Georgia State University
ScholarWorks @ Georgia State University

Computer Science Theses | Department of Computer Science

8-3-2007

A Distributed Approach to Crawl Domain Specific Hidden Web

Lovekeshkumar Desai

Follow this and additional works at: http://scholarworks.gsu.edu/cs_theses

Recommended Citation
Desai, Lovekeshkumar, "A Distributed Approach to Crawl Domain Specific Hidden Web." Thesis, Georgia State University, 2007.
http://scholarworks.gsu.edu/cs_theses/47

This Thesis is brought to you for free and open access by the Department of Computer Science at ScholarWorks @ Georgia State University. It has been accepted for inclusion in Computer Science Theses by an authorized administrator of ScholarWorks @ Georgia State University. For more information, please contact scholarworks@gsu.edu.

A DISTRIBUTED APPROACH TO CRAWL DOMAIN SPECIFIC HIDDEN WEB

by

LOVEKESHKUMAR DESAI

Under the Direction of Rajshekhar Sunderraman

ABSTRACT

A large amount of on-line information resides on the invisible web – web pages generated dynamically from databases and other data sources hidden from current crawlers which retrieve content only from the publicly indexable Web. Specially, they ignore the tremendous amount of high quality content “hidden” behind search forms, and pages that require authorization or prior registration in large searchable electronic databases. To extracting data from the hidden web, it is necessary to find the search forms and fill them with appropriate information to retrieve maximum relevant information. To fulfill the complex challenges that arise when attempting to search hidden web i.e. lots of analysis of search forms as well as retrieved information also, it becomes eminent to design and implement a distributed web crawler that runs on a network of workstations to extract data from hidden web. We describe the software architecture of the distributed and scalable system and also present a number of novel techniques that went into its design and implementation to extract maximum relevant data from hidden web for achieving high performance.

INDEX WORDS: Distributed Web crawler, Search spider, Breadth-first crawler, Deep Web, Hidden Web, Content Extraction, task-specific and Domain Specific.

A DISTRIBUTED APPROACH TO CRAWL DOMAIN SPECIFIC HIDDEN WEB

by

LOVEKESHKUMAR DESAI

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science

In the College of Arts and Sciences

Georgia State University

2007

Copyright by

Lovekeshkumar Desai

2007

A DISTRIBUTED APPROACH TO CRAWL DOMAIN SPECIFIC HIDDEN WEB

By

LOVEKESHKUMAR DESAI

Major Professor: Rajshekhar Sunderraman

Committee: Anu Bourgeois

Saeid Belkasim

Electronic Version Approved:

Office of Graduate Studies

College of Arts and Sciences

Georgia State University

August 2007

Dedicated to everyone who was a part of this
For all the support

ACKNOWLEDGEMENTS

I am very much thankful to my advisor, Professor Dr. Rajshekhar Sunderraman for his constant support and encouragement without which I could not have accomplished this thesis. I am thankful for his interest in this new field and allowing me to work in it. He has constantly supervised guided and suggested ideas for improving my work and led it to a new dimension.

I am also thankful to my committee members, Professor Dr. Saeid Belkasim and Professor Dr. Anu Bourgeois for their valuable encouragement and support. I am thankful for their innovative ideas and their interest in new technologies which motivated me to go forward in this prototype.

I would also like to thank my parents for their constant love and support which made this possible.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS…………………………………………………………… v

LIST OF TABLES……………………………………………………………. ix

LIST OF FIGURES…………………………………………………………... x

LIST OF ABBREVIATIONS…………………………………………………. xi

1. INTRODUCTION………………………………………………………….. 1

1.1. Motivation…………………………………………………………………. 2

1.2. Challenges……………………………………………………………….. 3

1.3. Related Work……………………………………………………………... 3

1.4. Need for Distributed Solution……..………………………………………. 4

1.5. Organization of Thesis…………………………………………………... 5

2. EXISTING WEB CRAWLERS AND RELATED WORK……………………. 6

2.1. The Design of the PIW crawler…………………………………………. 6

2.1.1. The Search Engine………………………………………………… 8

2.1.2. Indexing mode……………………………………………………… 9

2.1.3. Agents………………………………………………………………. 10

2.2. Overview of Hidden-Web crawler………………………………………. 11

2.3. Comparing the basic approach for PIW crawler and HW crawler….. 12

3. SYSTEM ARCHITECTURE & DESIGN……………………………………. 14

3.1. Crawl Manager…………………………………………………………… 16

3.2. Crawling Application…………………………………………………….. 16

3.2.1. Breadth-first Algorithm…………………………………………….. 17

3.2.2. Robot Protocol……………………………………………………... 18

3.3. Crawl System (Content Extraction)………………………………….. 20

3.3.1. The Basic Structure……………………………………………….. 20

3.3.2. Analyzer………………………………………………………….. 20

3.3.3. Parser……………………………………………………………. 21

3.3.4. Composer………………………………………………………... 22

3.3.5. Result Analyzer…………………………………………………. 23

3.4. Scaling the System…………………………………………………. 23

3.5. Limitations and Assumptions……………………………………… 26

4. DESIGN MODULES: IMPLEMENTATION DETAILS……………………… 27

4.1. Crawling Application……………………………………………….. 27

4.1.1. Crawl Function………………………………………………….. 27

4.1.2. retrieveLinks Function………………………………………….. 30

4.1.3. isRobotAllowed Function……………………………………….. 30

4.2. Crawl Manager……………………………………………………… 31

4.3. Crawl System (Content Extraction)……………………………….. 32

4.3.1. Analyzer………………………………………………………….. 32

4.3.2. Parser and Composer…………………………………………... 34

4.3.3. Result Analyser…………………………………………………. 39

5. OBSERVATIONS AND EXPERIMENTAL RESULTS……………………… 43

5.1. Analyser: //table or //table//table regular expression…………………. 44

5.2. Good and Bad URLs……………………………………………….. 45

5.3. Measurements of execution time…………………………………… 50

6. FUTURE WORKS………………………………………………………… 52

6.1. Knowledge base…………………………………………………………… 53

6.2. Open Databases…………………………………………………………… 54

6.3. Image Search Crawler…………………………………………………... 55

7. CONCLUSION…………………………………………………………………... 56

REFERENCES……………………………………………………………………….. 57

APPENDIX – A……………………………………………………………………….. 59

APPENDIX – B……………………………………………………………………….. 69

APPENDIX – C………………………………………………………………………. 78

APPENDIX – D………………………………………………………………………. 80

APPENDIX – HtmlUnit……………………………………………………………. 82

LIST OF TABLES

Table 1. Search results with //table//table regular expression……………….. 43

Table 2. Search results with // table regular expression……………………… 44

Table 3. Execution time results………………………………………………….. 51

LIST OF FIGURES

Figure 1: Software Components of the PIW Crawler…………………………. 7

Figure 2: The Web as a graph…………………………………………………… 9

Figure 3: Interacting with Forms………………………………………………… 11

Figure 4: Comparing the basic approach of a PIW crawler and HW crawler. 12

Figure 5: Architecture – Distributed HW crawler configuration………………. 15

Figure 6: Large Configuration for HW crawler…………………………………. 24

Figure 7: Future prototype of HW crawler……………………………………... 52

LIST OF ABBREVIATIONS

HW	Hidden Web
PIW	Publicly Indexable Web

1. INTRODUCTION

The World Wide Web has grown from a few thousand pages in 1993 to more than two billion pages at present. This source of information is available in various forms; Websites, databases, images, sound, videos and many more. Web based information is usually organized and is available in an ad hoc manner. Due to its vastness, web search engines are becoming increasingly important as the primary means of locating relevant information.

Searching for information is a primary activity on the Web and about 80% of the Web users use search engines to retrieve information from the Web [2]. General-purpose search engines such as Google and Yahoo! rely on massive collections of web pages that are acquired with the help of web crawlers, which traverse the web by following hyperlinks and storing downloaded pages in a large database to be used to build search indexes. Powerful search engines such as Google which can crawl and index millions of Web pages everyday, provide a good start for information retrieval but may not be sufficient for complex information inquiry tasks that require not just an initial search, but then relevant classification of a large volume of results. In the past years, domain-specific search engines have however become equally popular, as they offer a higher precision for someone looking for a information specific to a domain of human activity. CiteSeer and Medline sites for searching publications in computer science and medicine are two well-known examples of such domain-specific services. One area where the search engines lack their expertise is in classifying

and indexing the information available from the databases which requires authorization or prior registration or querying interface to retrieve information.

1.1 Motivation

Current-day search and categorization services cover only a portion of the Web called the publicly indexable Web (PIW)[13]. This refers to the set of web pages reachable purely by following hypertext links, ignoring search forms and pages that require authorization or prior registration. Consider an example of a shopping Websites. According to internet studies, shopping is one wide part of internet that covers of almost 35 % of it [1]. Now, thinking as a user, if one wants to shop on internet for a particular product, the best way to begin is to start browsing the Websites popular in shopping for e.g. amazon.com, ebay.com, bestbuy.com and many more. More likely a shopper will surf for like 3 different shopping Websites before he concludes as to where to buy. Now imagine why not to use the search engines to shop. Each time a buyer has to visit at least 3 different shopping Websites, making searches and gathering all the information he needs. Observing closely, the buyer is manually doing the work of the WebCrawler, which is visiting the shopping Websites to gather necessary information and then picking up the right one for him. General purpose search engines crawls only what we call as surface Web. Whenever any WebCrawler visits a Web server, they gather information of all the Web pages been stored on that Web server. They are not able to penetrate deep into the Web server to access their databases and use the Web services served by that particular Web server to retrieve information. According to studies conducted in 2000, deep Web

contains 7500 terabytes of information in comparison to 19 terabytes on surface Web. There are more than 200,000 deep Web sites [3]. There is a need for Hidden Web (HW) crawler which can crawl and extract this vast source of information – hidden web.

1.2 Challenges

There are significant technical challenges in designing a hidden Web crawler. First, the crawler must be designed to automatically parse, process, and interact with form-based search interfaces that are designed primarily for human consumption. Second, unlike PIW crawlers which merely submit requests for URLs, hidden Web crawlers must also provide input in the form of search queries (i.e., “fill out forms”). This raises the issue of how best to equip crawlers with the necessary input values for use in constructing search queries. Hidden web crawler also needs result analyzer to analyze the results obtained because of queries to ensure relevance of information. There are two steps in building such HW crawler: *resource discovery*, wherein we identify sites and databases that are likely to be relevant to the task; and *content extraction*, where the crawler actually visits the identified sites to submit queries and extract hidden pages.

1.3 Related Work

This work is an extension of basic prototype on the same topic done before [23]. In the previous prototype, basic approach for the hidden web content extraction was implemented. The previous prototype was a stand-alone system, which used to take the list of URLs already collected manually in URL file for processing. As the previous system used to take long time usually in hours to

process a list of hundred URLs, it became eminent to develop a more efficient distributed system. In this work there are also many improvements in the content extraction part to enhance the performance and to extract more data with higher percentage of relevance.

1.4 Distributed domain-specific solution.

HW crawler aim to selectively crawl portions of the hidden Web, extracting content based on the requirements of a particular application or task. For example, consider a user who is interested in buying a car at particular location and he wants to find the best available deal. For such HW crawler human-assistance is critical to ensure that the crawler issues queries that are relevant to the particular task. For instance, in the above example, the buyer may provide the make (company) and zip of the car with other information that are of interest. This enables the crawler to use these values when filling out forms that require a make of the car and zip to be provided. The proposed HW crawler requires lots of analysis of each web page to understand the query interface and it also requires complex processing of all the result pages to assure correctness of extracted information. To fulfill all these complex challenges that arise when attempting to search hidden web it becomes eminent to design and implement a distributed web crawler that runs on a network of workstations to make entire application efficient to process large number of web pages.

The listed below in brief are the few contributions made by the work:

1. *Resource Discovery:* First HW crawler needs to create list of URL using some initial domain specific starting URLs. Second step is to download

each web page and identify Web pages that serve as entry points into the Hidden Web.

2. *Content Extraction:* HW crawler need to understand the query interfaces and the result pages of those Hidden Web entry points, in order to find the domain specific relevant information and reuse in an information integration framework.

1.5 Organization of thesis

The following chapters of this report will propose a prototype of a distributed HW crawler to access hidden databases, which does not require prior authorization for the querying interfaces and gets the most information from it. Chapter 2 talks about the background and related works and few models of the WebCrawler made so far. Chapter 2 gives an overview of "The Hidden Web" and about the technical challenges to extract data from hidden web. It also includes the differences in basic approach of general web crawler with that of Hidden web crawler. In chapter 3, we discuss about the architecture of the proposed distributed HW Crawler. Chapter 4 presents various aspects of the HW crawler's modules and working of it. It also explains scalability of the proposed distributed architecture. Chapter 5 covers the results obtained from the proposed prototype and also explains the results with observations. Chapter 6 covers the future proposed architecture for the system and chapter 7 concludes the paper.

2. BACKGROUD

Here we briefly discuss the PIW crawlers which will help to understand the technical challenges of HW crawler. The development of WebCrawler began in January 27, 1994 by Brain Pinkerton, a student at University of Washington. From 1994 to present, the WebCrawler has undergone through most significant and radical changes. The initial WebCrawler's were not very good in speed and in their precision. With passing of time, better search algorithms were used that helped crawlers to crawl more speedily and with higher precision. We will now take a look at the design of the basic WebCrawler for the surface web.

2.1. The Design of the PIW Crawler

The World Wide Web is decentralized by nature. Anybody can add Web pages, servers, documents and hypertext links. In WebCrawler's, discovering new Web pages is an important feature due to the dynamic organization of the Web. For any WebCrawler, discovering Web pages means to identify Uniform Resource Locators (URLs). These URLs point to different network resources. Once the WebCrawler has that URL, it can decide weather to search the document and retrieve it or not. Once the document is been retrieved, the WebCrawler will place the URL into the retrieved URL database. Each and every single time, when WebCrawler comes across any URL, it performs three basic steps.

