STUDY & IMPLEMENTATION OF PARALLEL WEB CRAWLER

Submitted For Partial Fulfillment for award of

Bachelor of Technology

Degree in

Computer Science And Engineering

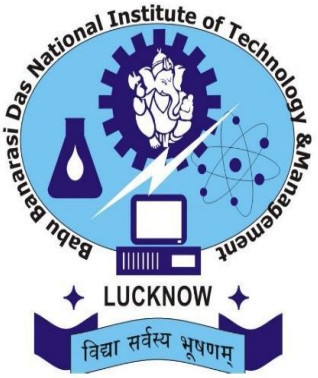
By

Ruchin Agarwal 1005410091

Vanshika Nigam 1005410117

Shivangi Rai 1005410105

Parul 1005410066



BABU BANARASI DAS
NATIONAL INSTITUTE OF TECHNOLOGY & MANAGEMENT

LUCKNOW,
UTTAR PRADESH, INDIA

(2013 – 2014)

CERTIFICATE

Certified that **Ruchin Agarwal (University Roll No. 1005410091), Vanshika Nigam (University Roll No. 1005410117), Shivangi Rai (University Roll No. 1005410105) and Parul (University Roll No. 1005410066)** has carried out the Project work presented in this report entitled **" Study & Implementation of Parallel Web Crawler "** for partial fulfilment for award of **Bachelor of Technology** Degree in **Computer Science & Engineering,** Final Year from **"Babu Banarasi Das National Technology & Management, Lucknow"** under my supervision. The report embodies result of original work and studies carried out by students themselves and the contents of the project do not form the basis for the award of any other degree to the candidates or to anybody else.

Mrs. Priyanka Dhuliya
SUPERVISOR
Sr. Lecturer,
Computer Science & Engineering
BBDNITM, Lucknow.

Dr. Shubha Mishra
SUPERVISOR
Associate Professor and Head,
Computer Science & Engineering
BBDNITM, Lucknow.

Date**:**

ACKNOWLEDGEMENT

It gives us a great sense of pleasure to present the report of the B.Tech. Project undertaken during B.Tech.Final Year. We owe special debt of gratitude to **Dr. Shubha Mishra, Associate Professor & Head, Department of Computer Science & Engineering,** BabuBanarasi Das National Institute of Technology & Management, for her constant support and guidance throughout the course of our work. Her sincerity, thoroughness and perseverance have been a constant source of inspiration for us. It is only her cognizant efforts that her endeavours have seen light of the day.

We also take the opportunity to acknowledge the contribution of **Mrs. PriyankaDhuliya**, **Senior Lecturer, Department of Computer Science & Engineering,** BabuBanarasi Das National Institute of Technology & Management, for her full support and assistance during the development of the project.

We also do not like to miss the opportunity to acknowledge the contribution of all faculty members of the department for their kind assistance and contribution during the development of our project. Last but not the least, we acknowledge our friends for their contribution in the completion of the project.

Ruchin Agarwal Vanshika Nigam Shivangi Rai Parul

APPROVAL SHEET

Project Title : Study And Implementation of Parallel Web Crawler

Name of Students **Signature**

1.Vanshika Nigam

2.Shivangi Rai

3.Parul

4.Ruchin Agarwal

Name of the Project Guide : Mrs. Priyanka Dhuliya

Signature of the Project Guide:

TABLE OF CONTENTS

LIST OF FIGURES

1.INTRODUCTION

The number of web pages is increasing by millions and trillions around the world. To make searching much easier for users, web search engines came into existence. Every search engine maintains a central repository or databases of HTML documents in indexed form.Whenever auser query comes, searching is performed within that database of indexed web pages.The size of repository of every search engine cannot accommodate each and every page available on the WWW. Soitisdesired that only the most relevant pages are stored in the database so as to increase the efficiency of search engines.To store most relevant pages from the World Wide Web, a suitable and better approach has to be followed by the search engines. This database of HTML documents is maintained by special software .The software that traverses web for capturing pages is called "Crawlers"or "Spiders".

In this project we study the challenges and issues faced in implementing an effective Web Crawler. A crawler is a program that retrieves and store pages from the Web, commonly for a web search engine. A crawler often has to download hundreds or millions of Pages in a short period of time and has to constantly monitor and refresh the downloaded pages. In addition, the crawler should avoid putting too much pressure on the visited Web sites and the crawler's local network, because they are intrinsically shared resources. Towards that goal, first we identify popular definition for the "importance" of pages and propose simple algorithms that can identify important pages at the early stage of a crawl. We then explore how we can parallelize a crawling process to maximize the download rate while minimizing the overhead from parallelization. Finally, we experimentally study how Web pages change over time and propose an optimal page refresh policy that maximizes the "freshness" of the retrieved pages.

1.1 Web Crawler

We use software known as "web crawlers" to discover publicly available webpages. The most well-known crawler is called "Googlebot." Crawlers look at webpages and follow links on those pages, much like you would if you were browsing content on the web. They go from link to link and bring data about those webpages back to Google's servers.

The crawl process begins with a list of web addresses from past crawls and sitemaps provided by website owners. As our crawlers visit these websites, they look for links for other pages to visit. The software pays special attention to new sites, changes to existing sites and dead links.

Computer programs determine which sites to crawl, how often, and how many pages to fetch from each site. Google doesn't accept payment to crawl a site more frequently for our web search results. We care more about having the best possible results because in the long run that's what's best for users and, therefore, our business.

So basically a Web crawler starts with a list of URLs to visit, called the *seeds*. As the crawler visits these URLs, it identifies all the hyperlinks in the page and adds them to the list of URLs to visit, called the *crawl frontier*. URLs from the frontier are recursively visited according to a set of policies.
The large volume implies that the crawler can only download a limited number of the Web pages within a given time, so it needs to prioritize its downloads. The high rate of change implies that the pages might have already been updated or even deleted.

The number of possible crawlable URLs being generated by server-side software has also made it difficult for web crawlers to avoid retrieving duplicate content. Endless combinations of HTTP GET(URL-based) parameters exist, of which only a small selection will actually return unique content.

1.2 Crawling Policies

- The behaviour of a Web crawler is the outcome of a combination of policies:

a *selection policy* that states which pages to download. As a crawler always downloads just a fraction of the Web pages, it is highly desirable that the downloaded fraction contains the most relevant pages and not just a random sample of the Web.
This requires a metric of importance for prioritizing Web pages. The importance of a page is a function of its intrinsic quality, its popularity in terms of links or visits, and even of its URL (the latter is the case of vertical search engines restricted to a single top-level domain, or search engines restricted to a fixed Web site). Designing a good selection policy has an added difficulty: it must work with partial information, as the complete set of Web pages is not known during crawling.

- **a *re-visit policy*** that states when to check for changes to the pages.

- The Web has a very dynamic nature, and crawling a fraction of the Web can take weeks or months. By the time a Web crawler has finished its crawl, many events could have happened, including creations, updates and deletions.
- From the search engine's point of view, there is a cost associated with not detecting an event, and thus having an outdated copy of a resource. The most-used cost functions are freshness and age.[26]
- **Freshness**: This is a binary measure that indicates whether the local copy is accurate or not. The freshness of a page *p* in the repository at time *t* is defined as:
- $$F_p(t) = \begin{cases} 1 & \text{if } p \text{ is equal to the local copy at time } t \\ 0 & \text{otherwise} \end{cases}$$
- **Age**: This is a measure that indicates how outdated the local copy is. The age of a page *p* in the repository, at time *t* is defined as:
- $$A_p(t) = \begin{cases} 0 & \text{if } p \text{ is not modified at time } t \\ t - \text{modification time of } p & \text{otherwise} \end{cases}$$

a *politeness policy* that states how to avoid overloading Web sites. Crawlers can retrieve data much quicker and in greater depth than human searchers, so they can have a crippling impact on the performance of a site. Needless to say, if a single crawler is performing multiple requests per second and/or downloading large files, a server would have a hard time keeping up with requests from multiple crawlers.

a *parallelization policy* that states how to coordinate distributed web crawlers.
A parallel crawler is a crawler that runs multiple processes in parallel. The goal is to maximize the download rate while minimizing the overhead from parallelization and to avoid repeated downloads of the same page. To avoid downloading the same page more than once, the

crawling system requires a policy for assigning the new URLs discovered during the crawling process, as the same URL can be found by two different crawling processes.

1.3TYPES OF CRAWLING

Repetitive Crawling- Once page has been crawled,some systems require the process to be repeated periodically so that indexes are kept updated which may be achieved by calling a second crawler in parallel,to overcome this problem,we must constantly update the index lists.

Targeted Crawling- The main objective is to retrieve the greatest number of pages relating to a particular subject by using the "Minimum Bandwidth". Most search enginesuse crawling process heuristics in order to target certain type of page on specific topic.

Random Walks & Sample-They focus on the effect of random walks on web graphs or modified versions of these graphs via sampling to estimate the size of documents.

***Deep web Crawling* –** The data which is present in the database may be downloaded through the medium of appropriate request or forms.This Deep Web name is given to this category of data.

1.3 Parallel Crawler

- As the size of the web grows, it becomes more difficult to retrieve the whole or a significant portion of the web using a single process.
- It becomes imperative to parallelize a crawling process, in order to finish downloading pages in a reasonable amount of time.
- We refer to this type of crawler as a *parallel crawle*r.
- The main goal in designing a parallel crawler, is to maximize its performance (Download rate) & minimize the overhead from parallelization.

2.Literature Survey

[1] In this paper we study how we can design an effective parallel crawler. As the size of the Web grows, it becomes imperative to parallelize a crawling process, in order to finish downloading pages in a reasonable amount of time.
A crawler starts off by placing an initial set of URLs, *S*0, in a queue, where all URLs to be retrieved are kept and prioritized. From this queue, the crawler gets a URL (in some order), downloads the page, extracts any URLs in the downloaded page, and puts the new URLs in the queue. This process is repeated until the crawler decides to stop. Collected pages are later used for other applications, such as a Web search engine or a Web cache. As the size of the Web grows, it becomes more difficult to retrieve the whole or a significant portion of the Web using a single process. Therefore, many search engines often run multiple processes in parallel to perform the above task, so that download rate is maximized. We refer to this type of crawler as a *parallel crawler*.

