Research on Prototype Framework of a Multi-threading Web Crawler for E-commerce

Wenqing Yuan
School of Information Management, WuHan University, WuHan, 430072, China
ywq@zj32.com

Abstract: **Web crawlers facilitate the search engine's work by following the hyperlinks in Web pages to automatically download a partial snapshot of the Web. Crawling is the initial and also the most important step during the web searching procedure. A prototype framework of a multi-threading web crawler for E-commerce application is proposed, in relationship to the former research of search engine. And the design and implementation of a multi-threading web crawler is described and discussed. The experiment result demonstrates this prototype of web crawler has better performance.**

Keywords*: *web crawler, partial snapshot, prototype framework, multi-threading, search engine

I. INTRODUCTION

Crawler (also called Spider, Robots, Worms, Web Wanderers, and Scooters), which is a main component of a search engine, is a program that retrieves Web pages [Pinkerton 1994] or a Web cache . Roughly, a crawler starts off with the URL for an initial page P0. It retrieves P0, extracts any URLs in it, and adds them to a queue of URLs to be scanned. Then the crawler gets URLs from the queue (in some order), and repeats the process.

In an effort to keep up with the tremendous growth of the World Wide Web, many research projects are targeted on how to retrieve and organize information in a way, that will make it easier for the end users to find the information they want efficiently and accurately. A Web Crawler searches through all the Web Servers to find information about a particular topic. However, searching all the Web Servers and the pages, are not realistic, given the growth of the Web and their refresh rates [1]. Crawling the Web quickly and entirely is an expensive, unrealistic goal because of the required hardware and network resources.

So, the design of a good crawler presents many challenges. The most prominent challenge faced by the current web crawlers is to select important pages for downloading [2]. The crawler cannot download all pages from the web. It is important for the crawler to select the pages and to visit "important" pages first by prioritizing the URLs in the queue properly. Other challenges are the proper refreshing strategy, minimizing the load on the websites crawled and parallelization of the crawling process. So, a prototype framework of a multi-threading web crawler for E-commerce application is proposed, in relationship to the former research of search engine. And the design and implementation of a multi-threading web crawler is described and discussed.

The section 2 discusses the related work done so far on this multi-threading crawler model and prototype framework. Section 3 gives a detailed description on the working of a web crawler based on prioritizing strategy. Section 4 deals with some experimental results comparison with other web crawlers.

II. PROTOTYPE FRAMEWORK OF ULTI-THREADING CRAWLER

We now discuss the requirements for a good crawler, and approaches for achieving them. Details on our solutions are the system in a variety of scenarios, with as few modifications as possible.

- Low Cost and High Performance: The system should scale to at least several hundred pages per second and hundreds of millions of pages per run, and should run on low-cost hardware. Note that efficient use of disk access is crucial to maintain a high speed after the main data structures, such as the "URL seen" structure and crawl frontier, become too large for main memory [3]. This will happen after downloading several million pages.

- Robustness: There are several aspects here. First, since the system will interact with millions of servers, it has to tolerate bad HTML, strange server behavior and configurations, and many other odd issues. Secondly, since a crawl may take weeks or months, the system needs to be able to tolerate crashes and network interruptions without losing (too much of) the data. Thus, the state of the system needs to be kept on disk.

- Manageability and Reconfigurability: An appropriate interface is needed to monitor the crawl, including the speed of the crawler, statistics about hosts and pages, and the sizes of the main data sets.

The objective here is to construct a prototype framework, which focus on a multi-threading crawler model, the process of determining the relevancy of the documents before downloading, and the crawling manager including multiple working threads. These works will facilitate our web crawler to satisfying above requirements.

978-1-4244-4639-1/09/$25.00 ©2009 IEEE

A. *Multi-threading crawler model*

A sequential crawling loop spends a large amount of time in which either the CPU is idle (during network/disk access) or the network interface is idle (during CPU operations). Multi-threading, where each thread follows a crawling loop, can provide reasonable speed-up and efficient use of available bandwidth [4]. Figure 1 shows a multi-threading version of the basic crawler. Note that each thread starts by locking the frontier to pick the next URL to crawl. After picking a URL it unlocks the frontier allowing other threads to access it. The frontier is again locked when new URLs are added to it. The locking steps are necessary in order to synchronize the use of the frontier that is now shared among many crawling loops (threads). The model of multi-threading crawler in Figure 1 follows a standard parallel computing model [5]. Note that a typical crawler would also maintain a shared history data structure for a fast lookup of URLs that have been crawled. Hence, in addition to the frontier it would also need to synchronize access to the history.