In the first step, it checks the URL with the retrieved URL database to see if the entry is there or not. If the entry is already in the retrieved URL database, then it means that the WebCrawler has already visited that page and hence no

need to retrieve it again. After placing the entry in the retrieved URL database, in the second step the WebCrawler will then parse the Web page for any outbound and inbound links. These outbound links are check and stored in retrieved URL database later on, while the inbound links are stored in a separate small hash table designed for that particular Website. After the Web page is parsed for the outbound and inbound links, during the third step the Web page is parsed and the information collected is indexed into the main database.

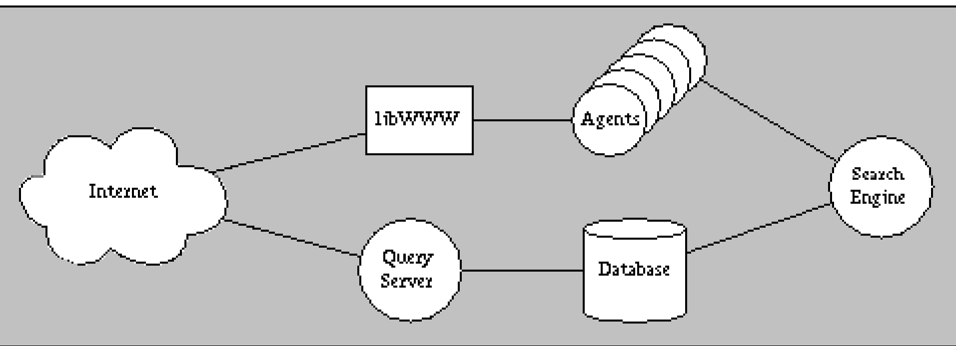

Figure 1: Software components of the PIW Crawler

For each of the three steps discussed above, the WebCrawler has a particular software component designed. The connections between the components are as shown in Figure 1. In the architecture, as shown above, the search engine controls the progress of WebCrawler. The search engine will decide which document should be explored first for the initiating the information retrieval. Normally, every search engine begins with a preset amount of Web links, formed in the Web directory [7]. The database handles the persistent storage of the Web page metadata, the links between them, and the full-text index. The "agents" are responsible for retrieving the Web pages from the network at the direction of the search engine. Finally, the Query Server

implements the query service provided to the Internet. Each of these components is described in detail below.

2.1.1. The Search Engine

The WebCrawler discovers new Web pages by starting with a known set of Web pages available from Web directory. Examining the outbound links from them, the WebCrawler will follow one of the links that leads to a new Web page, and then repeating the whole process. Another way to think of it is that the Web is a large directed graph and that the WebCrawler is simply exploring the graph using a graph traversal algorithm. Figure 2 shows an example of the Web as a graph. Imagine that the WebCrawler has already visited Web page A on Server 1 and Web page E on Server 3 and is now deciding which new Web pages to visit. Web page A has links to Web page B, C and E, while Web page E has links to Web page's D and F. The WebCrawler will select one of Web pages B, C or D to visit next, based on the details of the search it is executing. The search engine is responsible not only for determining which Web pages to visit, but which types of Web pages to visit. The file which WebCrawler is unable to index, such as pictures, sound, PostScript or binary data is not retrieved from the Web. If they are erroneously retrieved, they are ignored during the indexing step. This file type discrimination is applied during both kinds of searching. The only difference between running the WebCrawler in indexing mode and running it in real-time search mode is the discovery strategy employed by the search engine.

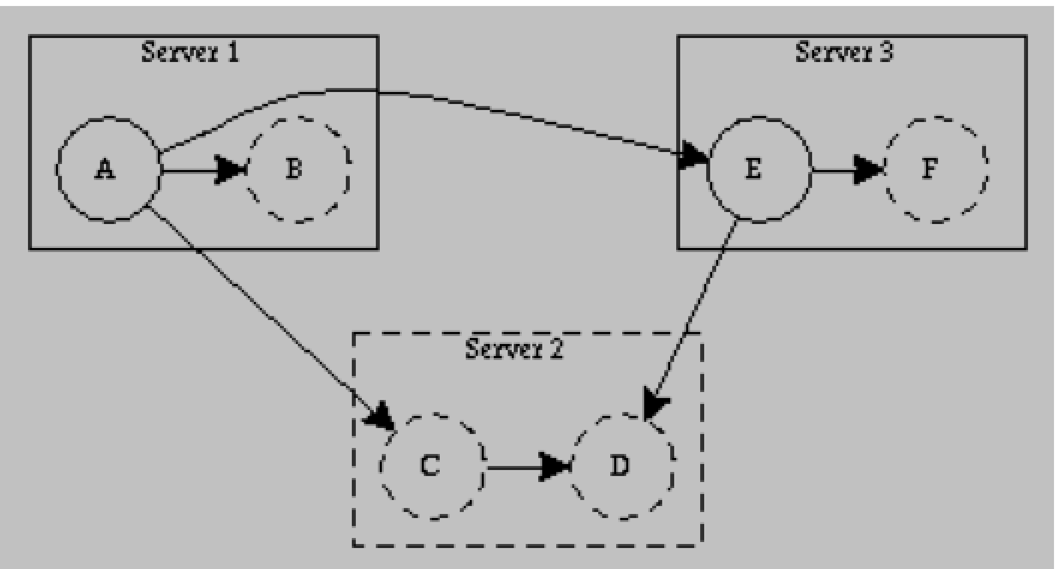

Figure 2: The Web as a graph

2.1.2. Indexing mode

The ultimate goal of any search engine is to build databases of larger indexes. If the WebCrawler index has enough space for 50,000 Web pages, then those Web pages should be more relevant ones. For a Web index, one solution is that those Web pages should come from as many different servers as possible. The WebCrawler takes the following approach: it uses a modified breadth-first algorithm to ensure that every server has at least one Web page represented in the index. This strategy is very effective. The most frequent feedback about the WebCrawler is that it has great coverage and that nearly every server is represented.

In detail, a WebCrawler indexing run proceeds as follows: every time a Web page on a new server is found, that server is placed on a list of servers to be visited right away. Before any other Web pages are visited, a Web page on each of the new servers is retrieved and indexed. When all known servers have

been visited, indexing proceeds sequentially through a list of all servers until a new one is found, at which point the process repeats. During indexing, the WebCrawler runs either for a certain amount of time, or until it has retrieved some number of Web pages. Normally, the WebCrawler can build an index at the rate of about 1000 Web pages an hour on a 486-based PC running NEXTSTEP [8]. The basic idea of following different Web pages from one to another using the links came was first demonstrated to work in the Fish search [10]. The WebCrawler extends that concept to initiate the search using the index, and to follow links in an intelligent order.

2.1.3. Agents

In order to actually retrieve Web pages, the search engines will invoke “agents”. The task of the agent is simple, to retrieve the URL; in response to the task assigned, the agent will either returned the whole Web page or a reason why the Web page could not be retrieved. The agent uses the CERN WWW library [9], which gives it the ability to access several types of content with several different protocols, including HTTP, FTP and Gopher.

These agents are all ran as separate process. Since there might be few issues like server been down or the network having a bottleneck. A typical WebCrawler contains like 15 agents in parallel. The search engine decides a new URL, finds a free agent and then assigns that URL to the free agent. When that agent responds back, it gets new URL. As a practical matter, running agents in separate processes helps isolate the main WebCrawler process from memory leaks and errors in the agent.

2.2 Overview of Hidden-Web Crawler

As described earlier the basic approach to crawl surface web is to follow different Web pages from one to another using the links. Where as crawling the hidden web requires completely different approach, which is more complex and requires lots of analysis. The fundamental difference between the actions of a hidden Web crawler and that of a traditional PIW crawler is with respect to pages containing search forms.

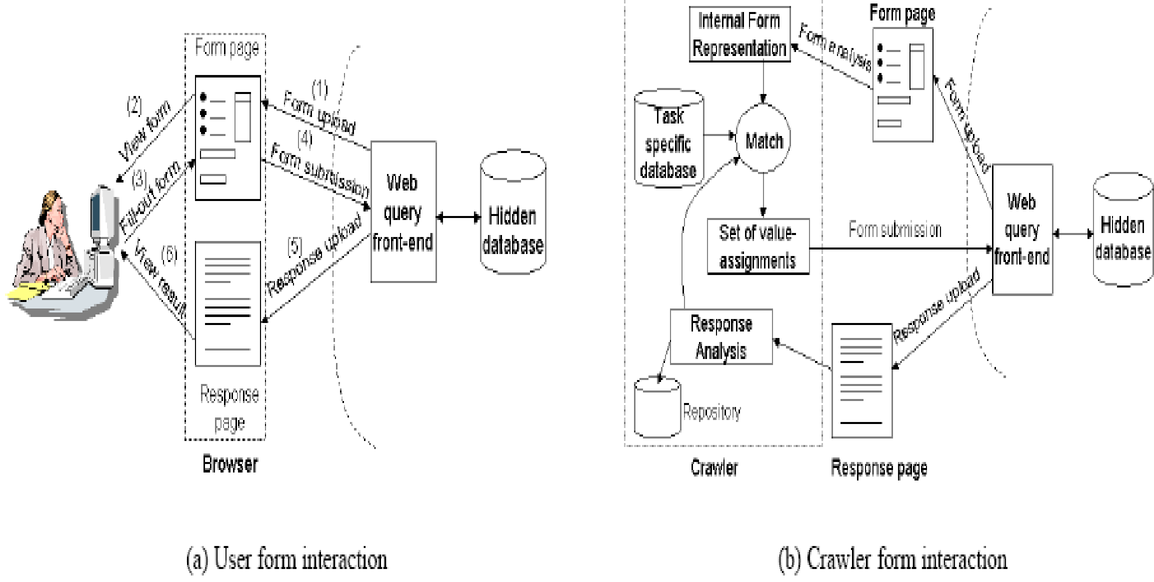

Figure 3: Interacting with Forms [14]

Figure 1(a) illustrates the sequence of steps (as indicated by the numbers above each arrow) that take place, when a *user* uses a search form to submit queries on a hidden database. Figure 1(b) illustrates the same interaction, with the *crawler* now playing the role of the human browser combination. Our prototype of distributed hidden web crawler consists of the four components namely analyzer, parser, composer and result analyzer described in the next section. We shall use

the term *form page*, to denote the page containing a search form, and *response page*, to denote the page received in response to a form submission.

2.3. Comparing the basic approach for PIW crawler and HW crawler.

In the traditional PIW crawler the basic idea is to follow different web pages from one to another using links. Another way to think it is that the Web is a large directed graph and that the PIW Crawler is simply exploring the graph using a graph traversal algorithm.

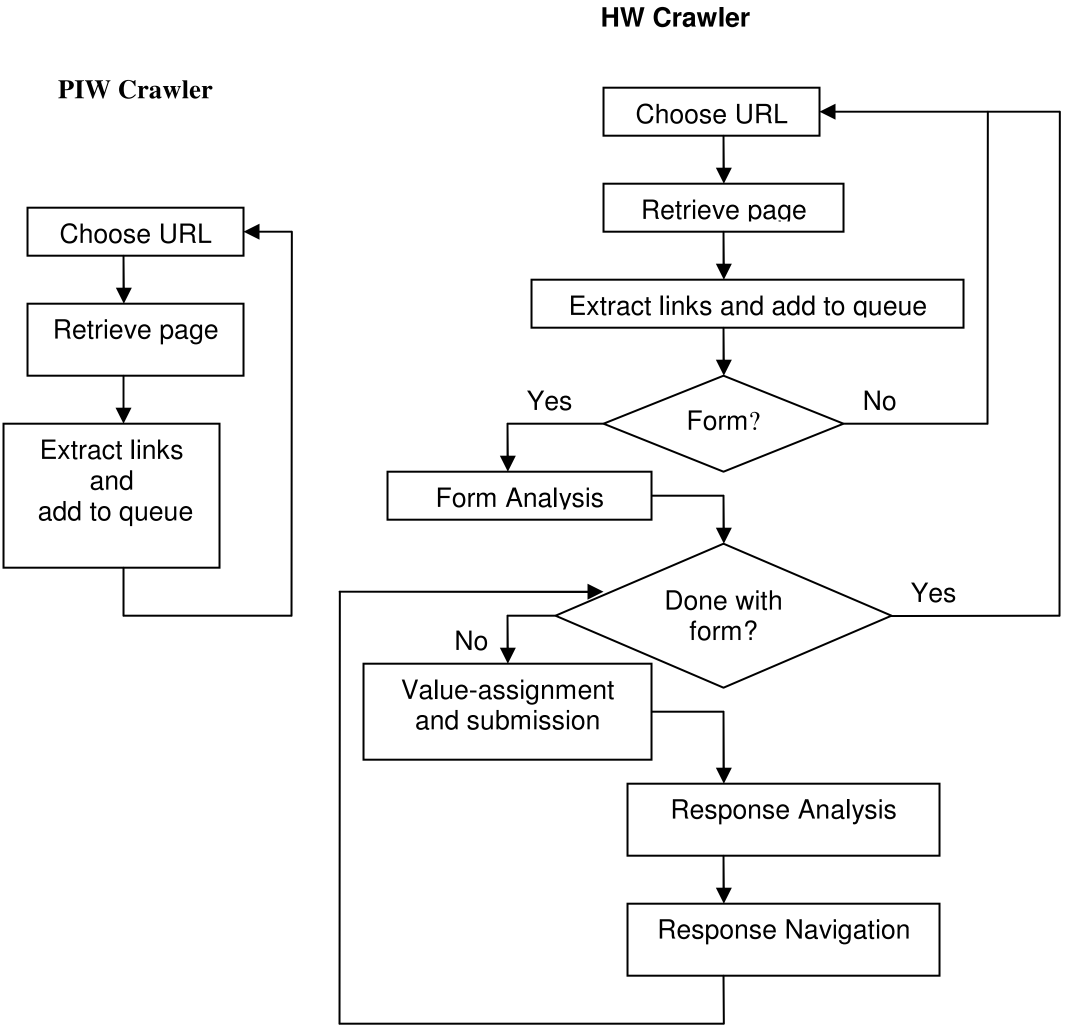

Figure 4: Comparing the basic approach of a PIW crawler and HW crawler.