We study how we should design a parallel crawler, so that we can maximize its performance (e.g., download rate) while minimizing the overhead from parallelization. We believe many existing search engines already use some sort of parallelization, but there has been little scientific research conducted on this topic. Thus, little has been known on the tradeoffs among various design choices for a parallel crawler. In particular, we believe the following issues make the study of a parallel crawler challenging and interesting:

1. Overlap
2. Quality
3. Communication bandwidth

In this paper the web has more than 350 million pages and is growing in the tune of one million pages per day. Such enormous growth and flux necessitates the creation of highly efficient crawling system Research is being carried out in the following areas:

- To develop strategies to crawl only the relevant pages
- To design architectures for paralle crawlers
- Restructuring of hypertext documents

We proposed augmentation to the hypertext documents so that they become suitable for downloading by parallel crawlers. The augmentations do not affect the current structure of hypertext system. This paper discusses hypertext documents, the proposed augmentations, and the design of an architecture of a parallel crawler based on the augmented hypertext documents (PARCAHYD). The implementation of this crawler in Java is in progress.

Crawling the Hidden Web

Abstract

Current-day crawlers retrieve content from the publicly indexable Web, i.e., the set of web pages reachable purely by following hypertext links, ignoring search forms and pages that require authorization or prior registration. In particular, they ignore the tremendous amount of high quality content ``hidden" behind search forms, in large searchable electronic databases. Our work provides a framework for addressing the problem of extracting content from this hidden Web. At Stanford, we have built a task-specific hidden Web crawler called the Hidden Web Exposer (HiWE). In this poster, we describe the architecture of HiWE and outline some of the novel techniques that went into its design.

Keywords: Crawling, Hidden Web, Content extraction, HTML Forms, HiWE

1. Introduction

A number of recent studies [1,2,3] have noted that a tremendous amount of content on the Web is *dynamic*. However, since current-day crawlers only crawl the *publicly indexable Web* [2], much of this dynamic content remains inaccessible for searching, indexing, and analysis. The hidden Web is particularly important, as organizations with large amounts of *high-quality* information (e.g., the Census Bureau, Patents and Trademarks Office, News media companies) are placing their content online, by building Web query front-ends to their databases.

Crawling the hidden Web is a very challenging problem for two fundamental reasons: (1) scale (a recent study [1] estimates the size of the hidden Web to be about 500 times the size of the publicly indexable Web) and (2) the need for crawlers to handle search interfaces designed primarily for humans.

We address these challenges by adopting a *task-specific human-assisted* approach to crawling. Specifically, we selectively crawl portions of the hidden Web, extracting content based on the requirements of a particular application or task. We also provide a framework that allows the human expert to customize and assist the crawler in its activity.

2. HiWE

At Stanford, we have built a task-specific hidden Web crawler called the Hidden Web Exposer (HiWE). **Figure 1 illustrates HiWE's architecture and execution flow.**

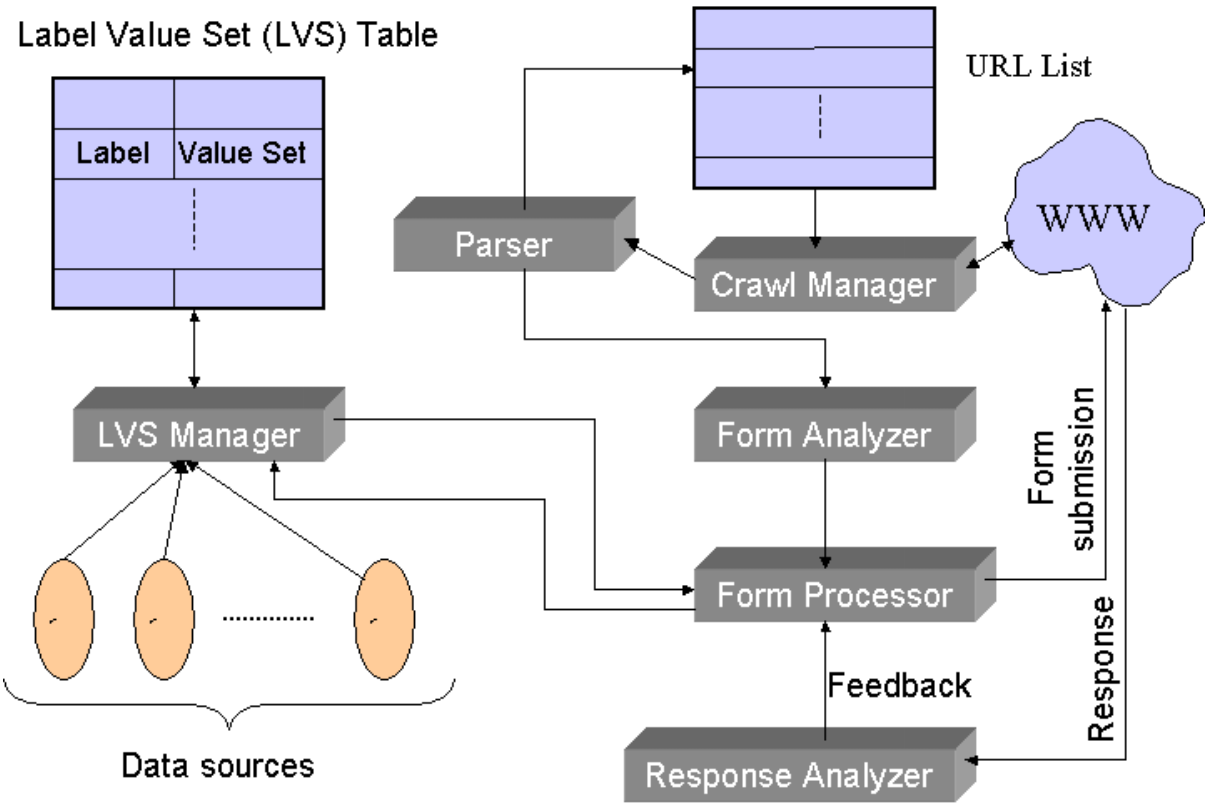


Figure 1: HiWE Architecture

Since search forms are the entry-points into the hidden Web, HiWE is designed to automatically process, analyze, and submit forms, using an internal model of forms and form submissions. This model treats forms as a set of*(element, domain)* pairs. A form element can be any one of the standard input objects such as selection lists, text boxes or radio buttons. Each form element is associated with a finite or infinite domain and a text *label* that semantically describes the element (see Figure 2).

The values used to fill out forms are maintained in a special table called the LVS (Label Value Set) table (Figure 1). Each entry in the LVS table consists of a label and an associated *fuzzy/graded set* of values (e.g., Label = ``State'' and value set = { (``California'', 0.8), (``New York'', 0.7) }). The weight associated with a value represents the crawler's estimate of how effective it would be, to assign that value to a form element with the corresponding label. Methods to populate the LVS table and assign weights are described in detail in [4].

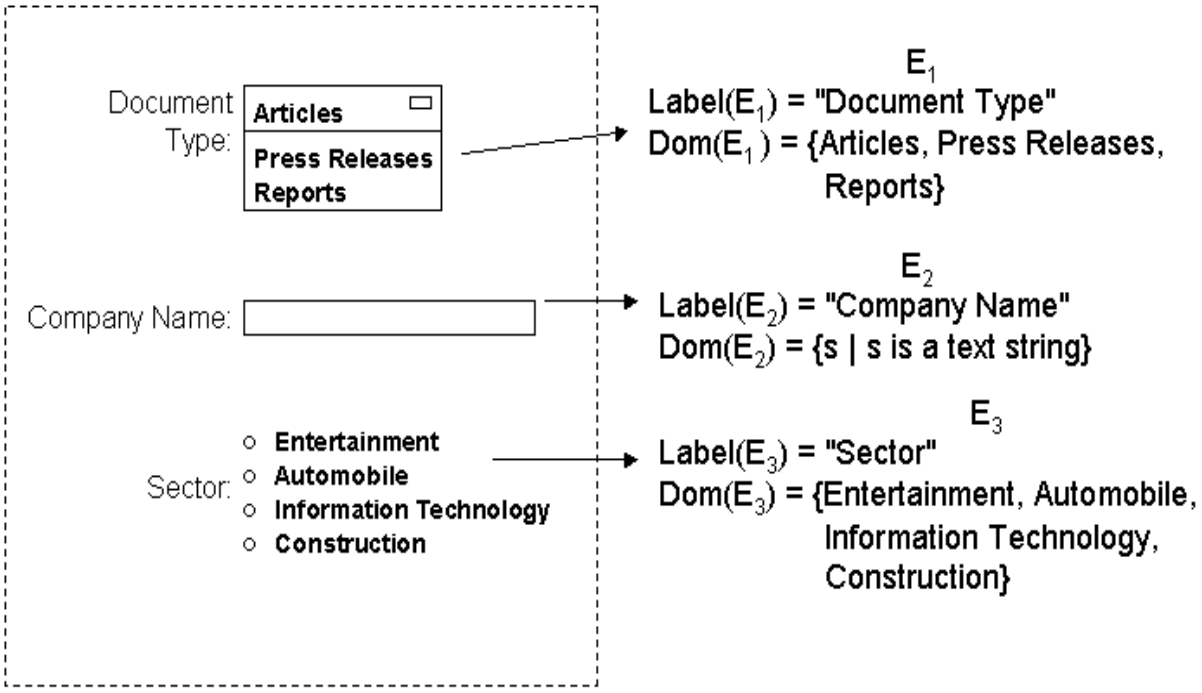


Figure 2: Sample labeled form

The basic actions of HiWE (fetching pages, parsing and extracting URLs, and adding the URLs to a URL list) are similar to those of traditional crawlers. However, whereas the latter ignore forms, HiWE performs the following sequence of actions for each form on a page:

1. ***Form Analysis:*** **Parse and process the form to build an internal representation based on the above model.**
2. ***Value assignment and ranking:*** **Use approximate string matching between the form labels and the labels in the LVS table to generate a set of candidate value assignments. (A *value assignment* is an assignment of a value to each element of a form.) Use fuzzy aggregation functions to combine individual weights into weights for value assignments and use these weights for ranking the candidate assignments.**
3. ***Form Submission:*** **Use the top ``N'' value assignments to repeatedly fill out and submit the form.**
4. ***Response Analysis and Navigation:*** **Analyze the response pages (i.e., the pages received in response to form submissions) to check if the submission yielded valid search results. Use this feedback to tune the value assignments in Step 2. Crawl the hypertext links in the response page to some pre-specified depth.**

3. Layout-based Extraction

Search forms and response pages are designed for human consumption. As a result, it is a significant task for a crawler to process and extract semantically useful information (e.g, the labels of form elements) from such pages.