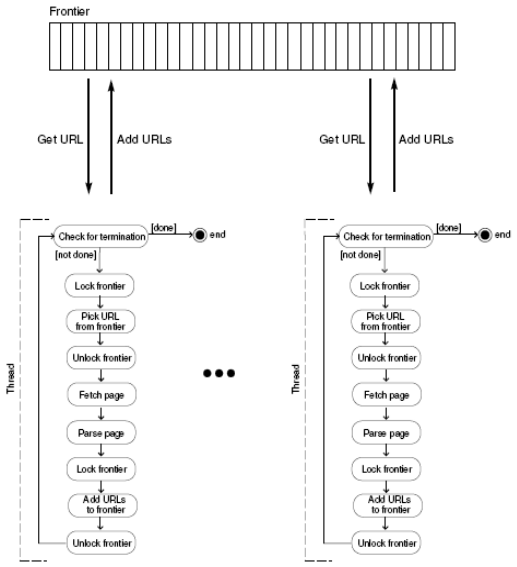

Figure 1. A multi-threading crawler model

The multi-threading crawler model needs to deal with an empty frontier just like a sequential crawler. However, the issue is less simple now. If a thread finds the frontier empty, it does not automatically mean that the crawler as a whole has reached a dead-end. It is possible that other threads are fetching pages and may add new URLs in the near future. One way to deal with the situation is by sending a thread to a sleep state when it sees an empty frontier. When the thread wakes up, it checks again for URLs. A global monitor keeps track of the number of threads currently sleeping. Only when all the threads are in the sleep state does the crawling process stop. More optimizations can be performed on the multi-threading model to decrease contentions between the threads and to streamline network access. Our prototype system is designed and implemented based on these features of multi-threading crawler model.

B. *Prototype framework of multi-threading crawler*

After above crawler model is discussed, this section briefs the overview of the framework and the components of the multi-threading crawler prototype system. Figure 2 shows the functional architecture of the prototype framework.

The prototype collects the Ontology and builds the Ontology Repository. The prototype accepts keyword, on which the crawler has to function. The relevant Ontology is imparted to the Crawler Manager. Meanwhile, the Seed Detector resolves and stores the Seed URLs into the URL Repository. The URLs to be crawled are moved on to the URL Buffer of the Crawler Manager.

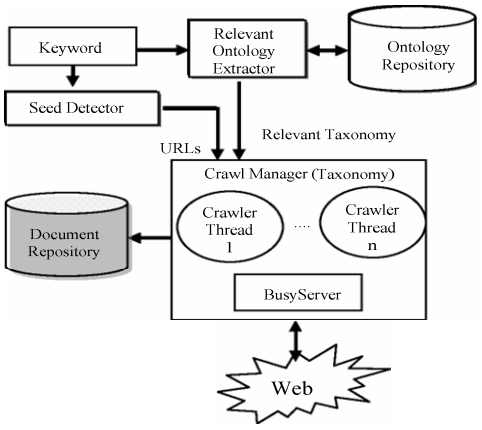

Figure 2. Prototype framework of multi-threading crawler

The Crawler Manager, creates the multi-threading crawler, and assigns the URLs to them. Each URL assigned is to be retrieved from the Global Network. BusyServer is examined to identify whether request has been sent to the same server by the other crawlers. The URL is shipped to the HTTP Module. The HTTP module forwards the request to the corresponding Web Server. Upon receiving the document, the Crawler transports and stores the document in the Document Repository. The Crawler also updates the address in the database - URL Fetched.

Link Extractor parses the document for the hyperlinks and extracts the hypertext and the surrounding text. It also eradicates the Stop Keywords from the hypertext. The hypertext is then passed on to the Hypertext Analyzer. The Hypertext Analyzer checks whether the URL is already fetched by referring the URL Fetched database. The URL along with the priority is stored in the URL Repository for the extraction of the document.

C. *Small multi-threading crawler configuration*

We now focus on the crawling part of above prototype system, further and give a more detailed description of the architecture of our multi-threading crawler. We partition the crawler into two major components - crawling system and crawling application. The crawling system itself consists of several specialized components, in particular a crawl manager, one or more downloaders, and one or more DNS resolvers. The crawl manager is responsible for receiving the URL input stream from the applications and forwarding it to the available downloaders and DNS resolvers while enforcing rules about robot exclusion and crawl speed. A down-loader is a high-performance asynchronous HTTP client capable of

downloading hundreds of web pages in parallel, while a DNS resolver is an optimized stub DNS resolver that forwards queries to local DNS servers. An example of a small configuration with three downloaders is given in Figure 3, which also shows the main data flows through the crawler.

This configuration is very similar to the one we used for our crawls, except that most of the time we used at most 2 downloaders. A configuration as the one in Figure 3 would require between 3 and 5 workstations, and would achieve an estimated peek rate of 250 to 400 HTML pages per second4.