The execution loop for the HW crawler is much more complex than PIW

crawler. HW crawler starts same as PIW crawler by downloading the required web page, but then after it requires lots of analysis and intelligence to extract information from the hidden web. First it starts with analysis of the downloaded page, whether it is a correct search form to form query. All such search forms found by analyzer are then parsed by parser to extract the required fields to be filled by the composer. Finally after querying the database, results are obtained by the crawler. Now all the results obtained by crawler must be analyzed to get the required information and to make sure that the information is relevant to required search results. We will discuss all these processes in detail in the next section.

3. SYSTEM ARCHITECTURE & DESIGN

As mentioned earlier there are two steps in HW crawler: *resource discovery*, wherein we identify sites and databases that are likely to be relevant to the task or domain; and *content extraction*, where the crawler actually visits the identified sites to submit queries and extract hidden pages. In our prototype of HW crawler we have used the terms crawling application for resource discovery component and crawling system for content extraction.

The crawling application crawls the web pages already stored in the repository or in other words starting URLS to create the list of links in those web pages. Crawling application takes care of task such as robot exclusion, which algorithm to use for crawling, assuring each new link page having search string. Crawling application stores files of URLS in the repository, which are later downloaded by the crawling system for content extraction. The basic structure of HW crawler is entirely implemented in the Crawling system component, which can run on number of machines to increase the efficiency and reduced execution time. An example of a small configuration with three crawling system is given in Figure 5(a), which also shows the main data flows through the system. This configuration is very similar to the one we used for over prototype, except that most of the time we used at most 2 crawling systems. A configuration as the one in Figure 5(a) would require 4 workstations. We discuss scaling in more detail further below.

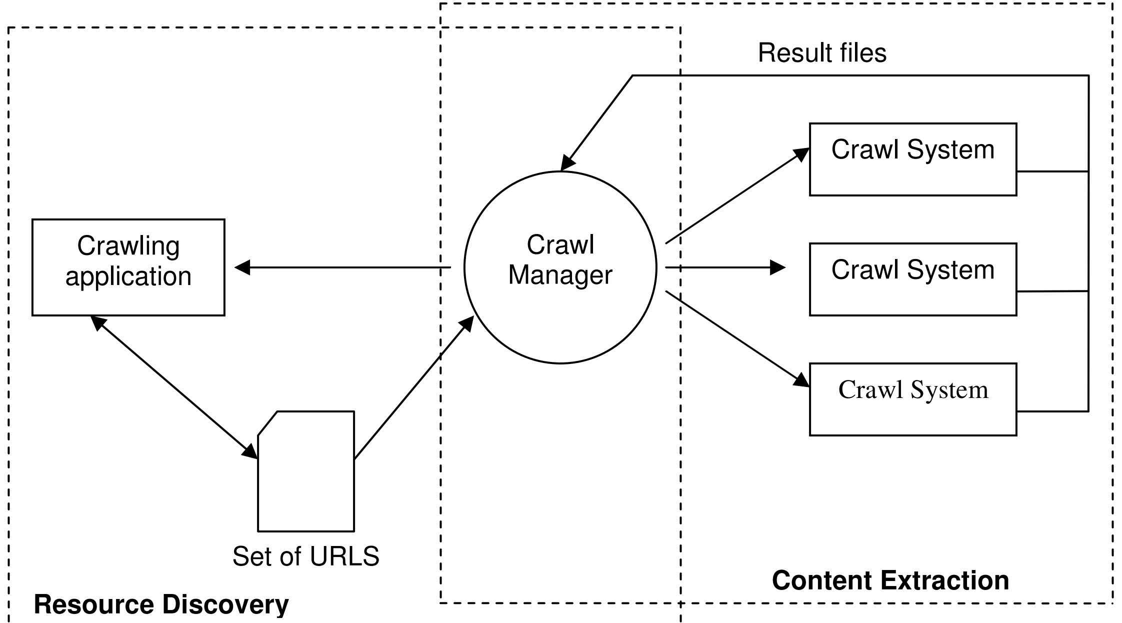

Figure 5(a): Distributed web Crawler Configuration

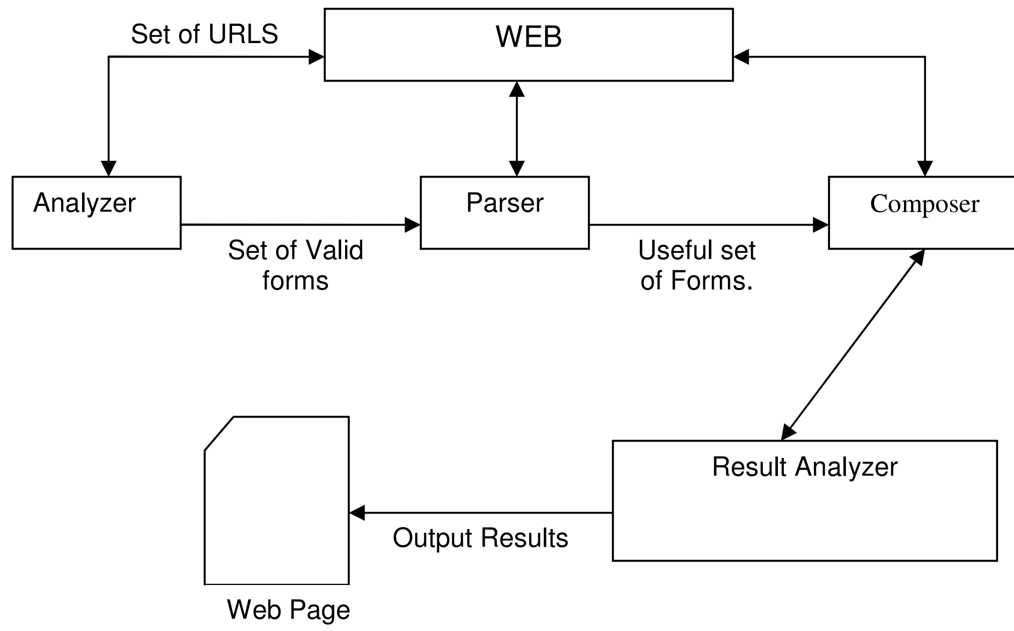

Figure 5(b): Crawl System

Figure 5: Architecture-Distributed HW crawler Configuration

Communication in our prototype is done via file transfer system through JAVA RMI. The use of file transfer system makes the design very flexible. While file transfer system has its performance limitations, we believe that the basic approach will scale well for network of workstations and that if necessary it would be easy to switch to a more optimized file transfer mechanism. We discuss some more detail information on each component in this section.

3.1 Crawl Manager

The crawl manager is the central component of the system, and the first component that is started up. Afterwards, all the Crawl System components from different workstations are started and register with the manager to request URL files. Each Crawl system downloads the list of URLS in the form of text file from the crawl managers, where each Crawl system gets different set of URLS. After processing the URL file each Crawl System uploads the result text file. In general the goal of crawl manager is to allow each Crawl system to download URL text files and to upload the results files. The central role of crawl manager is to distribute the work load among available Crawl system to enhance the system efficiency.

3.2 Crawling Application.

The crawling application we consider in our prototype is a breadth-first crawl starting out at a set of seed URLs, in our case the URLs of the main pages of several auto trader websites. The application then parses each downloaded page for hyperlinks, check whether these URLs have already been encountered before, and if not then whether the current web page has the specified search

string used to find relevant URL forms, For e.g. In our prototype we have selected vehicle search then we would like to have links of those pages which has “Enter the zip code” or “Zip Code” text somewhere in that page. Crawling Application is also responsible to take care of robot exclusion. Next we discuss about the breadth-first algorithm and robots.txt exclusion.

3.2.1. Breadth-first algorithm

Despite the numerous algorithms for crawlers such as breadth-first algorithm, depth-first algorithm, focus crawling, at the core they are all fundamentally the same. Following is the process by which crawlers work:

1. Download the Web page.
2. Parse through the downloaded page and retrieve all the links.
3. For each link retrieved, repeat the process.

In the first step, a Web crawler takes a URL and downloads the page from the Internet at the given URL. In the second step, a Web crawler parses through the downloaded page and retrieves the links to other pages. Each link in the page is defined with an HTML anchor tag similar to the one shown here:

```
<A HREF="http://www.host.com/directory/file.html">Link Text</A>
```

After the crawler has retrieved the links from the page, each link is added to a list of links to be crawled. The third step of Web crawling repeats the process. All crawlers work in a recursive or loop fashion till the specified time or maximum number of URLs found. In our prototype we have used breadth-first crawling, but as our configuration for entire system is flexible in such a way, if later on we want to change it to Depth-first crawling we can change it by changing Crawling

Application component only. Breadth-first crawling checks each link on a page before proceeding to the next page. Thus, it crawls each link on the first page and then crawls each link on the first page's first link, and so on, until each level of links has been exhausted.

3.2.2 Robot Protocol

As you can imagine, crawling a Web site can put an enormous strain on a Web server's resources as a myriad of requests are made back to back. Typically, a few pages are downloaded at a time from a Web site, not hundreds or thousands in succession. Web sites also often have restricted areas that crawlers should not crawl. To address these concerns, many Web sites adopted the Robot protocol, which establishes guidelines that crawlers should follow. Over time, the protocol has become the unwritten law of the Internet for Web crawlers. The Robot protocol specifies that Web sites wishing to restrict certain areas or pages from crawling have a file called robots.txt placed at the root of the Web site. Ethical crawlers will reference the robot file and determine which parts of the site are disallowed for crawling. The disallowed areas will then be skipped by the ethical crawlers.

Following is an example robots.txt file and an explanation of its format:

```
# robots.txt for http://somehost.com/
User-agent: *
Disallow: /cgi-bin/
Disallow: /registration # Disallow robots on registration page
Disallow: /login
```

The first line of the sample file has a comment on it, as denoted by the use of a hash (#) character. Comments can be on lines unto themselves or on statement lines, as shown on the fifth line of the preceding sample file. Crawlers reading robots.txt files should ignore any comments. The third line of the sample file specifies the User-agent to which the Disallow rules following it apply. User-agent is a term used for the programs that access a Web site such as browsers. Web crawlers also typically send a User-agent value along with each request to a Web server. The use of User-agents in the robots.txt file allows Web sites to set rules on a User-agent–by–User-agent basis. However, typically Web sites want to disallow all robots (or User-agents) access to certain areas, so they use a value of asterisk (*) for the User-agent. This specifies that all User-agents are disallowed for the rules that follow it. The lines following the User-agent line are called disallow statements. The disallow statements define the Web site paths that crawlers are not allowed to access. For example, the first disallow statement in the sample file tells crawlers not to crawl any links that begin with "/cgi-bin/". Thus, the URLs

http://somehost.com/cgi-bin/

http://somehost.com/cgi-bin/register

are both off limits to crawlers according to that line. Disallow statements are for paths and not specific files; thus any link being requested that contains a path on the disallow list is off limits.

3.3 Crawl System (Content Extraction)

Consider the following scenario. A user is doing shopping for a “used car” on the web. He visits numerous auto traders Websites looking for different types of used cars available in his locality. He looks up the price on all different Websites for available options. Finally, after collecting information from various auto Websites, he summarizes them and makes his choice. Now as we see in this, its kind of a tedious work the user is doing; which involves visiting numerous Websites and entering the keywords into the search forms and then collecting the results.

3.3.1. The Basic Structure

The prototype of the Crawl system for content extraction from hidden web been presented here will do the same way as described above. The whole crawler can be divided into 4 different stages.

1) Analyzer

2) Parser

3) Composer

4) Result analyzer

3.3.2. Analyzer

The World Wide Web has millions of Web pages. Some of them will have forms. These forms will act as an entry point for the vast information hidden behind them. Our prototype WebCrawler will be scanning millions of Web pages to find such kinds of form. However, different domain Websites will have different types of forms in it. For e.g. a hospital Website might have a form used to find

information about the patients and other health diseases. A Website related with weather information, will have a form used to get the weather and other related information for that particular area. Our prototype will be analyzing most of the Websites and will extract the forms that are related with shopping. The part of going to the Web and looking for maximum URLs and then extracting specially shopping URLs which has forms will be taken care by our Analyzer part.

The analyzer will be analyzing each and every Web page the crawler comes across. It will scan the Web page to see if the Web page can be used as search page to retrieve information or not. It will basically see if the Web page has some form fields or not. Like for e.g. product name or type of product looking for. Also, the analyzer will help us to find the difference between a registration page and a query page also. This is one of the most important components while looking into the shopping Web pages since most of the shopping Web pages also have registration forms. Hence, in order to increase the efficiency of the analyzer in searching for best shopping Web pages and thereby providing us with the best results; in the crawler, we have to place a module which can identify between shopping forms and other types of forms as well.

3.3.3. Parser

A form can have various kinds of input fields. Once the Web page is detected from where the form can be filled and made a search; parser is called. The main job of the parser is to look for various types of forms found on that Web page. Once a suitable form is found from where the query can be made; that particular form is been separated out and passed it on to the composer. It is very

essential that the form which is been extracted out by the parser can be used to perform users query. For e.g. a user wants to search for “laptop”, hence the form been extracted should be able to make a search with keyword “laptop”. Sometimes the Web page have many other forms like ones for sign in. hence, since the fields for the sign in form does not match with the ones in which “laptop” keyword should fit, such sign in form can be ignored.