As part of form and response analysis, HiWE uses a ``Layout-based Information Extraction Technique (LITE)" to achieve this task. LITE is based on the principle that semantic information can be robustly extracted by exploiting visual cues (i.e., using information about how various objects are laid out on a page). For example, the label associated with a given form element is most likely to be the piece of text or phrase that is visually (not necessarily textually) closest to the form widget, when the page is displayed by the browser. Hence, HiWE employs a custom layout engine that approximately lays out form pages and response pages and can be used to compute visual distances between different elements in a page. Our preliminary experiments indicate a 93% success rate in using LITE to correctly identify labels for form elements.

4. Conclusion

We have addressed the problem of crawling and extracting content from the ``hidden Web", the portion of the Web hidden behind searchable HTML forms. Due to the tremendous size of the hidden Web, comprehensive coverage is very difficult, and possibly less useful, than task-specific crawling. Our work exploits this specificity to design a configurable crawler that can benefit from knowledge of the particular application domain.

Our initial experiments [4] indicate that human-assisted crawling of the hidden Web is feasible. They also indicate that LITE is a powerful method for extracting semantic information from search forms and response pages.

SOFTWARE DEVELOPMENT LIFE CYCLE

The **systems development life cycle (SDLC)**, also referred to as the **application development life-cycle**, is a term used insystems engineering, information systems and software engineering to describe a process for planning, creating, testing, and deploying an information system.[1] The systems development life-cycle concept applies to a range of hardware and software configurations, as a system can be composed of hardware only, software only, or a combination OF both.[2]
A systems development life cycle is composed of a number of clearly defined and distinct work phases which are used by systems engineers and systems developers to plan for, design, build, test, and deliver information systems. Like anything that is manufactured on an assembly line, an SDLC aims to produce high quality systems that meet or exceed customer expectations, based on customer requirements, by delivering systems which move through each clearly defined phase, within scheduled time-frames and cost estimates.[3] Computer systems are complex and often (especially with the recent rise of service-oriented architecture) link multiple traditional systems potentially supplied by different software vendors. To manage this level of complexity, a number of SDLC models or methodologies have been created, such as "waterfall"; "spiral"; "Agile software development"; "rapid prototyping"; "incremental"; and "synchronize and stabilize".[4]

- **Preliminary analysis**: The objective of phase 1 is to conduct a preliminary analysis, propose alternative solutions, describe costs and benefits and submit a preliminary plan with recommendations.

    - **Systems analysis, requirements definition**: Defines project goals into defined functions and operation of the intended application. Analyzes end-user information needs.

    - **Systems design**: Describes desired features and operations in detail, including screen layouts, business rules, process diagrams, pseudocode and other documentation.

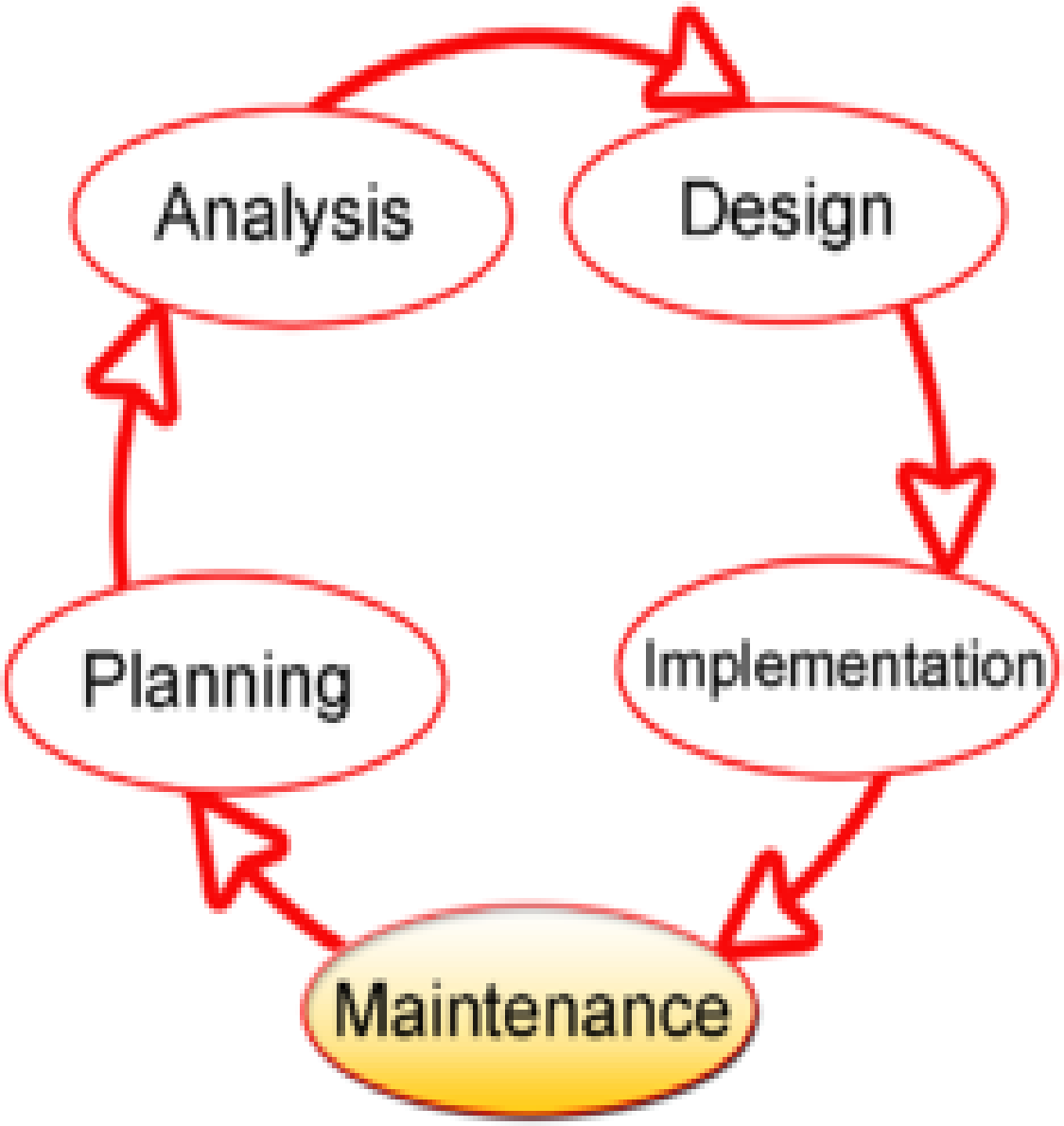
    - **iMPLEMENTATION(CODING)**: The real code is written here.

    - **Integration and testing**: Brings all the pieces together into a special testing environment, then checks for errors, bugs and interoperability.

    - **Acceptance, installation, deployment**: The final stage of initial development, where the software is put into production and runs actual business.

    - **Maintenance**: During the maintenance stage of the SDLC, the system is assessed to ensure it does not become obsolete. This is also where changes are made to initial software. It involves continuous evaluation of the system in terms of its performance.

FIGURE 3 : SOFTWARE DEVELOPMENT LIFE CYCLE



PHASE 1:FeasibilityStudy

Feasibility study tries to determine whether agiven solution work or not. It's main objectivies not to solve the problem, but to acquire its scope.It focuses on the following:

- Meet the requirements
- Best utilization of the available resources
- Develop a cost effective system
- Develop a technically feasible system

There are three aspects of the feasibility study:

- Technical feasibility
- Economic feasibility
- Operational feasibility

Prerequisites of a crawling system

The main requirements for any large scale crawling systemare as follows :

Flexibilty : "Our system should be suitable for various scenarios"

High Performance : The system should be scalable with a minimum of thousand pages to millions so the quality and disk assurance are crucial for maintaining high performance.

Fault Tolerance : The first goal is to identify the problems like invalid HTML and having good communicative protocols.Secondly the system should be persistent(Eg: restart after failure). Since the crawling process takes about 2 to 5 days.

Maintainability and Configurability : There should be appropriate interfacefor the monitoring for crawling process including download speed, statistics and the administrator can adjust the speed of crawler.

PHASE 2:Requirement Analysis

- This is the first phase of SDLC (Software Development Life Cycle).
- In this phase the preliminary analysis of the software is done.
- The nature , scope and objective of the problem is assessed.

The requirements of the problem are also analyzed.

4.2.1 SCOPE OF THE PROJECT

The World Wide Web has a broad coverage. Various sites enter the market on a daily basis. For keeping up with the business search engines are required so as to secure steady traffic.

Search is the most important tool in the present world but the scope of the searching has largely increased now. The web not only contains the surface data but also the hidden data therefore using a simple search engine will not solve the problem. We require a crawler that can produce information from the web.

The complex nature at the web level forms the scope of our project so that the focus is on producing the contents not reachable by the simple crawler to the user efficiently and correctly.

4.2.2 OBJECTIVES

- To develop the parallel web Crawler for gathering information from the web by using user input.
- To apply composing techniques that links similar types of information from the database and generates relevant query results.
- To design a graphical user interface for users from where they can submit their queries for desired information.
- To incorporate multi-threaded functionality within the crawler. To design a fetcher that downloads the pages from web efficiently.

