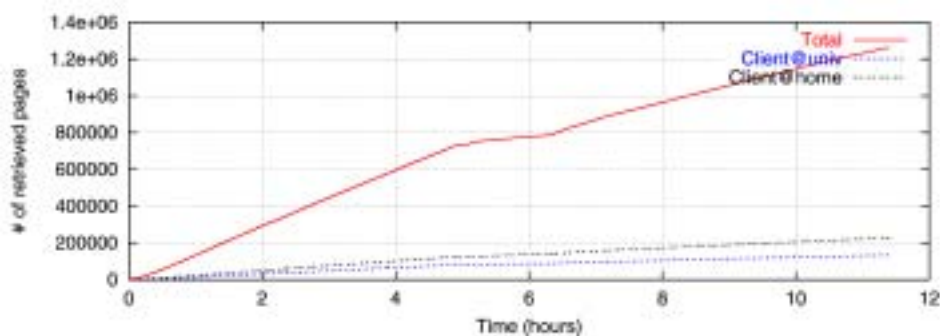


Figure 2. Crawling performance: number of pages retrieved

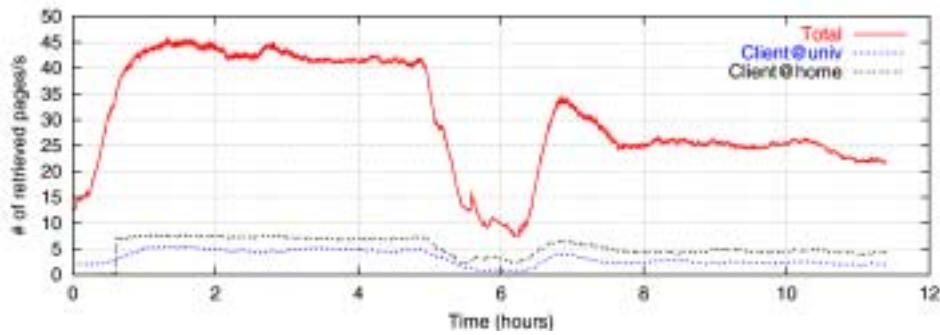


Figure 3. Crawling performance: number of pages retrieved per second

4. Summary

We have discussed issues in making crawling more scalable and proposed a world-wide distributed architecture. We reported on some evidences that motivate designed in this direction. The first is the fact that a simple implementation of a single server can sustain only 125 pages per second due to DB operations, which suggests multiple servers with some scalable mechanisms to balance load among servers. The second is the fact that home machines perform as well as university machines. We sketched our design in progress in which all participating machines share work based on hash values of URLs, and communicate via an automatically reconfigured network.

5. REFERENCES

1. Google, <http://www.google.com/>.
2. M. Najork and A. Heydon. On High-Performance Web Crawling, *SRC Research Report*, Compaq Systems Research Center, forthcoming.
3. Vladislav Shkapenyuk and Torsten Suel. Design and Implementation of a High-Performance Distributed Web Crawler, *ICDE*, 2002.
4. Sleepycat Software Inc. Berkeley DB, <http://www.sleepycat.com/>.
5. Kenjiro Taura, Phoenix: A Parallel Programming Platform Supporting Dynamically Joining/Leaving Resources, *IPSJ SIGNotes High Performance Computing*, No.087-024, 2001, (in Japanese)