3.3.4. Composer

The user is been asked to store the keywords that he would like to be in the search. These keywords are been stored and are used by composer in filling up the forms. Sometimes Web pages are been designed to trick WebCrawler’s. Hence, it is more likely that parser would have selected the wrong form to fill. The composer will try to fill up the form with the information provided by the user. While trying to do so, if the information does not fit in the form it means that such kind of form is useless. For e.g. WebCrawler is trying to make a search with keywords “laptop” in a shopping Website of jewelers.

Those forms that are been successfully filled are then executed and the results obtained are been stored for result analyzer. The designing of the composer needs to be taken special care because once these forms are filled and submitted; most of the shopping Websites will return cookies along with the search results. These cookies are needed to be handled or else the program will start throwing exceptions. Thus, composer is one of the key components in our WebCrawler.

3.3.5. Result analyzer

Once the composer fills up the forms and executes them, the results of the forms are been obtained in the form of Web pages. Now it is the job of the result analyzer to analyze each and every result on that particular Web page and then extract the necessary results from it. The input to the result analyzer will be a set of Web pages containing results obtained after executing the forms. These results will be evaluated depending upon the type of search made. If a particular Web page has results that are more in number corresponding to the other ones then we can say that particular Web site is more likely to be the right one and hence, if similar keywords search will be made on that particular Web site then the search results will be more likely to be the right ones. Hence, result analyzer will play an important role to determine the accuracy of the WebCrawler. The result analyzer will be modifying the knowledge base till it reaches optimization level. In the Figure 5(b), we can see various different modules of our prototype for Crawl System to extract content from hidden web.

3.4 Scaling the System

One of our main objectives was to design a system whose performance can be scaled up by adding additional low-cost workstations and using them to run additional components. Starting from the configuration in Figure 5(a), we could simply add additional Crawl Systems to improve performance. We would estimate that a single manager would be fast enough for about 8 Crawl Systems.

Beyond this point, we would have to create a second crawl manager and the application would have to split its requests among the two managers.

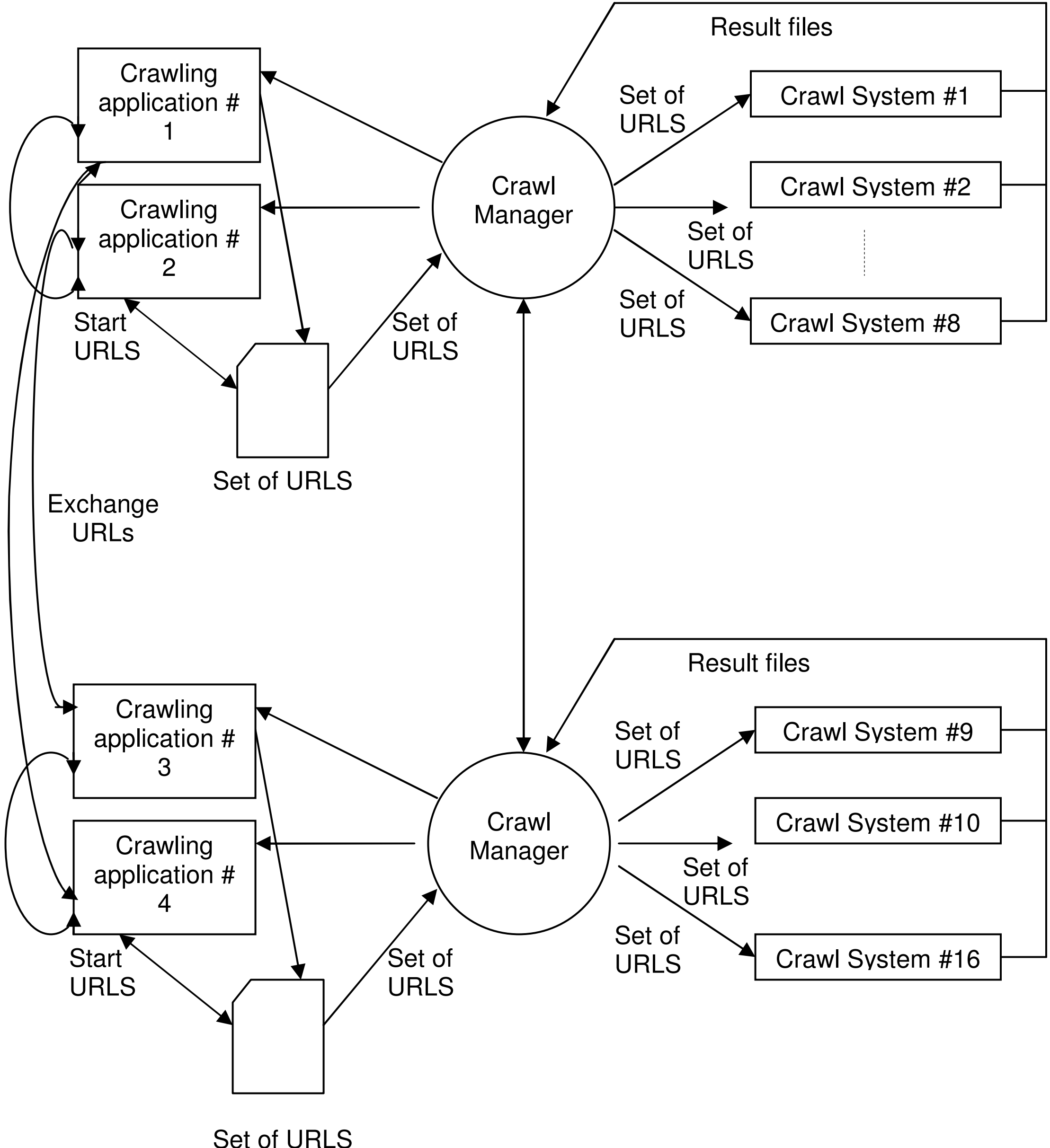

Figure 6: Large Configuration for HW Crawler

However, the first bottleneck in the system would arise before that, in our crawling application, to create set of URLs within single domain, because one

typical workstation has limit for number of pages to download as well as process to retrieve more links. So eventually it would become necessary to partition the Crawling application onto several machines. Figure 6 shows a scaled up version of our system that uses two crawl managers, with 16 Crawl Systems with 4 crawling application components in total. Let's point out that we have never been able to test the performance of such a large configuration, which would involve about 20 machines. Thus, the following is somewhat speculative. Partitioning the breadth-first crawler into 4 components is actually quite simple, using a technique similar to that employed by the Internet Archive crawler [16]. We simply partition the space of URLs into 4 subsets using a hash function, such that each application component is responsible for processing and requesting one subset. Recall that the manager will make sure that pages downloaded for different applications are stored in separate directories, as determined by the applications. If during parsing, a component encounters a hyperlink belonging to a different subset, then that URL is simply forwarded to the appropriate application component. Each crawl manager could be used by two application components; to the manager, they would appear as completely unrelated applications. Of course, in such a system synchronization of shared resources needs more care, though there are standard techniques for this. We note that the only communication involving significant amounts of data is the transmission of the result files and URLs files. Thus, in principle our system could be used even in a wide-area distributed environment, assuming we can tolerate the fact that the result-files and URLs-files may end up in several remote locations.

3.5 Limitations and Assumptions.

Assumptions for out prototype are as below:

1. Domain is been specified already and Crawling application is provided with all seed URLs from one single domain.

Limitations for prototype are as follow:

1. Crawling application which is used to retrieve links from specific domain supports only HTTP links, not HTTPS or FTP.
2. URLs that redirect to another URL are not supported.
3. Experimental results for distributed systems are measure in terms of execution time only. As the network speed is not consistent as well as not known correctly the measurements in terms of pages/second are not performed.
4. Our prototype is not supporting or is not able to process those websites when called loaded cookies in extensive numbers so as to cause the analyzer to throw exceptions.
5. Websites has numerous html forms tag in it that makes the crawlers internal cache run out of resources and hereby causing memory leaks.
6. Websites which are designed using Ajax and JavaScript are a limitation to our prototype. The limitation has been aroused because of the development environment been used does not support fully JavaScript and Ajax.

4 DESIGN MODULES: IMPLEMENTATION DETAILS

The design modules for our prototype of domain-specific HW crawler are as below. We discuss technical and implementation details for each module.

4.1 Crawling Application.

As we have mentioned earlier, crawling application uses breadth-first algorithm to crawl the seed URLs and to create list of URLs from seed URLs in the specific domain only. We are using some various techniques to match the URL pattern in the downloaded html webpage as well some pattern matching for the search string also.

4.1.1 Crawl Function

The crawl ( ) method is the core of the Crawling Application because it performs the actual crawling and thus controls the order of crawling (breadth-first algorithm). It begins with these lines of code:

```
// Set up crawl lists.
HashSet crawledList = new HashSet();
LinkedHashSet toCrawlList = new LinkedHashSet();
// Add start URL to the To Crawl list.
toCrawlList.add(startUrl);
```

There are several techniques that can be employed to crawl Web sites, recursion being a natural choice because crawling itself is recursive. Recursion, however, can be quite resource intensive, so the Search Crawler uses a queue technique. Here, toCrawlList is initialized to hold the queue of links to crawl. The start URL is then added to toCrawlList to begin the crawling process. After initializing the To

Crawl list and adding the start URL, crawling begins with a while loop set up to run until the To Crawl list has been exhausted which is maximum number of URLs specified in Crawling System, as shown here:

```
/* Perform actual crawling by looping
through the To Crawl list. */
while (crawling && toCrawlList.size() > 0)
{
/* Check to see if the max URL count has
been reached, if it was specified.*/
if (maxUrls != -1) {
if (crawledList.size() == maxUrls) {
break;
}
}
```

First, the URL at the bottom of the To Crawl list is “popped” off. Thus, the list works in a first in, first out (FIFO) manner. After retrieving the next URL from the To Crawl list, the string representation of the URL is converted to a URL object using the verifyUrl( ) method. Next, the URL is checked to see whether or not it is allowed to be crawled by calling the isRobotAllowed( ) method.

Next, the page at the given URL is downloaded with a call to downloadPage( ). If the downloadPage( ) method successfully downloads the page at the given URL, the following code is executed:

```
/* If the page was downloaded successfully, retrieve all of its
```

```
links and then see if it contains the search string. */
if (pageContents != null && pageContents.length() > 0)
{
// Retrieve list of valid links from page.
ArrayList links =
retrieveLinks(verifiedUrl, pageContents, crawledList,limitHost);
// Add links to the To Crawl list.
toCrawlList.addAll(links);
/* Check if search string is present in
page, and if so, record a match. */
if (searchStringMatches(pageContents, searchString,caseSensitive))
{
        addMatch(url);
}
}
```

First, the page links are retrieved by callingFirst, the page links are retrieved by calling the retrieveLinks( ) method. Each of the links returned from the retrieveLinks( ) call is then added to the To Crawl list. Next, the downloaded page is searched to see if the search string is found in the page with a call to searchStringMatches( ). If the search string is found in the page, the page is recorded as a match with the addMatch( ) method.

4.1.2 retriveLinks Function

The retrieveLinks( ) method uses the regular expression API to obtain the links from a page. It begins with these lines of code:

```
// Compile link matching pattern.
Pattern p =
Pattern.compile("<a\\s+href\\s*=\\s*\"?(.*?)[\"|>]",
Pattern.CASE_INSENSITIVE);
Matcher m = p.matcher(pageContents);
```

Many of the links found in Web pages are not suited for crawling. First, empty links are skipped so as not to waste any more time on them. Second, links that are simply anchors into a page are skipped by checking to see if the first character of the link is a hash (#). Next, “mailto” links are skipped. Mailto links are used for specifying an e-mail link in a Web page. For example, the link

mailto:contact@autotrader.com

Additionally, JavaScript functionality can be accessed from links. Similar to mailto links, JavaScript links cannot be crawled; thus they are overlooked.

4.1.3. isRobotAllowed Function

The isRobotAllowed( ) method fulfills the robot protocol. In order to efficiently check whether or not robots are allowed to access a URL, Search Crawler caches each host’s disallow list after it has been retrieved. This significantly improves the performance of Search Crawler because it avoids downloading the disallow list for each URL being verified. Instead, it just retrieves the list from

cache. If the disallow list for URL is null that means it needs to download disallow list.

```
try {
URL robotsFileUrl = new URL("http://" + host + "/robots.txt");
// Open connection to robot file URL for reading.
BufferedReader reader =
new BufferedReader(new InputStreamReader(
robotsFileUrl.openStream()));
```

As mentioned earlier, Web site owners wishing to prevent Web crawlers from crawling their site, or portions of their site, must have a file called robots.txt at the root of their Web site hierarchy. Finally function reads the content of the robots.txt file to check if it has disallow list. This function also takes care of the comments(#) in robots.txt file. Function adds the list of disallow to disallowcase and then checks disallow list to see if current URL is allowed for crawling.

4.2 Crawl Manager

One method of improving a Crawl Manager's performance is to create a thread for each remote method invocation, allowing multiple Crawl Systems to be processed concurrently. Therefore, several Crawl systems are not blocked in a queue waiting for their call to be executed. Java RMI automatically provides you with this level of server-side threading; this policy is described as follows in the RMI specification:

"A method dispatched by the RMI runtime to a remote object implementation (a server) may or may not execute in a separate thread. Calls

originating from different clients Virtual Machines will execute in different threads. From the same client machine it is not guaranteed that each method will run in a separate thread"

Therefore, if you make remote calls from separate Crawl Systems (executing in different JVMs basically executing from different work-stations) each call will run in a separate thread. As RMI itself forces server-side multithreading, Crawl manager takes care of shared resources to be accessed in synchronized manner.

```
// This method uploads the result file from the client.....
// As RMI itself forces server-side multithreading, the method must be synchronized as Resultfile on/ server is shared resource.
synchronized public void uploadFile(byte[] fileData) { …. }
```

Crawl manager makes sure that result file from each client is appended correctly in synchronized manner in the results file on server side.