4.2.3 Hardware & Software Requirement

Software Requirement For Development

- Microsoft Windows XP/Vista/7 OperatingSystem
- NetBeansIDE 6.1 or higher
- JDK5.0 orhigher
- MySQLServer 5.0orhigher

Hardware Requirement For Run Time

- Intel or AMD 2.4 GHzprocessor (Recommended).
- 1 GBRAM (Recommended).
- 100 GBhard Disk space(Recommended).
- Keyboardand Mouse
- TFT/CRT Monitor

20 GBDatabasespace (Recommended).

Networkrequirement:

100 Mbps LAN connection (Recommended)

PHASE 3:DESIGNING

- In Designing stage of SDLC we describe the desired features and operations in detail, including screen layouts, process diagrams, pseudo code and other documentation.

4.3.1 Proposed System

This proposed architecture of parallel crawler is based on multithreading and intelligent database analysis. Crawling on this base makes the task more effective in terms of relevancy and load sharing. The fig.7.1 [2] shows the proposed architecture of simple parallel crawler or floe of crawling process. The users interact through the graphical user interface by providing inputs like seed URL and maximum limit of crawling. The seed URL is first stored in the seed Queue, which contains the recently visited URLs. Then this URL dispatched to Link Extractor which extracts the links from this page and forwards the extracted links to link downloader and link analyzer, and finally the crawling process proceeds.

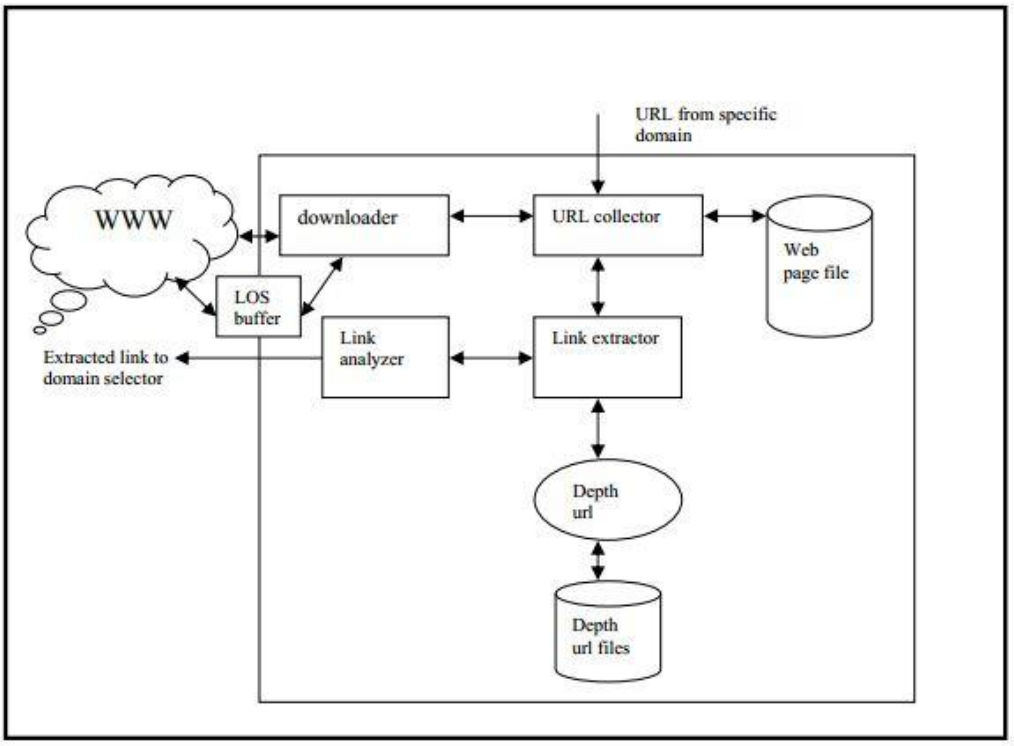


Figure 4 : Flow of CrawlingProcess[2]

A Novel architecture of parallel crawler which based on intelligent domain specific crawling is being proposed. Crawling on this base makes the task more effective in terms of relevancy and load sharing .The fig.7.1 [2], shows the proposed architecture of domain specific intelligent parallel crawler. The users interact through the search engine interface, for the specific information in the form of 'keyword'. The seed URL is first searched in the Web Cache, which contains the recently visited URLs; else the search is made in the repository of the mother server, which maintains the database for the keyword and the corresponding URL's as a tree of nodes where each node is a domain. The URL dispatcher dispatches the seed URL to the concerned DNS Queues through the URL distributor, and finally the crawling process proceeds. On the other hand if the search for the URL for the specific key word fails (i.e. the keyword has not been visited as yet) the search/insert technique is followed, where the keyword is added to the database and its corresponding URL is added in future when encountered. The architecture has number of domain specific queues, for the various domains like, .edu, .org, .ac, .com etc. The URLs for these queues are sent to the respective Crawl Workers. This leads to the load sharing by the parallel crawlers on the basis of specific domains.

4.3.2 ARCHITECTURE OF A PARALLEL WEB CRAWLER

- A parallel crawler consists of multiple crawling processes: "c-proc".
- C-proc performs single-process crawler tasks:
  - Downloads pages from the web.
  - Stores downloaded pages locally.
  - Extracts URLs from downloaded pages and follows links.
- Depending on how the c-proc's split the download task, some of the extracted links may be sent to other c-proc's.e

FIGURE 5 : DOWNLOADING OF PAGES FROM WEB

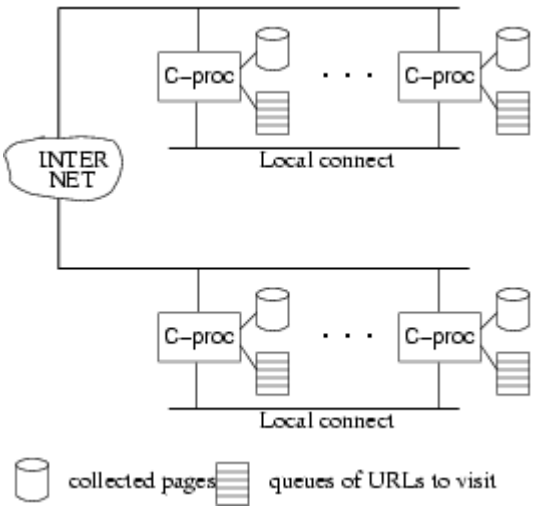
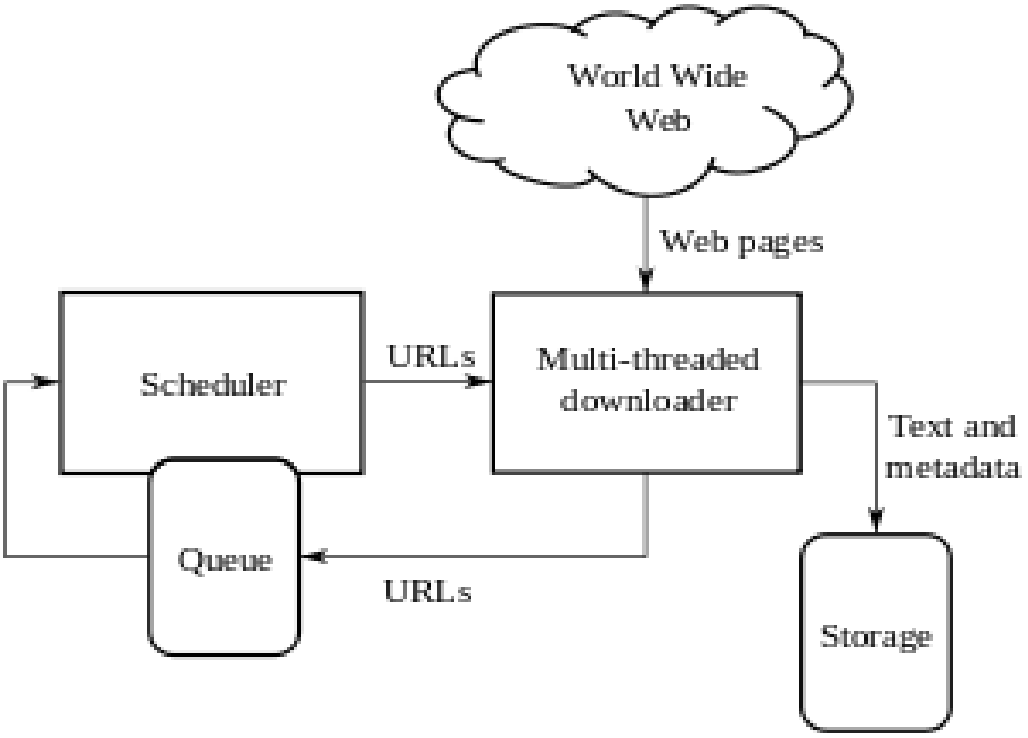


FIGURE 6 : PROCESSING OF URLs



4.3.3 ALGORITHMS

the basics:

1. A list of unvisited URLs - seed this with one or more starting pages
2. A list of visited URLs - so you don't go around in circles
3. A set of rules for URLs you're not interesting - so you don't index the whole Internet
4. Put these stored in a database is necessary, since crawler may stop and need to restart with the same place without losing state.

Algorithm is as follows.

```
while(list of unvisited URLs is not empty){
    take URL from list
    fetch content
    record whatever it is you want to about the content
if content is HTML{
        parse out URLs from links
foreachURL{
if it matches your rules
            and it's not already in either the visited or unvisited list
            add it to the unvisited list
        }
    }
}
```

A more detailed version of the algorithm is:

Crawler ()

Begin

While (URL set is not empty)

Begin

Take a URL from the set of seed URLs;

Determine the IP address for the host name;

Determine the protocol of underlying host like http, ftp, gopher etc.;

Based on the protocol of the host, download the document;

Identify the document format like doc, html, or pdf etc.