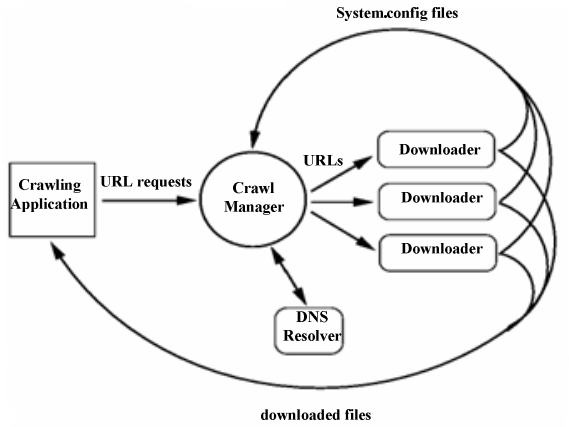

Figure 3. Small multi-threading crawler configuration

III. PROTOTYPE DESIGN OF A MULTI-THREADING CRAWLER WITH PRIORITY

This section will further describe the design of the multi-threading Web crawler. By using pre-emptive multi-threading, you can index a Web page of URL links, start a new thread to follow each new URL link for a new source of URLs to index.

The prototype project uses the MDI CDocument with a custom MDI child frame to display a CEditView when downloading Web pages and a CListView when checking URL links. The project also uses the CObArray, CInternetSession, CHttpConnection, CHttpFile, and CWinThread MFC classes. The CWinThread class is used to produce multiple threads.

The crawler project uses simple worker threads to check URL links or download a Web page. The CSpiderThread class is derived from the CWinThread class, so each CSpiderThread object can use the CwinThread MESSAGE_MAP() function. By declaring a "DECLARE_MESSAGE_MAP()" in the CSpiderThread class the user interface is still responsive to user input. This means you can check the URL links on one Web server and at the same time download and open a Web page from another Web Server. The only time the user interface will become unresponsive to user input is when the thread count exceeds MAXIMUM_WAIT_OBJECTS which is defined as below.

In the constructor for each new CSpiderThread object we supply the ThreadProc function and the thread Parameters to be passed to the ThreadProc function.

```
CSpiderThread* pThread;
pThread = NULL;
pThread=newSpiderThread(CSpiderThread::ThreadFunc,p
ThreadParams);    // create a new CSpiderThread object
```

In the CSpiderThread constructor we set the CWinThread* m_pThread pointer in the thread Parameters structure so we can point to the correct instance of this thread;

```
pThreadParams->m_pThread = this;
The CSpiderThread ThreadProc Function
// simple worker thread Proc function
UINT CSpiderThread::ThreadFunc(LPVOID pParam)
{
   ThreadParams * lpThreadParams = (ThreadParams*)
   pParam;
   CSpiderThread* lpThread = (CSpiderThread*)
   lpThreadParams->m_pThread;
   lpThread->ThreadRun(lpThreadParams);          /* Use
   SendMessage instead of PostMessage here to keep the
   current thread count Synchronizied. If the number of
   threads is greater than MAXIMUM_WAIT_OBJECTS
   (64) the program will become unresponsive to user
   input */
   ::SendMessagelpThreadParams->m_hwndNotifyProgres
   s,WM_USER_THREAD_DONE,0,(  (LPARAM)lpThre
   adParams);
   //deletelpThreadParams and decrements the thread count
   return 0;
}
```

The structure passed to the CSpiderThread ThreadProc Function.

```
typedef struct tagThreadParams
{
    HWND m_hwndNotifyProgress;
    HWND m_hwndNotifyView;
    CWinThread* m_pThread;
    CString m_pszURL;
    CString m_Contents;
    CString m_strServerName;
    CString m_strObject;
    CString m_checkURLName;
    CString m_string;
    DWORD m_dwServiceType;
    DWORD m_threadID;
    DWORD m_Status;
    URLStatus m_pStatus;
    INTERNET_PORT m_nPort;
    INT m_type;
    BOOL m_RootLinks;
} ThreadParams;
```

After the CSpiderThread object has been created we use the CreatThread function to start the execution of the new thread object.

```
if(!pThread->CreateThread())//Starts execution of a
CWinThread object
{
   AfxMessageBox ("Cannot Start New Thread");
   delete pThread;
```

```
    pThread = NULL;
    delete pThreadParams;
    return FALSE;
  }
```

Once the new thread is running we use the ::SendMessage function to send messages to the CDocument's-> CListView with the status structure of the URL link.

Each new thread creates a new CmyInternetSession (derived from CInternetSession) object with EnableStatusCallback set to TRUE, so we can check the status on all InternetSession callbacks. The dwContext ID for callbacks is set to the thread ID.

The key to use the MFC WinInet classes in a single or multithread program is to use a try and catch block statement surrounding all MFC WinInet class functions. The internet is very unstable at times or the web page you are requesting no longer exist, which is guaranteed to throw a CInternetException Error.

```
try
  {
    // some MFC WinInet class function
  }
catch (CInternetException* pEx)
  {
    // catch errors from WinINet
    //pEx->ReportError ();
    pEx->Delete();
    return FALSE ;
}
```

The maximum count of threads is initially set to 64, but you can configure it to any number between 1 and 100. A number that is too high will result in failed connections, which means you will have to recheck the URL links.