4.3 Crawl System (Content Extraction)

The design modules for prototype of Craw System are as below.

4.3.1. Analyzer

The main task of the analyzer is to look for Web pages that have forms in it. A set of URL's of Websites are been given to the analyzer from a text file. The analyzer will read each and every single URL from that text file and will then start loading them into URL class. The URL class object is then passed on to the WebClient object which will then load the URL object and will return a Web page object. This Web page object has the whole Web page of that particular Website into it.

It's just like the Document class of java. Once the Web page object is been obtained, it is been passed on to a separate function to look for forms in it. In order to look for forms into a Web page object we are using a DOM Parser. This parser is been provided with a regular expression "//table" which means to look for html tag table at any depth in that particular Web page object.

The function been called is of the form

```
ArrayList a =getHtmlElementSByXPath ("//table", HtmlParser object);
```

As we see above, we are passing the regular expression and a parser object. Below, is the code that is been executed when the function is been called. We see that DOM Parser is been provided with the regular expression path. Later on, the DOM Parser returns with all the selected Nodes that are been obtained corresponding to the regular expression. All these nodes will be containing html tag table and hence, they will be representing actual tables in the form of nodes. All these nodes, where each node corresponds to a particular table will be stored in ArrayList object.

```
ArrayList getHtmlElementSByXPath (final String exp, final
HtmlPage page) throws Exception {
Final HtmlUnitXPath xpath = new HtmlUnitXPath(exp);
return (ArrayList) xpath.selectNodes(page);
}
```

4.3.2 Parser and Composer

The main job of the parser is to look for html forms on a Web page and see if those forms are actually the search forms and not registration forms. Based upon the survey been made on large number of Websites, it is been observed that these forms which are designed for search function has specific type of format. Normally, the submit buttons of these forms are found to be named as “GO” or “Search”. Besides this, they will contain one common structure of like having a text field named as “ZipCode” or “Zip Box” as we are working on domain having auto trader website and a submit button. However, this find of format of having text field with that particular name and a submit button are not found in all the Websites, but they are the most common structures found and it also results into large number of successful search results. Hence, in our prototype design, we will be using these structures in our parser.

The function that does the job of the Parser is as shown below:-

```
static void formRipper(String UrlLink){
try{
final WebClient webClient = new WebClient();
final URL url = new URL(UrlLink);
final HtmlPage page1 = (HtmlPage)webClient.getPage(url);
List l = page1.getForms();
int i=0;
HtmlInput ip = null;
HtmlTextInput textField = null;
```

```
while(i < l.size())
{
    final HtmlForm form = (HtmlForm)l.get(i);
    try{
        ip = (HtmlInput)form.getInputByName("Search");
    }catch(Exception e){}
    if(ip == null) // Search did not work
    {
    try{
        ip = (HtmlInput) form.getInputByName("search");
    }catch(Exception e){}
    if(ip == null) // search did not work
    {
    try{
        ip = (HtmlInput) form.getInputByName("go");
    }catch(Exception e){}
    if(ip == null)
    {
    try{
        ip = (HtmlInput) form.getInputByName("GO");
    }catch(Exception e){}
    } // if ends
    } // if ends
```

```
} // if ends
if(ip == null)
{
        /* get all inputs and see if there is any image input with
        * name search or go
        */
        ArrayList AllInputs = new
        ArrayList(form.getInputsByName(""));
        try{
                ip = (HtmlInput)AllInputs.get(0);
        }
        catch(Exception e){}
        /* searching for image ends */
        if(ip == null)
        { i++;
                continue;
        }
}
// now looking for the text part in the same form
ArrayList ListOfText = new
ArrayList(form.getInputsByValue(""));
if(ListOfText.size() == 0)
ListOfText = new
```

```
ArrayList(form.getInputsByName("Zip Box"));
for(int k=0;k < ListOfText.size(); k++)
        System.out.println(k + " " +
ListOfText.get(k).toString());
if(ListOfText.size() >= 1){
textField = (HtmlTextInput)ListOfText.get(0);
try{
        submitThisForm(ip,textField);
}
catch(Exception e){}
break;
}
else
        i++;
} // while ends
}catch(Exception e)
{System.out.println(e);}
} // formRipper ends
```

We begin by picking up a URL from the input file and submit it to WebClient. The WebClient will return an HtmlPage object. Then we will be using a method called getForms( ) which will be internally using a parser having regular expression "//form". This will return to us a list of all nodes that begin with html tag form. Now, we enter into the heart of the parser. Once, we get the list of forms, we start

iterating through it one after another. While iterating through each and every form, we will do the following actions.

1. Look for a keyword “search” or “go”. If we encounter any of these keywords, it will mean that we found the submit button of the form. If the form does not have any of these keywords, it will mean that the form does not possess any submit button. Hence we can go to step 2.

2. Look for images that have keywords like “search” or “go”. Shopping Websites are now more interactive and attractive. Hence, in place of using traditional looking submit button, it might be possible that they have used an image button with name search or go. And looking at the number of URL’s found on the Web for shopping, around 60 % of the Websites will use image or any other method rather than a plain submit type button. Once, we find such kind of input type button, we can proceed to step 3 or we can pick up the next form and proceed to step 1.

3. After finding the submit button, now its time to look for text field. The text field will be helpful in filling up the keywords that user has specified. Once we find the text field we are all set to fill it up with the user keywords (done using text field.setAttributes (“user keywords”) and clicking on submit button. It is done through submitButton.click ( ) the function submitButton.click ( ) will return a new Web page object that will have the result page from that particular Website.

After the result page is been obtained, we pass it on to Result Analyzer to analyze the results from that particular Web page. Please note, in all of the

above steps, getInputsByName ( ) is the function been used to obtain the nodes that correspond to specific name. For e.g. getInputsByName (“search”) will return with the node that has name = “search”. Since the function getInputsByName ( ) is case-sensitive, the results obtained by getInputsByName (“search”) and getInputsByName (“Search”) will be different.

4.3.3. Result Analyzer

The result analyzer will analyze each and every Web page it receives for search results obtained by making the query done by composer. The piece of code that does this is presented below:-

```
void findTables(HtmlPage hp,PrintStream p) throws Exception
{
ArrayList a = getHtmlElementSByXPath("//table//table",hp);
findExactOne(a,p);
} // fn ends
/* The function given below will take one single table as
* input and will extract all the rows from it. It will look
* for the one that has “$” sign.
*/
void findExactOne(ArrayList a, PrintStream p)
{
try{
for(int i=0;i<a.size();i++)
{
```

```
HtmlTable ht = (HtmlTable)a.get(i);
List rows = ht.getRows();
for(final Iterator rowIterator = rows.iterator();
rowIterator.hasNext();)
{
final HtmlTableRow row = (HtmlTableRow)
rowIterator.next()
final List cells = row.getCells();
for(final Iterator cellIterator = cells.iterator();
cellIterator.hasNext();)
{
Final HtmlTableCell cell = (HtmlTableCell)
cellIterator.next();
if(cell.asText().indexOf("$",0) > 0){
rowExtractor(ht,p);
}
}
}
}
}catch(Exception e)
{
System.out.println("found in findExactOne :"+e);
}
```

```
} // fn ends

/* this function will extract rows and will stuff it into a

* file it will extract only those rows that has $ sign in

* it

*/

void rowExtractor(HtmlTable ht,PrintStream p)

{

int rowSize = ht.getRows().size();

List rows = ht.getRows();

p.println("<p>");

for(int i=0; i<rowSize; i++)

{

HtmlTableRow htr = (HtmlTableRow)rows.get(i);

p.println("<i>Entry # "+i+"</i>");

for(int j=0; j<htr.getCells().size();j++)

{

p.println("<div align='left'>");

p.println(ht.getCellAt(i,j).asText());

p.println("</div>");

}

}

p.println("</p>");

} // fn ends
```

```
/* this function will extract one single row from a given
table */
void rowCellExtractor(HtmlTableCell htc,PrintStream p)
{
p.println(htc.asXml());
} // fn ends
```

The result analyzer is called by passing a Web page as an object. Once the Web page object is obtained, the following steps take place. The Web page is looked for html tag table. Once all these tables are obtained in the form of ArrayList they will be passed on to step 2. In this step, the function findExactTable ( ) is called which will take one table at a time and will look for “$”. Since each and every Web page result obtained after making a search on auto trader website will have $ tag, with the price for that particular car. We will be looking for all the tables that have this tag in their rows. If the table has that particular tag, it means that we found the right table.

Once the right table is found, rowExtractor ( ) is called. This function will extract each and every single row of the table preceding it. The results on selecting the table preceding it leads to better accuracy rather than selecting the table that has $ tag. The rows of the preceding table is been printed into the output file.

5. OBSERVATIONS AND EXPERIMENTAL RESULTS

As we have mentioned earlier we have selected auto trader websites as the domain. Considering that only and using such URLs base several tests were run by making searches with following keyword – value pairs:-

1. Zip Box - 10001
2. Zip Box - 30324
3. Zip Code - 10001
4. Make Box – Any
5. Make Box – Toyota.

The results obtained by making above searches are as below.

Keyword Search	Keyword Value	Number of successful URL's	Total number of URL's	Total number of entries T	Relevant entries (approx.) R	%age of success (R/T) * 100
Zip Box	10001	47	103	1456	522	35.85
Zip Box	30324	47	103	923	307	33.26
Zip Code	10001	37	103	1087	322	.29.62
Make Box	Any	32	103	812	259	31.89
Make Box	Toyota	32	103	563	139	24.51

Table 1: Search results with //table//table regular expression

5.1 Analyzer: //table OR //table//table regular expression

A change was made in the analyzer by changing the regular expression from //table//table to //table. Due to this, the analyzer will look only for the top level table tags. These table tags may or may not contain sub tables in them. This should degrade the performance of the analyzer. The reason on selecting //table//table to be more suitable is that this regular expression will look for table tags which are child tables of some parent table. It is because the table tag at the top level is mainly used for formatting the whole Web page. The subsequent table tags are then used for actual contents in them. Hence, selecting //table tag will give us the master table and also the subsequent child tables. Thus, there will be more duplication of the results been produced by the WebCrawler if master table is introduced. The performance chart is shown below in Table 2.

Keyword Search	keyword Value	Number of successful URL's	Total number of URL's	Total number of entries T	Relevant entries (approx.) R	%age of success (R/T) * 100
Zip Box	10001	47	103	1678	607	36.17
Zip Box	30324	47	103	1020	343	33.62
Zip Code	10001	37	103	1236	372	30.09
Make Box	Any	32	103	1008	334	31.89
Make Box	Toyota	32	103	789	216	27.37

Table 2: Search results with //table regular expression

5.2 Good and Bad URLs

The URL's used in the above experimental searches are related to auto trading Websites. Out of 100 URL's been searched over the internet, on average 17 were recognized with its form credibility. In other words, the WebCrawler was presented with 100 auto trading URL's. During Parsing of these 100 URL's, the WebCrawler was only able to find 17 good URL's. These 17 URL's were the ones that matched the form criteria (the form will have a submit/image button saying search or go and it will contain one textfield for keywords entry). These 17 search forms where genuine to search only the vehicles in specified locality. Some auto trading websites also has search forms to search within web site, which are not important to us. Though 17 out of 100 is not a good number to say, but there were many other issues as well that prevented the WebCrawler from identifying the Web page.

Today looking at the World Wide Web, the Web pages are just not written in plain HTML. There are numbers other scripting languages like PHP, JavaScript, AJAX been used in one single Web page. The key component that is been used in designing this WebCrawler is HtmlUnit tm. The limitation of HtmlUnit is that it's still not able to function properly on advanced scripting languages like JavaScript and AJAX where the Web page is generated on the fly. This limitation of HtmlUnit was one main drawback that resulted into accuracy loss in identifying potential shopping URL's. Some of the pioneering Web sites like amazon.com, ebay.com had special piece of code within that would prevent

the WebCrawler from reading the Web page successfully. Hence, when the parser came across such Web sites it would normally throw exceptions and would continue searching for another URL.