Check whether the document has already been downloaded or not;

If the document is fresh one

Then

Read it and extract the links or references to the other sites from that documents;

Else

Continue;

Convert the URL links into their absolute URL equivalents;

Add the URLs to set of seed URLs;

End**;**

FIGURE 7 : PROCESS FLOW OF PARALLEL WEB CRAWLER

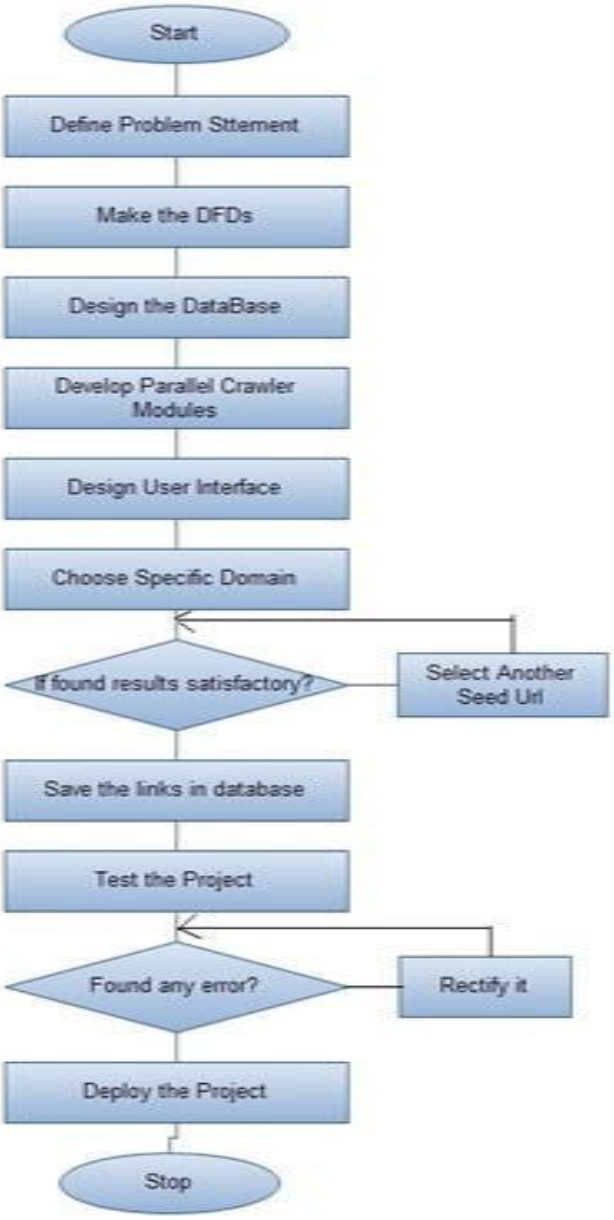


FIGURE 8 : DATA FLOW DAIGRAM 1

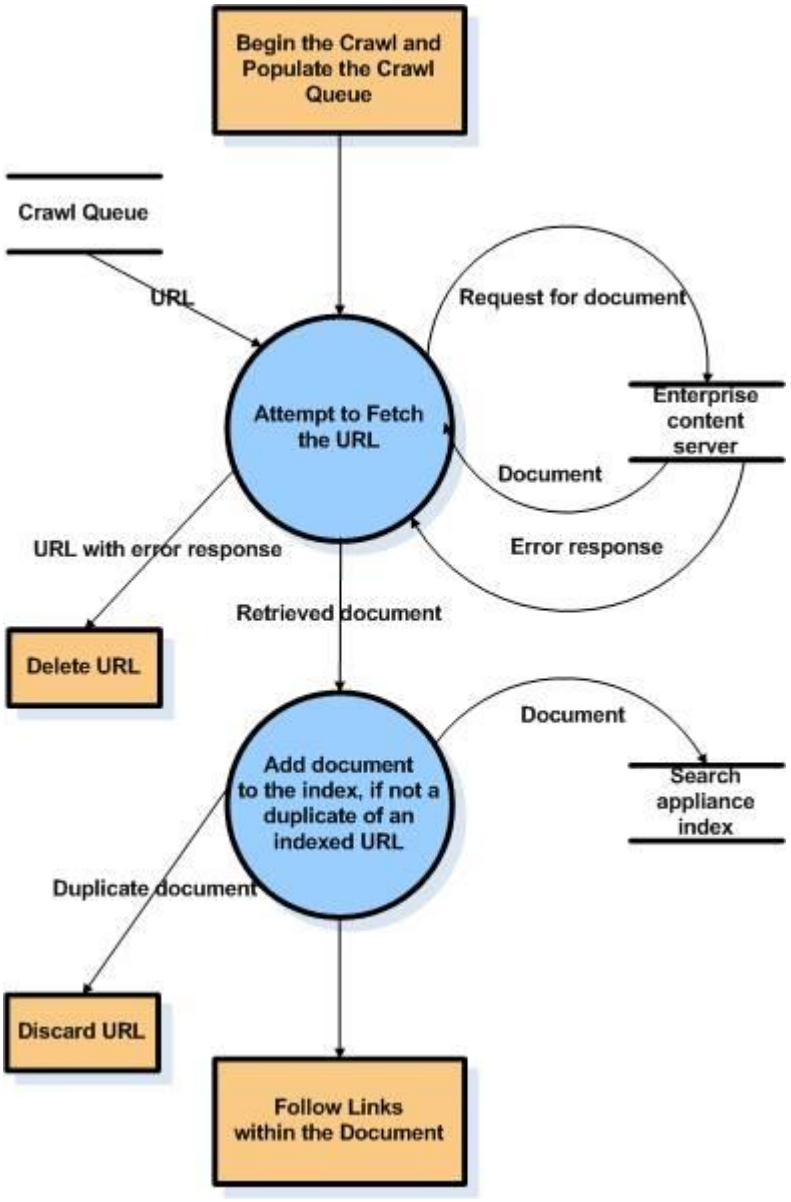


FIGURE 9 : DATA FLOW DIAGRAM 2

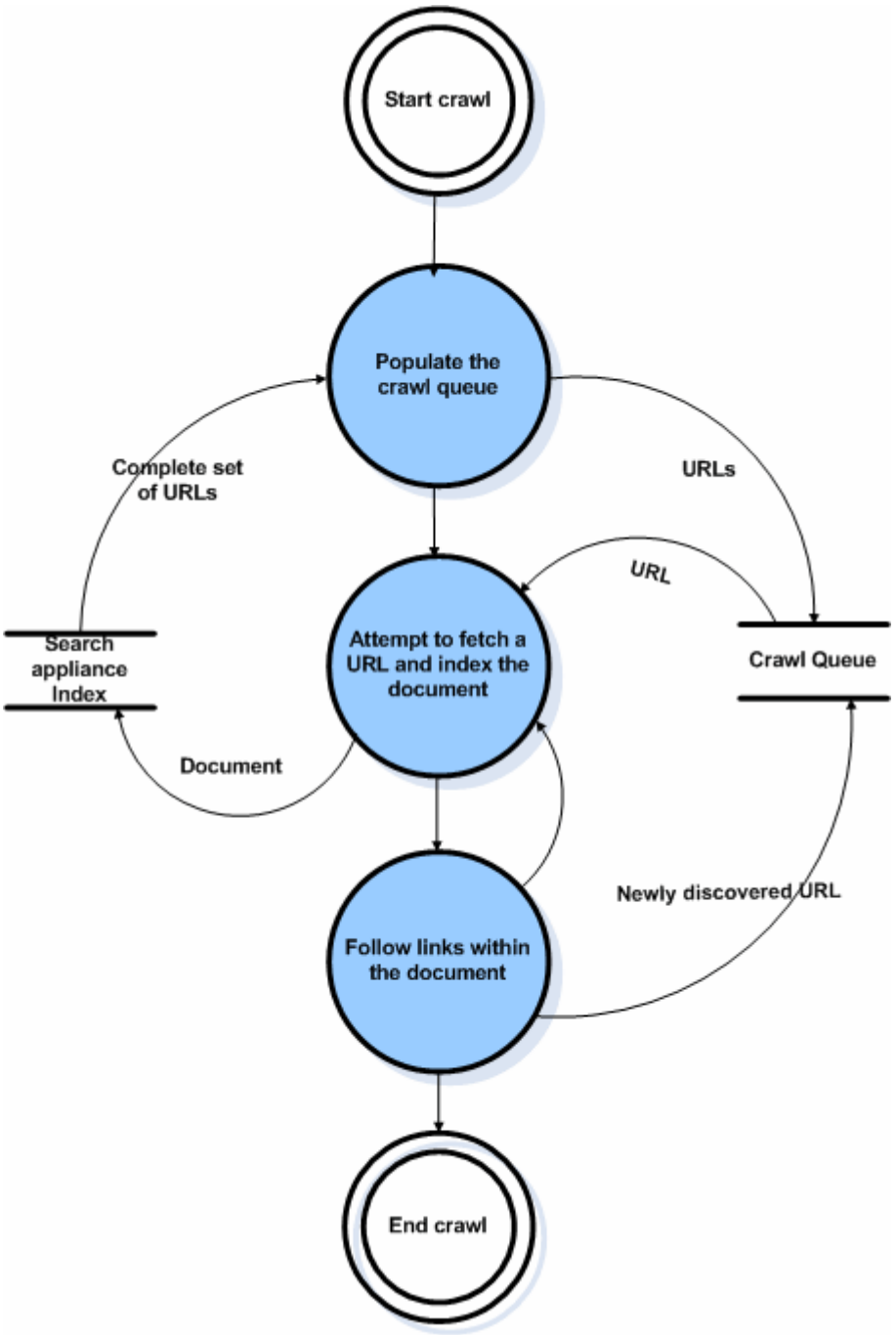


FIGURE 10 : FLOW DIAGRAM

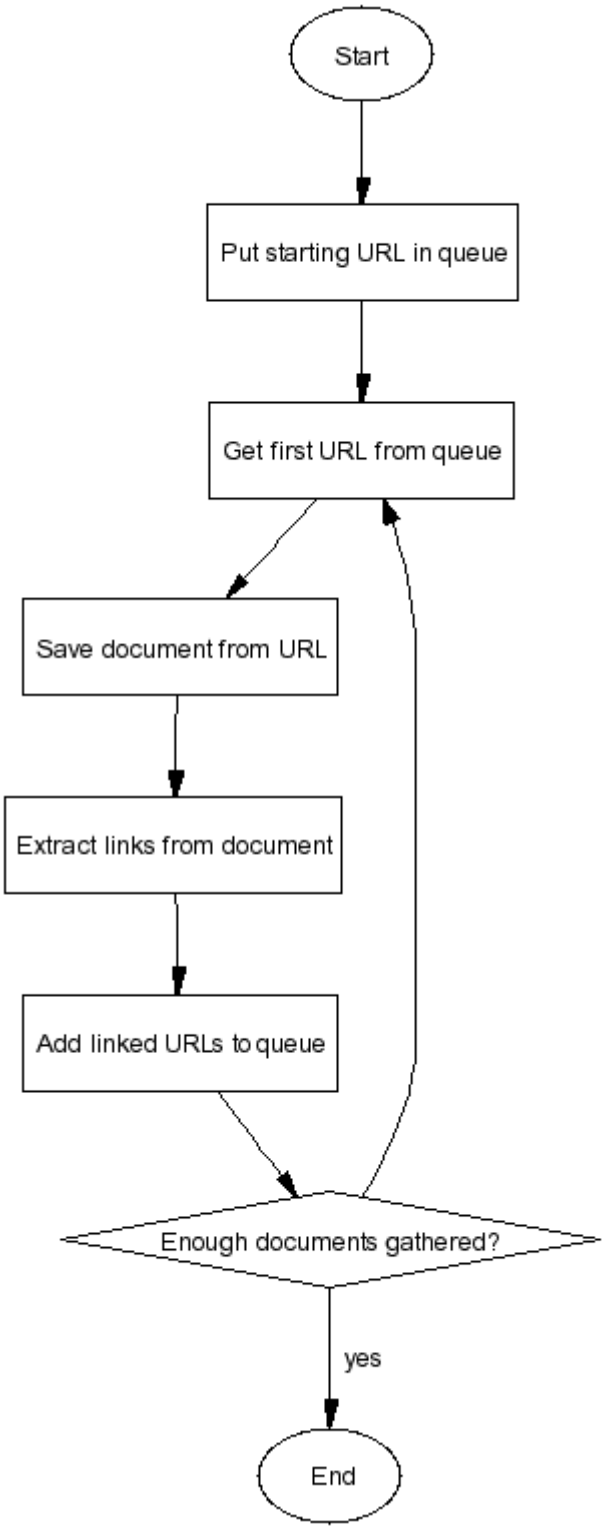


FIGURE 11 : DATA FLOW DIAGRAM 3

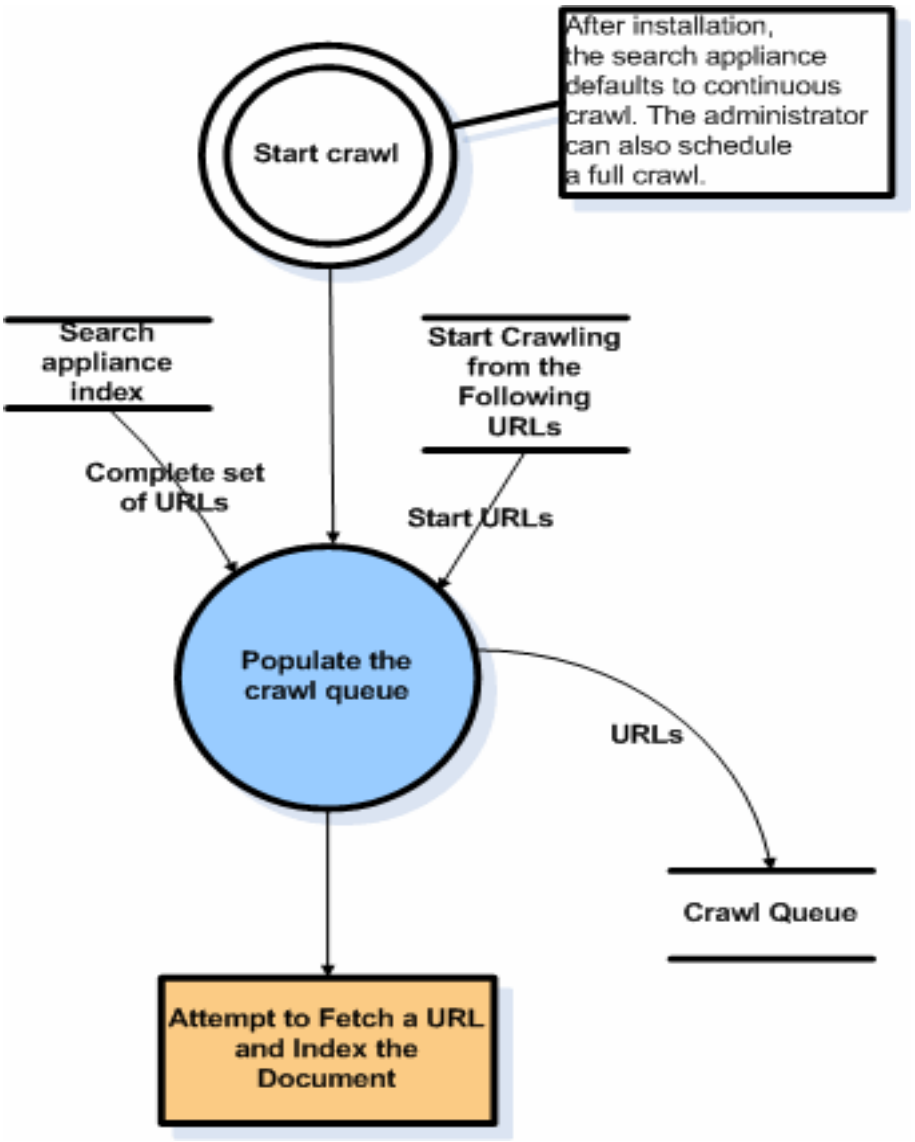
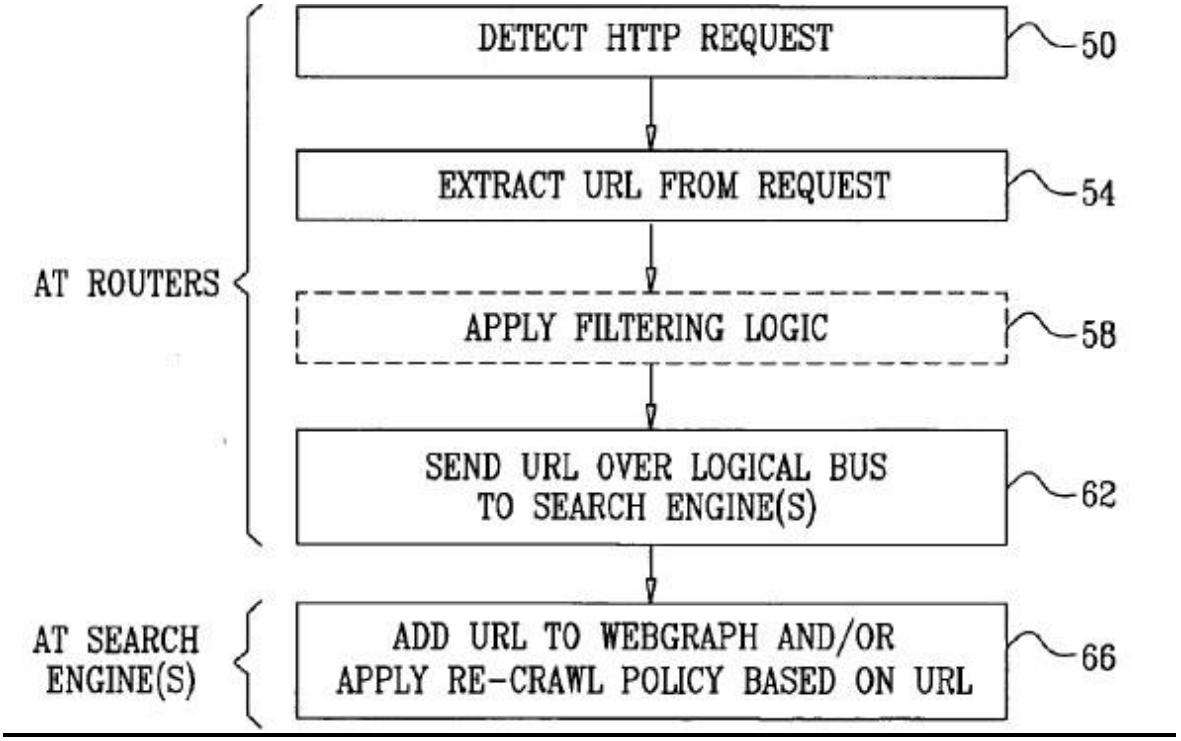