When the SPIDER program checks URL links its goal is to not request too many documents too quickly. A rapid fire succession of HTTP requests could bring a server to its knees. The SPIDER program adheres somewhat to the standard for robot exclusion. This standard is a joint agreement between robot developers that allows WWW sites to limit what URL's the robot requests. A server can prevent any directory from being indexed by creating a robots.txt file for that directory. By using the standard to limit access, the robot will not retrieve any documents that Web Server's wish to disallow.

Before checking the Root URL, the program will check if there is a robots.txt file in the main directory. If the SPIDER program finds a robots.txt file the program will abort the search. The program also checks for the META tag in all Web pages. If it finds a META NAME="ROBOTS" CONTENT ="NOINDEX, NOFOLLOW" tag it will not index the URLs on that page.

```
//search for robot.txt
pFile = pServer->OpenRequest (_T("GET"),"/robots.txt");
pFile->SendRequest ();
```

Some other codes are not listed because of space limitation.

IV. PRECISION COMPARISON OF WEB CRAWLERS

After prototype system is established, we consider crawling precision as performance metrics of web crawlers. Precision is defined as the proportion of retrieved and relevant web pages to all the web pages retrieved. In this section, we compare the precision among different web crawlers and also inspect the improvements due to individual heuristics. The Multi-threading web crawler, context graph crawler [6] and CINDI Robot are employed as comparisons.

For all web crawlers, we use the same initial seed URLs which contains only 15 randomly selected web sites from Seed Finder. A small initial URL set ensures the performances of all web crawlers are compared fairly since for the Multi-threading Crawler the precision is maintained during a really long crawling session due to the higher seed URL quality achieved by the Seed Finder.

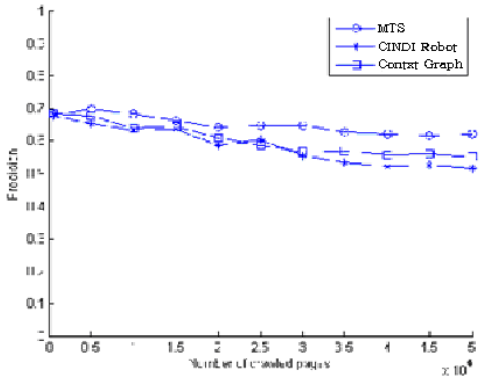

Figure 4. The precision of multi-threading web crawler

Figure 4 gives the precision of the crawling process for different web crawlers, where "MTS" denotes Multi-Threading Search. We can observe that the Multi-threading Crawler outperforms other web crawlers.

V. CONCLUSION AND FUTURE WORK

We have described the architecture and implementation details of our crawling prototype, and presented some preliminary experiments. There are obviously many improvements to the system that can be made. A major open issue for future work is a detailed study of the scalability of the system and the behavior of its components. This could probably be best done by setting up a simulation testbed, consisting of several workstations, that simulates the web using either artificially generated pages or a stored partial snapshot of the web. We are currently considering this, and are also looking at testbeds for other high-performance networked systems.

Our main interest is in using the crawler in our research group to look at other challenges in web search technology, and several students are using the system and acquired data in different ways.

ACKNOWLEDGMENT

We would like to acknowledge our team for their works for

this paper. This research is supported by the AOE Important Project of Philosophy and Social Science. The Project Number is 05JJD870158. It is also supported by NSFC under Grant 70473068.

REFERENCES

[1] S. Chakrabarti, M. van den Berg, and B. Dom. Focused crawling: a new approach to topic-specific Web resource discovery. Computer Networks (Amsterdam, Netherlands: 1999), 31(11–16):1623–1640, 1999.

[2] Monika R.Henzinger, "Algorithmic Challenges in Web Search Engines", Internet Mathematics Volume 1, pages 115— 126, December 2002.

[3] Z. Bar-Yossef, A. Berg, S. Chien, J. Fakcharoenphol, and D. Weitz. Approximating aggregate queries about web pages via random walks. In Proc. of 26th Int. Conf. on Very Large Data Bases, September 2000.

[4] T. I. Hiroshi Takeno. Distributed information gathering and full text search system infobee / evangelist. NTTR&D, February. (in Japanese).

[5] V. Kumar, A. Grama, A. Gupta, G. Karypis. Introduction to Parallel Computing: Design and Analysis of Algorithms[M]. U.S.A:Benjamin/Cummings, 1994:112-113

[6] M.Diligenti et al, "Focused Crawling using Context Graphs", 26th International Conference on Very Large Databases, VLDB 2000, pp.527-534.