Below is the piece of code that resulted parser to create an exception in amazon.com

```
<script type="text/javascript">//<![CDATA[
n2RunThisWhen(
n2sRTWTBS,
function() {
goLolPop = new N2SimplePopover();
goN2Events.registerFeature('lolPop', 'goLolPop', 'n2MouseOverHotspot',
'n2MouseOutHotspot');
goN2Events.setFeatureDelays('lolPop', 200, 400, 200);
goLolPop.initialize('lolPopDiv', 'goLolPop', null, null, 'below', 'c');
if (document.getElementById('lolPop_1') != undefined)
{       goN2U.insertAdjacentHTML(document.getElementById('lolPop_1'),
'beforeEnd',
' \;<img src="http://ec1.images-amazon.com/images/G/01/xlocale/
common/icons/drop-down-icon-small-arrow.gif"          width="11"
style="margin:0px
2px -1px 4px;" height="11" border="0" />');
}
},
```

```
'Your Lists popover' );
//]]></script>
<script type='text/javascript'>
n2RunThisWhen(
'holiday06Loaded',
function()
{
goUSHolidayNavSwf.loadHolidaySwf(
"swfContainer",
"navbarTabsTable", "subnavTable", 11,
"subnavAndSearchTable",
"_po_holidayNavSnow",
"http://ec1.images-amazon.com/images/G/01/nav2/images/skins/holiday-
2006/gw_snow08_nobtn._V37040087_.swf",
"7.0r24"
);
},
"initialize holiday swf logic"
);
</script>
```

The piece of code that lead parser an exception on ebay.com is as below.

```
<script
src="http://include.ebaystatic.com/js/e485/us/homepage_e4852us.js"
```

```
type="text/javascript"></script><link rel="stylesheet" type="text/css"
href="http://include.ebaystatic.com/aw/pics/us/css/homepage.css"><style
type="text/css">
A.whitelinks:link{color:#ffffff;}
.subtext{font-size: 11px;}
.buttonsm {font-size: 11px; cursor: hand;}
.btmbrdr {background: #FFFFE5
url(http://pics.ebaystatic.com/aw/pics/userSitePrefs/bottomDropShadow_2
0x20.gif)
repeat-x bottom;}
.rtbrdr {background: #FFFFE5
url(http://pics.ebaystatic.com/aw/pics/userSitePrefs/sideDropShadow_20x
20.gif) repeat-y
left;}
.rt1brdr {background: #FFFFE5
url(http://pics.ebaystatic.com/aw/pics/userSitePrefs/dropshadow2_20x10.g
if) repeat-y
left;}
.lftbrdr {background-color: #FEEEA3;border-left: 2px solid #F9B709;}
.lft1brdr {background-color: #FFFFE5;border-left: 2px solid #F9B709;}
.topbrdr {background-color: #FEEEA3;border-top: 2px solid #F9B709;}
.favSelect {width: 100%;}
```

```
.favNavHeader { margin: 0;padding: 0 5px 0 0;background-color:
#CECEFF;border:
1px solid #A9A9F7;}
.favNavHeaderBuy {margin: 0;padding: 0 5px 0 0;background-color:
#E2E3FF;bordertop:
1px solid #A9A9F7;border-right: 1px solid #A9A9F7;
border-left: 1px solid #A9A9F7;}
.favNavContent {margin: 0; padding: 5px;background-color: #FFF;border-
width: 0 1px
1px 1px;border-style: solid;border-color: #A9A9F7;}
.favCenter {padding: 0 16px 0 16px;}
@media all {
IE\:HOMEPAGE {behavior:url(#default#homepage)}
}
</style><style type="text/css"><!--
.favsearchbar {
margin-top:0px;
width:785px;
height:24px;
padding-top:2px;
background-color:#F3F3F3;
padding-bottom:2px;
padding-left:10px;
```

```
margin-bottom:0px;}
--></style></head><body bgcolor="#FFFFFF" link="#0000FF"
onLoad="init();toolboxOnLoad();" onUnload="cleanUp();"><script
language="javascript" type="text/javascript"><!--
if(document.all)
document.write("<IE:HOMEPAGE ID = 'oHomePage' />");
//--></script>
```

The above were some pieces of code that resulted parser to throw the exception. Also, there were many other some Websites which mislead the analyzer to get the correct outputs. As there are no universal standards for Web site building different website uses different layouts, uses nested tables or frames differently. Also different company has different naming standards for the input fields even in simple html code. In such cases it becomes really difficult to generalize our analyzer to handle all the possibilities.

5.3 Measurements of execution time

As we have discussed there is lot of complex analysis and processing requires for each URL to extract content from the hidden data base. Some of the pioneering Web sites like amazon.com, ebay.com had special piece of code within that would prevent the WebCrawler from reading the Web page successfully. Crawl system has to take care of large number of exceptions because of such websites. Our prototype is not supporting or is not able to process those websites when called loaded cookies in extensive numbers so as to cause the analyzer to throw exceptions. Many websites has numerous html

forms tag in it that makes the crawlers internal cache run out of resources and hereby causing memory leaks. All these complexities together make it very eminent to develop distributed system for efficiency. Table 3 shows some observations about the difference between execution time for distributed HW crawler and standalone HW crawler. Through out these execution time experiments we have used 4 work-stations including one server.

Domain	No. of URLs	Stand-alone HW crawler execution time in sec.	Distributed HW crawler execution time in sec. (4 nodes including Server)
Auto Links	103	834.19	345.07
Comp Links	150	1482.57	614.23

Table 3: Execution time results

We can see the significant speedup in distributed system, but dividing list of URLs into three workstations should ideally reduces the execution time by 1/3 times which is not the case. The reason for such difference is that system takes different time to process each URL, because some URLs has too many html tags while others may have less. Some website use simple html code while some may use complex web technologies. This vast variation in the properties of URLS is not allowing the ideal execution time which should be approximately 1/3 times by using one server and 3 work-stations.

6. FUTURE WORKS

The prototype of the WebCrawler been presented here is a best-first crawler. It is been designed to search within shopping Websites. This prototype will execute queries on various shopping Websites and will gather the necessary results from them. Using this prototype, the WebCrawler can further be extended to enhance its performance by using a knowledge base. The Figure 7 shows new extended prototype of the WebCrawler. The prototype for future Web Crawler will be more intelligent compare to current one as it will make use of Knowledge base.

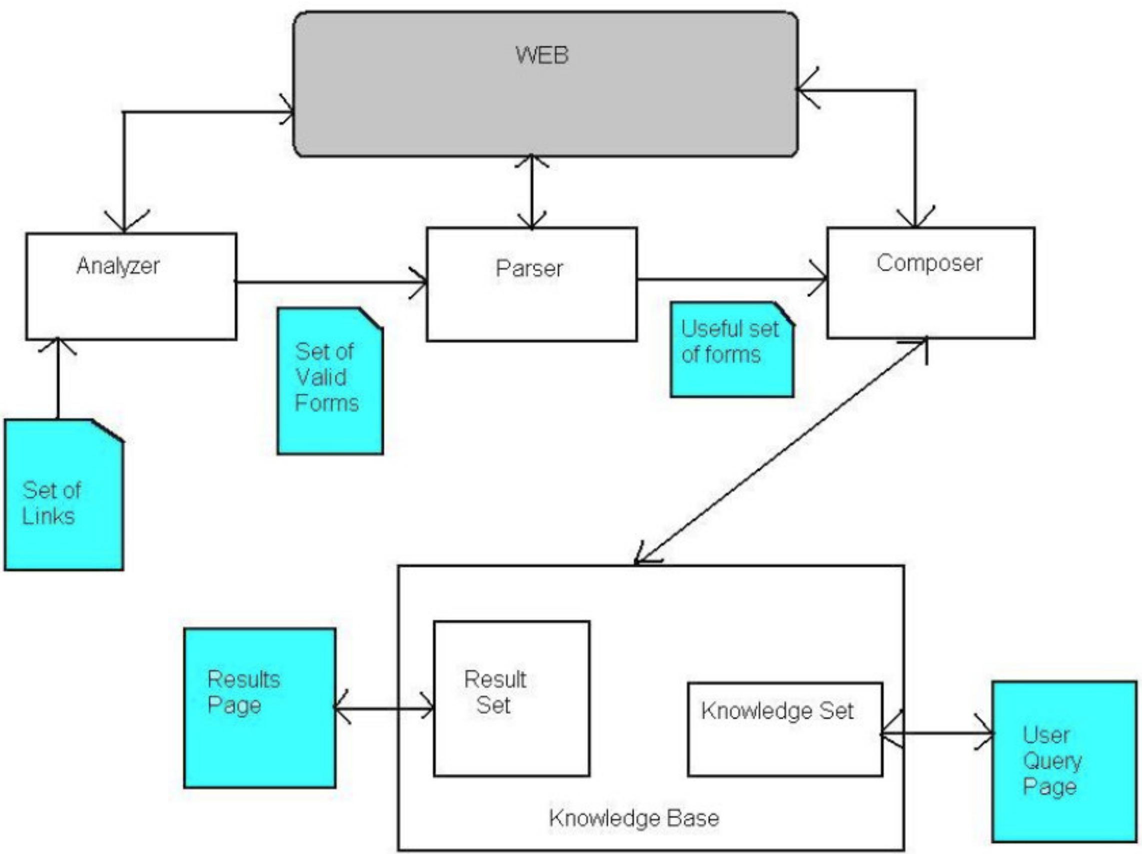

Figure 7: Future prototype of HW crawler

The whole crawler can be divided into 5 different stages.

1) Analyzer

2) Parser

3) Composer

4) Knowledge base

5) Result analyzer

Analyzer, Parser, Composer will remain the same. There will be no change in designing and its implementation of them. Compared to our old prototype, this design contains Knowledge Base. Our old prototype of WebCrawler shows no intelligence. It just crawls for the Web pages and makes the query search depending upon what the user has provided. But by using this knowledge base concept, the new prototype will have intelligence which will improve the performance of the WebCrawler.

6.1. Knowledge base

Knowledge base will be like a database providing intelligence to the WebCrawler. User will provide a set of keywords that will be related with each other. For e.g. if the user is searching for computer speakers, he might enter words like computer desktop speakers, speakers for computer, surround speakers etc. This knowledge base will grow itself as the user will keep providing more and more words for making up the query. The composer will be using knowledge set and will be making various queries to make search. The result set will be analyzing the results it obtained from the composer based on various searches been made using all the queries. The performance statistics (performance = total number of

good results / total number of results) will be used to place the weights on the query words. For e.g. the results returned by making a search “computer surround speakers” resulted into 65 % of overall results, while search named “computer speaker system” resulted into 85 % of overall results. Thus, using a mathematical formula, the weights of keywords computer and speakers will be more compared to “system” and “surround”. Thus, when user enters keywords like “computer speakers”, the performance of the results will be somewhere around 90 ~ 95 %.

6.2. Open Databases

Another suggested prototype of the WebCrawler can be for open databases. Currently, our prototype of the WebCrawler will look for shopping URL’s and extracting data from it. With the help of using open databases, the crawler can extract and analyze data flawlessly. There can be numerous open databases. One of them is the names of students in a university. Every university in USA offers a directory service in which user can type in the name of the person and get its contact information. Our crawler can be programmed to visit such Web pages and perform a people search. In some cases, the results can be obtained from more than one particular university as well. For e.g. if making a search with a persons name can get us results from GSU and from other universities like NYU as well. This will mean that, that particular person with that name can be found in GSU and in NYU as well. Imagine the possibility of having access to numerous university databases to get information of people.

6.3. Image Search Crawler

The WebCrawler can also be used to search for images with a few modifications. The WebCrawler can search for images on the World Wide Web. The image search can be based on captions [19]. Some of the WebCrawler's use a more complex algorithm to identify the images [20]. The modifications in our prototype will be done in analyzer and in parser. In the image WebCrawler, the analyzer at the front will not be looking for shopping URL's anymore. It will crawl into the World Wide Web, potentially visiting every single Website. There will be no composer in this model. The next phase will be the parser, who will look for the image captions that matches the user interest. An alternative model can involve the use of database in storing all the Web pages into the database from where the images can be searched by parser more speedily than looking dynamically on the Web.

7. CONCLUSION

There are billions and billions of Web pages on World Wide Web. Efficient generalized search on publicly indexable web pages is always there but the enormous size and significance of hidden web is too much to be overlooked. Hence, there is very momentous need to develop domain specific crawlers for hidden web. In this report, we talk about such a domain specific distributed hidden web crawler.

The domain that it is been focused on is auto-trading websites. Each auto-trading Websites have their own database to store information of all used and new cars. The prototype of the WebCrawler presented in this paper will visit all such auto trading Websites and will perform users query in order to retrieve results from their databases. With the use of this WebCrawler the users does not have to go to each and every specific auto trading Website and make its query search. Now for such web crawler to extract content from all possible auto-trading website for all possible location requires massive processing, so it becomes eminent to develop distributed system for such application

The prototype presented here is just an experiment combining of domain specific hidden web Crawler for *content extraction* and breadth-first crawler for *resource Discovery*. Combining these two techniques and developing a complete distributed system, we can see a reasonable performance been achieved. Using future techniques we can improve the prototype to work on large scale with high performance and hence achieving more accurate results.

REFERENCE

[1] Resource Link: http://www.thinkpink.com/bp/WebCrawler/History.html

[2] Kobayashi, M. AND Takeda, K. Information Retrieval of the Web, ACM Computing Surveys, 2000

[3] J. Hirai, S. Raghavan, H. Garcia-Molina, and A. Paepcke. WebBase : A repository of web pages. In *Proc. of the 9th Int. World Wide Web Conference*, May 2000.

[4] M. R. Henzinger, A. Heydon, M. Mitzenmacher, and M. Najork. Measuring index quality using random walks on the web. In *Proc. of the 8th Int.WorldWideWeb Conference (WWW8)*, pages 213–225, 1999.

[5] Bergman, M. The Deep Web: Surfacing Hidden Value, The Journal of Electronic Publishing , 2000

[6] S. Raghavan and H. Garcia-Molina. Crawling the hidden web. In *Proc. of 27th Int. Conf. on Very Large Data Bases*, September 2001. to appear.

[7] J. Talim, Z. Liu, P. Nain, and E. Coffman. Controlling robots of web search ngines. In *SIGMETRICS Conference*, June 2001.

[8] Pinkerton, B. Finding What People Want: Experiences with the WebCrawler, Second International WWW Conference,1994

[9] Resource Link: http://www.w3.org/Library/Status.html

[10] DeBra, P. and Post, R., Information Retrieval in the World-Wide Web: Making Client-based searching feasible, Proceedings of the second international WWW conference 1994 “Mosaic and the Web”,1994

[11] J. Cho, H. Garcia-Molina, and L. Page. Efficient crawling through url ordering. In *Proc. of the 7th Intl.WWW Conf.*, 1998.

[12] Y. Papakonstantinou, H. Garcia-Molina, A. Gupta,and J. Ullman. A query translation scheme for rapid implementation of wrappers. In *Proc. of the4th Intl. Conf. on Deductive and Object-Oriented Databases*, pages 161–186, National University of Singapore(NUS), Singapore, 1995.

[13] Steve Lawrence and C. Lee Giles. Searching the World Wide Web. *Science*, 280(5360):98, 1998.