FIGURE 12 : WORKING PROCESS OF WEB CRAWLERS AT SEARCH ENGINES



FIGUIRE 13: DESIGNING PROCESS

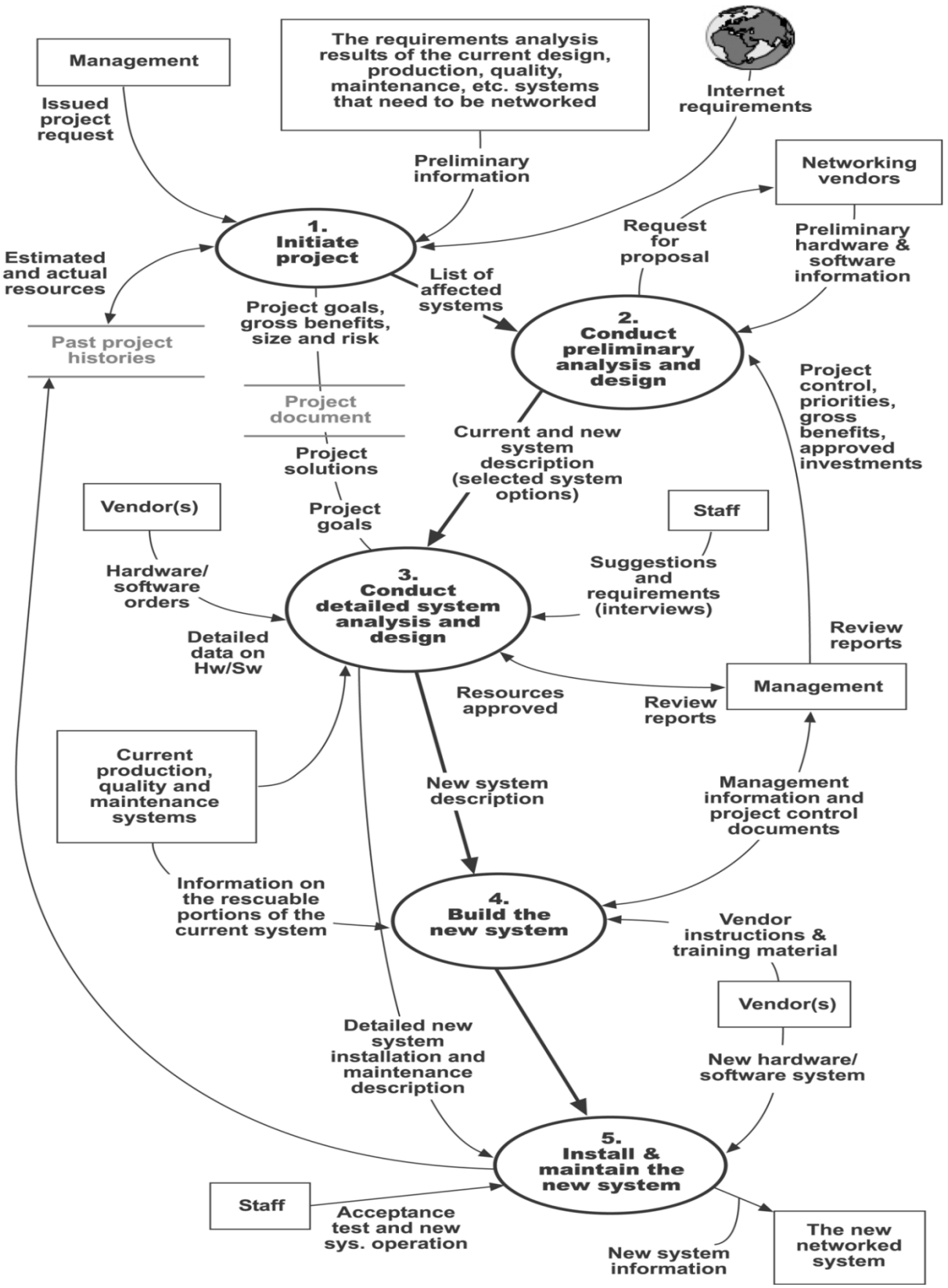
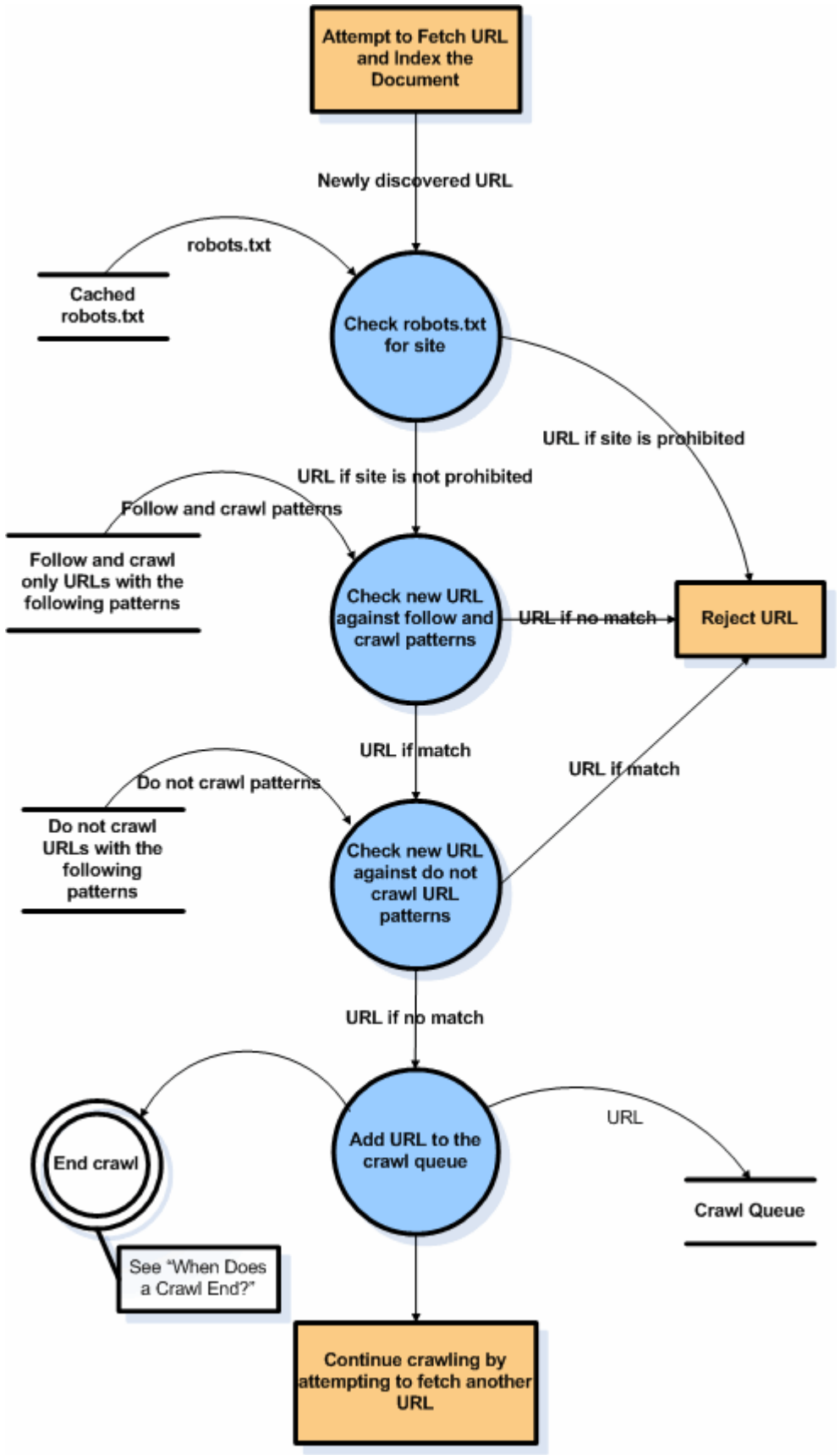


FIGURE 14: FETCHING THE URLs



PHASE 4 : IMPLEMENTATION (CODING)

THE FUNCTIONS USED IN THE PROGRAMs

1) gui()
2) void actionExit(_
3) void actionPerformed(ActionEvent e)
4) crawler(String url,int limit) // constructer
5) void downloader(String url)
6) void fileDownload(String fAddress,String destination)
7) void urlDownload()
8) void close()

4.1 MODULE 1: (GRAPHICAL USER INTERFACE)

```
package pacrawl;


 import java.awt.event.ActionEvent;

 import java.awt.event.ActionListener;

import java.awt.event.WindowAdapter;

import java.awt.event.WindowEvent;

import java.sql.*;

import javax.swing.*;


public class Gui extends JFrame implements ActionListener {

JButtonjb = new JButton("Start Crawl");
JButton jb2 = new JButton("Show Result");
JButton jb3 = new JButton("Stop");
JLabel jl1 = new JLabel("Starting Url");
```

```
JTextField jf1 = new JTextField(40);
JLabel jl2 = new JLabel("Limit on Url");
JTextField jf2 = new JTextField(14);
JTextArea ja = new JTextArea(10, 50);
int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
JScrollPanejsp = new JScrollPane(ja, v, h);

public Gui() {
setTitle("Parallel Web Crawler");
setSize(800, 600);

addWindowListener(new WindowAdapter() {
public void windowClosing(WindowEvent e) {
actionExit();
          }
       });
jb.addActionListener(this);
jb2.addActionListener(this);
jb3.addActionListener(this);

JPanel jp1 = new JPanel();
JPanel jp2 = new JPanel();
JPanel jp3 = new JPanel();
JPanel jp4 = new JPanel();
JPanel jp5 = new JPanel();

jp1.add(jl1);
jp1.add(jf1);
jp2.add(jl2);
jp2.add(jf2);
jp3.add(jb);
jp3.add(jb2);
jp3.add(jb3);
jp4.add(jsp);
jp5.add(jp1);
jp5.add(jp2);
jp5.add(jp3);
```

```
jp5.add(jp4);
setContentPane(jp5);
setVisible(true);
   }

private void actionExit() {
System.exit(0);
   }

public void actionPerformed(ActionEvent e) {
      Connection conn = null;
      String uri = "jdbc:mysql://localhost:3306/";
      String dbName = "pm";
      String driver = "com.mysql.jdbc.Driver";
      String userName = "root";
      String password = "root";
      String su = "";
int cl = 0;
      Object src = e.getSource();
try {
Class.forName(driver).newInstance();
conn = (Connection) DriverManager.getConnection(uri + dbName, userName,
password);
System.out.println("Connected to the database");
      } catch (Exception fe) {
fe.printStackTrace();
      }
if (src == jb) {
try {
Thread.sleep(100);
Class.forName("com.mysql.jdbc.Driver");
conn                                                                    =
DriverManager.getConnection("jdbc:mysql://localhost/pm?user=root&password=r
oot");
            Statement st=conn.createStatement();
st.executeUpdate("delete from bm");
         } catch (Exception e1) {
e1.printStackTrace();
```

```
            }
su = jf1.getText();
cl = Integer.parseInt(jf2.getText());

            Crawler t = new Crawler(su, cl);
try {
Thread.sleep(100);
System.out.println(".....Downloading thread is working..... ");
t.urlDownload();
            } catch (Exception e2) {
System.out.println(e2);
            }
        }
if (src == jb2) {

try {
ja.setText("");
                Statement st = (Statement) conn.createStatement();
                String sql = "select urls from bm";
ResultSetrs = st.executeQuery(sql);
while (rs.next()) {
ja.append(rs.getString(1) + "\n");

                }
            } catch (Exception ge) {
ge.printStackTrace();
            }
        }

if (src == jb3) {
actionExit();
        }
    }
}
```