[14] Raghavan, S AND Garcia-Molina, H. Crawling the Hidden Web. In Proc. of VLDB, 2001

[15] M. Najork. Atrax: A distributed web crawler. Presentation given at AT&T Research, March 20, 2001.

[16] M. Burner. Crawling towards eternity: Building an archive of the World Wide Web. 1997. http://www.webtechniques.com/archives/1997/05/burner/.

[17] Wu, W, Yu, C, Doan, A, AND Meng, W. An Interactive Clustering-based Approach to Integrating Source Query interfaces on the Deep Web. In Proc. of SIGMOD, 2004

[18] Chang, K., He, B., Li, C., Patel, M., AND Zhang, Z. Structured Databases on the Web: Observations and Implications. SIGMOD Record, 2004

[19] Rowe, N. Marie-4: A High-Recall, Self-Improving WebCrawler That Finds Images Using Captions, IEEE Intelligent Systems, 2002

[20] Kompatsiaris, I., Triantafyllou, E., AND Strintzis, M. A World Wide Web Region-Based image search engine, IEEE, 2001

[21] Shkapenyuk, V. AND Suel, T. Design and implementation of high performance distributed WebCrawler, Proceedings of the 18th International Conference on Data Engineering, IEEE, 2002

[22] Fei, L., Fan-Yuan, M., Yun-Ming, Y., Ming-Lu, L. AND Jia-Di, Y. Distributed High-Performance Web Crawler Based on Peer-to-Peer Network, Parallel and Distributed Computing, Applications and Technologies, 2004

[23] A domain based approach to crawl the hidden web, Milan Pandya, 2006.

APPENDIX A

This class holds the procedure to crawl the webpage and to retrieve links based on specified search string in that link page.

```
import java.io.*;
import java.net.*;
import java.util.*;
import java.util.regex.*;

/*
 * Created on June 8, 2007
 * this program will read the start URL from Crawling Application and will crawl start URL to create
 * list of URLs till maximum number is reached based of certain search criteria.
 */

/**
 * @author Lovekesh
 *
 * TODO To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Style - Code Templates
 */

public class actionSearchUrls
{

	// Max URLs drop-down values.
	private final int MAX_URLS = 50;

	// Cache of robot disallow lists.
	private HashMap disallowListCache = new HashMap();

	// Flag for whether or not crawling is underway.
	private boolean crawling;

	// Matches log file print writer.
	private PrintWriter logFileWriter;

	//Matches log file:
	private String urlMatchedFile = "urlMatchedFile.txt";

	// String to store the start Url;
	private String startUrl;

	// String to store the search string;
```

```
	private String searchString;

	//boolean flag for caseInsensitive
	private boolean caseInsensitive = true;


	public void startActionSearch(){

		// Validate that start URL has been entered.
		System.out.println("\n Please Enter your Start URL: ");
		BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
		try {
		startUrl = br.readLine();
	   } catch (IOException ioe) {System.out.println("unable to recognize it,
please try again");}

		startUrl = startUrl.trim();
		if (startUrl.length() < 1) {
			System.out.println("Missing or invalid Start URL.");
			System.exit(0);
		}

		// Verify start URL.
		else if (verifyUrl(startUrl) == null) {
			System.out.println("Invalid Start URL.");
			System.exit(0);
		}


		// Validate that search string has been entered.
		System.out.println("\n Please Enter your Search String: ");
		BufferedReader br1 = new BufferedReader(new
InputStreamReader(System.in));
		try {
		searchString = br1.readLine();
	   } catch (IOException ioe) {System.out.println("unable to recognize it,
please try again");}

		searchString = searchString.trim();
		if (searchString.length() < 1) {
			System.out.println("Missing Search String.");
			System.exit(0);
		}

		// Remove "www" from start URL if present.
```

```
		startUrl = removeWwwFromUrl(startUrl);

		// Start the Search Crawler.
		search(urlMatchedFile, startUrl, MAX_URLS, searchString);

	}	// Main method ends here.......


	private void search(final String logFile, final String startUrl,final int
maxUrls, final String searchString)
	{

		// Open matches log file.
		try {
			logFileWriter = new PrintWriter(new FileWriter(logFile));
		} catch (Exception e) {
			System.out.println("Unable to open matches log file.");
			return;
		}

		// Perform the actual crawling.
		crawl(startUrl, maxUrls, searchString, caseInsensitive);

		// Close matches log file.
		try {
			logFileWriter.close();
		} catch (Exception e) {
			System.out.println("Unable to close matches log file.");
			return;
		}

	}

	// Add match to matches table and log file.
	private void addMatch(String url) {

		// Add URL to matches log file.
		try {
			logFileWriter.println(url);
		} catch (Exception e) {
			System.out.println("Unable to log match.");
		}
	}


	// Verify URL format.
```

```
        private URL verifyUrl(String url) {
                // Only allow HTTP URLs.
                if (!url.toLowerCase().startsWith("http://"))
                return null;

                // Verify format of URL.
                URL verifiedUrl = null;
                try {
                        verifiedUrl = new URL(url);
                } catch (Exception e) {
                        return null;
                }
                return verifiedUrl;
        }

        // Check if robot is allowed to access the given URL.
        private boolean isRobotAllowed(URL urlToCheck) {

                String host = urlToCheck.getHost().toLowerCase();

                // Retrieve host's disallow list from cache.
                ArrayList disallowList = (ArrayList) disallowListCache.get(host);

                // If list is not in the cache, download and cache it.
                if (disallowList == null) {
                        disallowList = new ArrayList();
                        try {
                                URL robotsFileUrl = new URL("http://" + host +
"/robots.txt");

                                // Open connection to robot file URL for reading.
                                BufferedReader reader = new BufferedReader(new
InputStreamReader(robotsFileUrl.openStream()));

                                // Read robot file, creating list of disallowed paths.
                                String line;
                                while ((line = reader.readLine()) != null) {
                                        if (line.indexOf("Disallow:") == 0) {
                                                String disallowPath =
line.substring("Disallow:".length());

                                                // Check disallow path for comments
and remove if present.
                                                int commentIndex =
disallowPath.indexOf("#");
                                                if (commentIndex != - 1) {
```

```
                                                disallowPath                        =
disallowPath.substring(0, commentIndex);
                                        }

                                        // Remove leading or trailing spaces
from disallow path.
                                        disallowPath = disallowPath.trim();

                                        // Add disallow path to list.
                                        disallowList.add(disallowPath);
                                }
                        }

                // Add new disallow list to cache.
                disallowListCache.put(host, disallowList);
                }
                catch (Exception e) {
                        /* Assume robot is allowed since an exception
                                is thrown if the robot file doesn't exist. */
                        return true;
                }
        }

        /* Loop through disallow list to see if
                        crawling is allowed for the given URL. */
        String file = urlToCheck.getFile();
        for (int i = 0; i < disallowList.size(); i++)
        {
                String disallow = (String) disallowList.get(i);
                if (file.startsWith(disallow)) {
                        return false;
                }
        }
        return true;
}


// Download page at given URL.
private String downloadPage(URL pageUrl) {
        try {

                // Open connection to URL for reading.
                BufferedReader    reader    =    new    BufferedReader(new
InputStreamReader(pageUrl.openStream()));

                // Read page into buffer.
```

```
                        String line;
                        StringBuffer pageBuffer = new StringBuffer();
                        while ((line = reader.readLine()) != null) {
                                pageBuffer.append(line);
                        }
                        return pageBuffer.toString();
                } catch (Exception e) { }

                return null;
        }

        // Remove leading "www" from a URL's host if present.
        private String removeWwwFromUrl(String url) {
                int index = url.indexOf("://www.");
                if (index != -1) {
                        return url.substring(0, index + 3) +
                        url.substring(index + 7);
                }
                return (url);
        }


        // Parse through page contents and retrieve links.
        private ArrayList retrieveLinks(URL pageUrl, String pageContents,
HashSet crawledList) {

                // Compile link matching pattern.
                Pattern p =
Pattern.compile("<a\\s+href\\s*=\\s*\"?(.*?)[\"|>]",Pattern.CASE_INSENSITIVE);
                Matcher m = p.matcher(pageContents);

                // Create list of link matches.
                ArrayList linkList = new ArrayList();

                while (m.find()) {
                        String link = m.group(1).trim();

                        // Skip empty links.
                        if (link.length() < 1) {
                                continue;
                        }

                        // Skip links that are just page anchors.
                        if (link.charAt(0) == '#') {
                                continue;
                        }
```

```
// Skip mailto links.
if (link.indexOf("mailto:") != -1) {
        continue;
}

// Skip JavaScript links.
if (link.toLowerCase().indexOf("javascript") != -1) {
        continue;
}

// Prefix absolute and relative URLs if necessary.
if (link.indexOf("://") == -1) {

        // Handle absolute URLs.
        if (link.charAt(0) == '/') {
                link = "http://" + pageUrl.getHost() + link;

                // Handle relative URLs.
        } else {
                String file = pageUrl.getFile();
                if (file.indexOf('/') == -1) {
                        link = "http://" + pageUrl.getHost() + "/" +
link;
                } else {
                        String path = file.substring(0,
file.lastIndexOf('/') + 1);
                        link = "http://" + pageUrl.getHost() + path
+ link;
                }
        }
}

// Remove anchors from link.
int index = link.indexOf('#');
if (index != -1) {
        link = link.substring(0, index);
}

// Remove leading "www" from URL's host if present.
link = removeWwwFromUrl(link);

// Verify link and skip if invalid.
URL verifiedLink = verifyUrl(link);
if (verifiedLink == null) {
        continue;
```

```
                }

                // Skip link if it has already been crawled.
                if (crawledList.contains(link)) {
                        continue;
                }

                // Add link to list.
                linkList.add(link);
            } // while (m.find()) ends here...............
            return (linkList);
    }

    /* Determine whether or not search string is
            matched in the given page contents. */
    private    boolean    searchStringMatches(String    pageContents,    String
searchString,boolean caseSensitive) {
            String searchContents = pageContents;
            /* If case-sensitive search, lowercase
                        page contents for comparison. */
            if (!caseSensitive) {
                    searchContents = pageContents.toLowerCase();
            }
            // Split search string into individual terms.
            Pattern p = Pattern.compile("[\\s]+");
            String[] terms = p.split(searchString);
            // Check to see if each term matches.
            for (int i = 0; i < terms.length; i++) {
                    if (caseSensitive) {
                            if (searchContents.indexOf(terms[i]) == -1) {
                                    return false;
                            }
                    } else {
                            if      (searchContents.indexOf(terms[i].toLowerCase())
== -1) {
                                    return false;
                            }
                    }
            }
            return true;
    }


    // Perform the actual crawling, searching for the search string.
    public void crawl(String startUrl, int maxUrls, String searchString, boolean
caseSensitive) {
```

```
// Set up crawl lists.
HashSet crawledList = new HashSet();
LinkedHashSet toCrawlList = new LinkedHashSet();

// Add start URL to the to crawl list.
toCrawlList.add(startUrl);

/* Perform actual crawling by looping
        through the To Crawl list. */

while (toCrawlList.size() > 0)
{

        /* Check to see if the max URL count has
                been reached, if it was specified.*/
        if (maxUrls != -1) {
                if (crawledList.size() == maxUrls) {
                        break;
                }
        }

        // Get URL at bottom of the list.
        String url = (String) toCrawlList.iterator().next();

        // Remove URL from the To Crawl list.
        toCrawlList.remove(url);

        // Convert string url to URL object.
        URL verifiedUrl = verifyUrl(url);

        // Skip URL if robots are not allowed to access it.
        if (!isRobotAllowed(verifiedUrl)) {
                continue;
        }

        // Add page to the crawled list.
        crawledList.add(url);

        // Download the page at the given URL.
        String pageContents = downloadPage(verifiedUrl);

        /* If the page was downloaded successfully, retrieve all its
                links and then see if it contains the search string. */
        if (pageContents != null && pageContents.length() > 0)
        {
```

```
                    // Retrieve list of valid links from page.
                    ArrayList    links    =    retrieveLinks(verifiedUrl,
pageContents, crawledList);

                    // Add links to the To Crawl list.
                    toCrawlList.addAll(links);

                    /* Check if search string is present in
                         page, and if so, record a match. */
                    if              (searchStringMatches(pageContents,
searchString,caseSensitive))
                    {
                         addMatch(url);
                    }
               }

          }
     }
}
```

APPENDIX B

This class holds the procedures to analyze the given URL page and then to fill the search forms to extract the content from hidden databases.

```
import java.io.BufferedInputStream;
import java.io.DataInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.PrintStream;
import java.net.URL;
import java.util.ArrayList;
import java.util.List;
import java.util.*;
import java.io.*;

import com.gargoylesoftware.htmlunit.WebClient;
import com.gargoylesoftware.htmlunit.html.HtmlForm;
import com.gargoylesoftware.htmlunit.html.HtmlInput;
import com.gargoylesoftware.htmlunit.html.HtmlPage;
import com.gargoylesoftware.htmlunit.html.HtmlSubmitInput;
import com.gargoylesoftware.htmlunit.html.HtmlTextInput;
import com.gargoylesoftware.htmlunit.html.HtmlTable;
import com.gargoylesoftware.htmlunit.html.HtmlElement;
import com.gargoylesoftware.htmlunit.html.xpath.HtmlUnitXPath;
import com.gargoylesoftware.htmlunit.html.*;

/*
 * Created on June 2, 2007
 * this program will read the inputs from file downloaded by CrawlSystem from
Server and will submit
 * url one by one to formFiller
 */

/**
 * @author Lovekesh
 *
 * TODO To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
public class ExtractData{

        static int found = 0;
        static int countFile = 0;
```

```
        /* file writing */
        static FileOutputStream out; // declare a file output object
    static PrintStream p; // declare a print stream object

    static String SEARCHFIELD;    // = new String("the alchemist");
    static int goodUrls = 0;

        public void processUrls(String fileName) throws Exception{

                out = new FileOutputStream("results.txt");
                p = new PrintStream( out );

                // getting user keyword
                System.out.println("\n Please Enter your Search: ");
                BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
                try {
               SEARCHFIELD = br.readLine();
            } catch (IOException ioe) {System.out.println("unable to recognize it,
please try again");}

                // reading a file

                File f = new File(fileName);
                FileInputStream fis = new FileInputStream(f);
                BufferedInputStream bis = new BufferedInputStream(fis);
                DataInputStream dis = new DataInputStream(bis);
                String urlEntry = null;

                int totalUrls = 0;

                  while ( (urlEntry=dis.readLine()) != null ) {
                        totalUrls++;
                   formRipper(urlEntry);
                   }
                  p.close();

                  printGoodBad(totalUrls);

        }       // main ends

        /* this function will print goodurls, badurls into the result file */
        static void printGoodBad(int totalUrls)
        {
                PrintStream p = null; // declare a print stream object
                try{
```

```
                    FileOutputStream out; // declare a file output object

                out = new FileOutputStream("finalResult.html",true);
                    p = new PrintStream( out );
                    p.println("<br></br>      Total      Urls:"+totalUrls+"      Good
Ones:"+goodUrls);

            }catch(Exception e)
            {System.out.println("found exception in printGoodBad fn");
            }
            p.close();
        }

        /* this function will search for forms and will extract the submit button and
text field
         * for keywords entry
         */

        static void formRipper(String UrlLink)
        {

            try{
                    final WebClient webClient = new WebClient();
                final    URL    url    =    new    URL(UrlLink);    //    overstock,
buy,about,buyitonline .....

                // Get the first page
                final HtmlPage page1 = (HtmlPage)webClient.getPage(url);
                List l = page1.getForms();

                int i=0;
                HtmlInput ip = null;
                HtmlTextInput textField = null;

                while(i < l.size())
                {
                    //p.println("---------------------------------------------"+"\r\n");

                    final HtmlForm form = (HtmlForm)l.get(i);

                    try{

                    ip = (HtmlInput)form.getInputByName("Search");
                    }catch(Exception e)
                        {}              // writing try - catch to keep control over
here
```

```
if(ip == null)          // Search did not work
{
        try{

        ip = (HtmlInput) form.getInputByName("search");
        }catch(Exception e)
                {}
        if(ip == null)  // search did not work
        {
                try{

                ip = (HtmlInput) form.getInputByName("go");
                }catch(Exception e)
                        {}
                if(ip == null)
                {
                        try{

                        ip              =               (HtmlInput)
form.getInputByName("GO");
                        }catch(Exception e)
                                {}
                }
        }
}
if(ip == null)  // means we did not got anything from above ...
then just continue
{
        /* get all inputs and see if there is any image input
with name search or go */

        ArrayList       AllInputs       =       new
ArrayList(form.getInputsByName(""));

        try{
        ip = (HtmlInput)AllInputs.get(0);         //    assuming
we will only get one input
        }
        catch(Exception e){}

        /* searching for image ends */

        /* no hope so lets contine ... */
        if(ip == null)
        {i++;
```

```
                continue;
                }

            }

            // now looking for the text part in the same form

            ArrayList          ListOfText          =          new
ArrayList(form.getInputsByValue(""));

            if(ListOfText.size() == 0)
                 ListOfText                =                new
ArrayList(form.getInputsByName("Zip Box"));

            for(int k=0;k < ListOfText.size(); k++)
                System.out.println(k               +              "
"+ListOfText.get(k).toString());

            if(ListOfText.size() >= 1)
                {

                textField = (HtmlTextInput)ListOfText.get(0);
                try{
                submitThisForm(ip,textField);
                }
                catch(Exception e){}

                break;
                }

            else
            i++;

          }   // while ends



        }catch(Exception e)
            {System.out.println(e);}


    }      // formRipper ends

    /* submitThisForm will take ip (button),textField and form
     * it will fillup the text field of the form and will click on the button
     * the page it will get will be stored in arraylist
```

```
         * @author Lovekesh
         */

        private static void submitThisForm(HtmlInput ip,HtmlTextInput textField)
throws Exception
        {
                try{
                FileOutputStream out; // declare a file output object
            PrintStream p; // declare a print stream object

            out = new FileOutputStream("finalResult.html",true);
                p = new PrintStream( out );

                p.println("<html><body>");

                goodUrls++;

//                  Change the value of the text field
            textField.setValueAttribute(SEARCHFIELD);

            final HtmlPage page2 = (HtmlPage)ip.click();
            System.out.println("\n webpage is ->"+page2.getTitleText());
            p.println("<b> WebPage is:- "+page2.getTitleText()+"</b>");
            p.println("<a   href='"+page2.getWebResponse().getUrl()+"'>Click   Here
To See Original Page</a>");
            p.println();
            findTables(page2,p);

            p.println("</body></html>");
            }
                catch(Exception e){}


            p.close();

           countFile++;
        }

        private   static   void   findTables(HtmlPage   hp,PrintStream   p)   throws
Exception
        {

                ArrayList a = getHtmlElementSByXPath("//table//table",hp);
                //ArrayList a = getHtmlElementSByXPath("//table",hp);// not a good
change, use only for experiment
```

```
            findExactOne(a,p);

            if(a.size() > 0)
                    System.out.println("\n numbers : "+a.size());

    }

    private static HtmlElement getHtmlElementByXPath(final String exp, final
HtmlPage page)
   throws Exception
    {
                    final HtmlUnitXPath xpath = new HtmlUnitXPath(exp);
                    return (HtmlElement) xpath.selectSingleNode(page);
}

    private static ArrayList getHtmlElementSByXPath(final String exp,final
HtmlPage page) throws Exception
    {
            final HtmlUnitXPath xpath = new HtmlUnitXPath(exp);
            return (ArrayList) xpath.selectNodes(page);
    }

    private static ArrayList getHtmlElementSByXPathForm(final String
exp,final HtmlForm form) throws Exception
    {
            final HtmlUnitXPath xpath = new HtmlUnitXPath(exp);
            return (ArrayList) xpath.selectNodes(form);
    }

    private static void findExactOne(ArrayList a, PrintStream p)
    {

            try{
                    for(int i=0;i<a.size();i++)
                    {

                            HtmlTable ht = (HtmlTable)a.get(i);
                            List rows = ht.getRows();

                            for (final Iterator rowIterator = rows.iterator();
rowIterator.hasNext(); )
                            {
                               final HtmlTableRow row = (HtmlTableRow)
rowIterator.next();
                               //System.out.println("Found row");
                               final List cells = row.getCells();
```

```
                                for (final Iterator cellIterator = cells.iterator();
cellIterator.hasNext();)
                                {
                                   final HtmlTableCell cell = (HtmlTableCell)
cellIterator.next();
                                   System.out.println("        Found cell : "+
cell.asText());

                                   //finding if the cell has price tag, then its the one
                                   // we are looking for
                                   if(cell.asText().indexOf("$",0) > 0)        //
returns > 0 if $ is found
                                   {        System.out.println("index            is:
"+cell.asText().indexOf("$",0));

                                            rowExtractor(ht,p);
                                   }
                                }
                             }
                     }

              }catch(Exception e)
              {
                     System.out.println("found in findExactOne :"+e);
              }
       }      // fn ends

       /* this function will extract rows and will stuff it into a file
        * it will extract only those rows that has $ sign in it
        */
       private static void rowExtractor(HtmlTable ht,PrintStream p)
       {

              System.out.println("rowExtractor is been called");
              //p.println("\n\t ----------------------------------------------");
              int rowSize = ht.getRows().size();
              List rows = ht.getRows();
              p.println("<p>");

              for(int i=0; i<rowSize; i++)
              {
                     HtmlTableRow htr = (HtmlTableRow)rows.get(i);
                     p.println("<i>Entry # "+i+"</i>");

                     for(int j=0; j<htr.getCells().size();j++)
                     {
```

```
                    p.println("<div align='left'>");
                    p.println(ht.getCellAt(i,j).asText());
                    p.println("</div>");

                }
            }
            p.println("</p>");

        }   // fn ends

        private static void rowCellExtractor(HtmlTableCell htc,PrintStream p)
        {

            p.println(htc.asXml());

        }   // fn ends

}       // class ends
```

APPENDIX C

This class defines the procedures on client side to interact with RMI server interface to download and upload files.

```
import java.io.*;
import java.rmi.*;

/*
 * Created on June 14, 2007
 * CrawlSystem is responsible to download URL file from the server and
 * to upload the results file having information from hidden web to Crawl Manager
 */

/**
 * @author Lovekesh
 *
 * TODO To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
public class CrawlSystem{
  public static void main(String argv[]) {
    if(argv.length != 2) {
      System.out.println("Usage: java crawlSystem fileName machineName");
      System.exit(0);
    }

    //Download file containing set of URLs from Crawl Manager
    // These all URLs are from single specific domain only.
    try {
      String name = "//" + argv[1] + "/CrawlManager";
      crawlManagerInterface fi = (crawlManagerInterface) Naming.lookup(name);
      byte[] filedata = fi.downloadFile(argv[0]);
      File file = new File(argv[0]);
      BufferedOutputStream output = new BufferedOutputStream(new
FileOutputStream(file.getName()));
      output.write(filedata,0,filedata.length);
      output.flush();
      output.close();
    } catch(Exception e) {
      System.err.println("CrawlSystem: exception in dowloading file form server,
"+ e.getMessage());
      e.printStackTrace();
    }

    //Each URL from the downloaded file is processed for content extraction...
```

```
    try{
        ExtractData getDataObj = new ExtractData();
        getDataObj.processUrls(argv[0]);
    } catch(Exception e) {
      System.err.println("Extract Data: "+ e.getMessage());
      e.printStackTrace();
    }

    // After processing the set of URLs file having the search results are uploaded
to CrawlManager.
    try{
              String name = "//" + argv[1] + "/CrawlManager";
              crawlManagerInterface     fi     =     (crawlManagerInterface)
Naming.lookup(name);
        File file = new File("finalResult.html");
        byte buffer[] = new byte[(int)file.length()];
        BufferedInputStream     input     =     new     BufferedInputStream(new
FileInputStream(file));
        input.read(buffer,0,buffer.length);
        fi.uploadFile(buffer);
        input.close();
    }   catch(Exception e) {
        System.err.println("CrawlSystem exception in uploading result file to
Server: "+ e.getMessage());
        e.printStackTrace();
    }
  }
}
```

APPENDIX D

This class implements the methods declared in server side interface with which clients interacts to download and to upload files. It also takes care of synchronization of shared resource on the server side.

```
import java.io.*;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

/*
 * Created on June 16, 2007
 * CrawlManagerImpl extends CrawlManagerInterface
 * CrawlManagerImpl implements all the methods declared in
CrawlManagerInterface
 * CrawlManagerImpl defines method to let CrawlSystem download URLs file and
also
 * defines a method to let CrawlSystem upload the file having search results.
 * As JAVA RMI platform itself provides server-side multithreading,
 * some methods are declared synchornized to control access to shared
resource.
 */

/**
 * @author Lovekesh
 *
 * TODO To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Style - Code Templates
 */

public class crawlManagerImpl extends UnicastRemoteObject
  implements crawlManagerInterface {

   private String name;
   private long before,after,executionTime;

   public crawlManagerImpl(String s) throws RemoteException{
      super(1090); // 1090 is specified port for the remote objects....
      name = s;
   }

          // This method is used by clients to download file having set of URLS........
   public byte[] downloadFile(String fileName){
      try {
            before = System.currentTimeMillis();
         File file = new File(fileName);
```

```
        byte buffer[] = new byte[(int)file.length()];
        BufferedInputStream input = new BufferedInputStream(new
FileInputStream(fileName));
        input.read(buffer,0,buffer.length);
        input.close();
        System.out.println("File containing set of URLS has been downloaded to
client");
        return(buffer);
      } catch(Exception e){
        System.out.println("crawlManager.downloadFile(string fileName):
"+e.getMessage());
        e.printStackTrace();
        return(null);
      }
   }

   // This method uploads the result file from the client.....
   // As RMI itself forces server-side multithreading, the method must be
synchronized as Resultfile on
   // server is shared resource.
   synchronized public void uploadFile(byte[] fileData) {
          try {
        File file = new File("finalResult.html");
        BufferedOutputStream output = new BufferedOutputStream(new
FileOutputStream(file.getName(),true));
        output.write(fileData,0,fileData.length);
        output.flush();
        output.close();
        System.out.println("File containing result has been uploaded from client");
        after = System.currentTimeMillis();
        executionTime = (after-before)/1000;
        System.out.println("Execution time in secs : " + executionTime);
     }catch(Exception e){
        System.out.println("crawlManagerImpl.uploadFile(byte[] fileData):
"+e.getMessage());
        e.printStackTrace();
           }
  }

}
```

APPENDIX - HtmlUnit

HtmlUnit is a java unit testing framework for testing and processing Web based applications. HtmlUnit is very similar in concept to httpunit (http://sourceforge.net/projects/httpunit) but is very different in implementation. Which one is better for you depends on how you like to write your tests. HttpUnit models the http protocol so you deal with request and response objects. HtmlUnit on the other hand, models the returned document so that you deal with pages and forms and tables.

HtmlUnit was originally written by Mike Bowler of Gargoyle Software and released under an apache style license. Since then, it has received many contributions from other developers and would not be where it is today without their assistance.

HtmlUnit is not a generic unit testing framework. It is specifically a way to simulate a browser for testing purposes and is intended to be used within another testing framework such as JUnit For more information, please visit http://htmlunit.sourceforge.net/.