MODULE 2:

```
packagepacrawl;

import java.io.*;
import java.net.*;
importjava.io.BufferedReader;
importjava.io.InputStreamReader;
import java.net.URL;

importjava.util.*;
importjava.util.List;
importjava.util.regex.Matcher;
importjava.util.regex.Pattern;
import java.sql.*;

class Crawler {

int seedno = 1, endurl, urlinlist = 0, count = 0, rowid = 0;
public static List<String> result;
public static List<String>tempresult = null;
public static String url = "";
int limit = 0;
public String textdata;

Crawler(String url, int limit) {
      this.url = url;
this.limit = limit;

System.out.println("Link extractor thread ");
      Thread t = new Thread() {
public void run() {
try {
```

```
downloader(Crawler.url);

            } catch (Exception e) {
System.out.println(e);
            }
          }
       };
t.start();
result = new ArrayList<String>();
   }

public void downloader(String url) {
try {

          URL my_url = new URL(url);
BufferedReaderbr = new BufferedReader(new
InputStreamReader(my_url.openStream()));
          String strTemp = "";
StringBuffer file = new StringBuffer("");
while (null != (strTemp = br.readLine())) {
file.append(strTemp);
          }
strTemp = file.toString();
result = extractUrls(strTemp);

for (inti = 0; i<endurl; i++) {
System.out.println();
System.out.println();

System.out.println(urlinlist + "....NEXT PAGE.....");

System.out.println(result.get(urlinlist));

urlinlist++;

downloader(result.get(urlinlist));

          }
```

```
        } catch (Exception ex) {
ex.printStackTrace();
        }

    }

    List<String>extractUrls(String value) {

        Connection conn = null;
        String uri = "Class.";
        String dbName = "pm";
        String driver = "com.mysql.jdbc.Driver";
        String userName = "root";
        String password = "root";

try {
Thread.sleep(100);

Class.forName("com.mysql.jdbc.Driver");
conn =
DriverManager.getConnection("jdbc:mysql://localhost/pm?user=root&password=r
oot");

        } catch (Exception e) {
e.printStackTrace();
        }

tempresult = new ArrayList<String>();

        String urlPattern =
"((https?|ftp|gopher|telnet|file):((//)|(\\\\))+[\\w\\d:#@%/;$()~_?\\+-=\\\\\\.&]*)";
        Pattern p = Pattern.compile(urlPattern, Pattern.CASE_INSENSITIVE);
        Matcher m = p.matcher(value);

while (m.find()) {
if (endurl<= limit) {
                String link = m.group(1);
```

```
tempresult.add(value.substring(m.start(0), m.end(0)));
            String g = """ + link + """;

try {
              Statement st = conn.createStatement();

              String sql = "insert into bmvalues(" + rowid++ + "," + g + ")";
textdata = g + "\n";
st.executeUpdate(sql);

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

System.out.println(endurl + "......." + value.substring(m.start(0), m.end(0)));
endurl++;

          } else {
urlDownload();
          }

       }

result.addAll(tempresult);

return result;
   }

voidfileDownload(String fAddress, String destinationDir) {

intslashIndex = fAddress.lastIndexOf('/');
intperiodIndex = fAddress.lastIndexOf('.');

       String fileName = fAddress.substring(slashIndex + 1);

if (periodIndex>= 1 &&slashIndex>= 0 &&slashIndex<fAddress.length() - 1) {
fileUrl(fAddress, fileName, destinationDir);
```

```
        } else {
System.err.println("path or file name.");
        }
    }

voidfileUrl(String fAddress, String localFileName, String destinationDir) {

int size = 1024;

OutputStreamoutStream = null;
URLConnectionuCon = null;

InputStreamik = null;
try {
            URL Url;
byte[] buf;
intByteRead, ByteWritten = 0;
Url = new URL(fAddress);
outStream = new BufferedOutputStream(new FileOutputStream(destinationDir +
"\\" + localFileName));

uCon = Url.openConnection();
ik = uCon.getInputStream();
buf = new byte[size];
while ((ByteRead = ik.read(buf)) != -1) {
outStream.write(buf, 0, ByteRead);
ByteWritten += ByteRead;
            }
System.out.println("Downloaded Successfully.");
System.out.println("File name:\"" + localFileName + "\"\nNoofbytes :" +
ByteWritten);
        } catch (Exception e) {
e.printStackTrace();
        } finally {
try {
ik.close();
outStream.close();
            } catch (Exception e) {
```

```
e.printStackTrace();
          }
       }
   }

voidurlDownload() {

      Connection conn = null;
try {
Class.forName("com.mysql.jdbc.Driver");
conn                                                                    =
DriverManager.getConnection("jdbc:mysql://localhost/pm?user=root&password=r
oot");

System.out.println("Connected to the database");

      } catch (Exception e) {
e.printStackTrace();
      }
try {
          Statement st = conn.createStatement();
          Statement st1 = conn.createStatement();
          Statement st2 = conn.createStatement();

if (count == 0) {
             String s2 = "drop table bm";
             String s1 = "create table bm (rowidint(4),urlsvarchar(400))";
int rs2 = st2.executeUpdate(s2);
int rs1 = st1.executeUpdate(s1);

count++;

          }
          String sql = "select urls from bm where rowid>=" + count + "";

ResultSetrs = st.executeQuery(sql);
while (rs.next()) {
System.out.println(rs.getString(1));
```

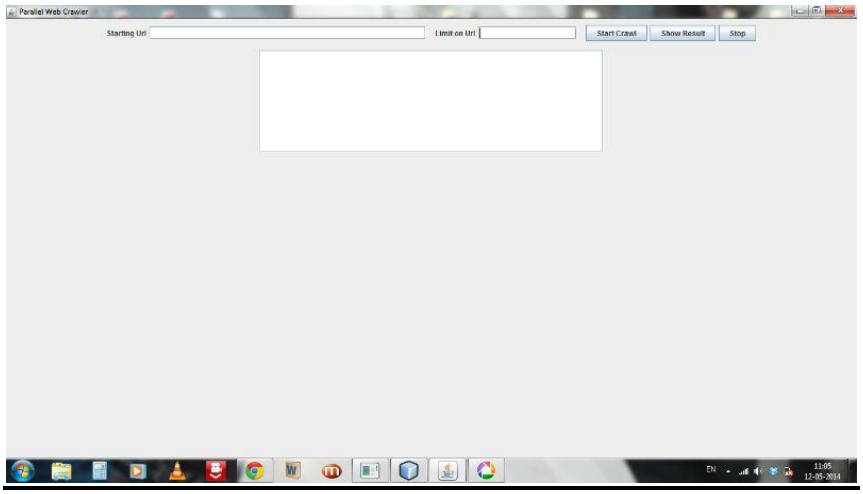
```
fileDownload(rs.getString(1), "E:/PCrawl");
count++;
          }

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

   }

void close() {

close();
   }
}

public class UrlFinder {

public static void main(String[] args) {
Gui bh1 = new Gui();

   }
```

MODULE 3: (MAIN)

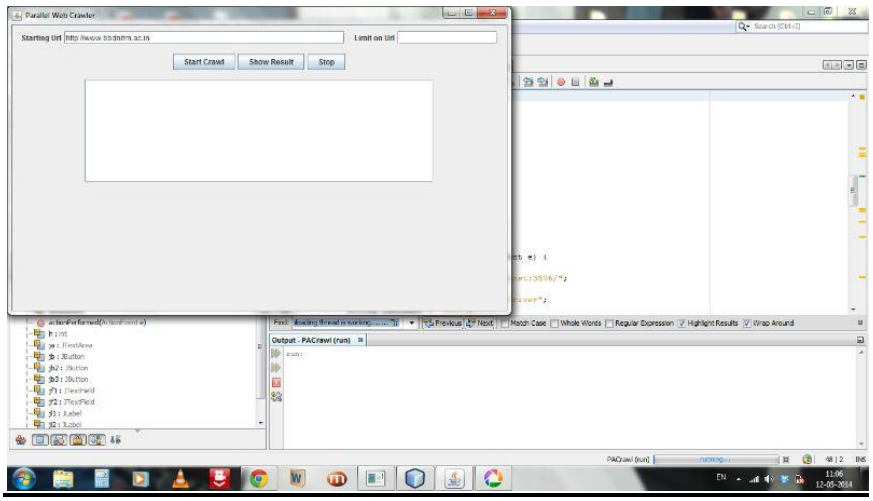
```
packagepacrawl;
public class Main {

public static void main(String[] args) {
      // TODO code application logic here
Gui bh1 = new Gui();
   }

}
```

SCREENSHOT OF THE SOFTWARE

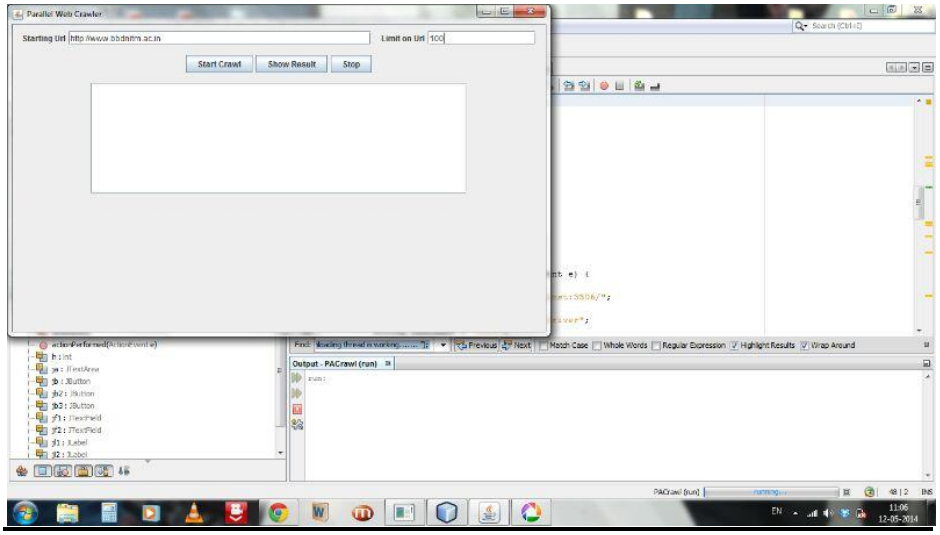
SCREENSHOT 1: GIU OF PARALLEL WEB CRAWLER



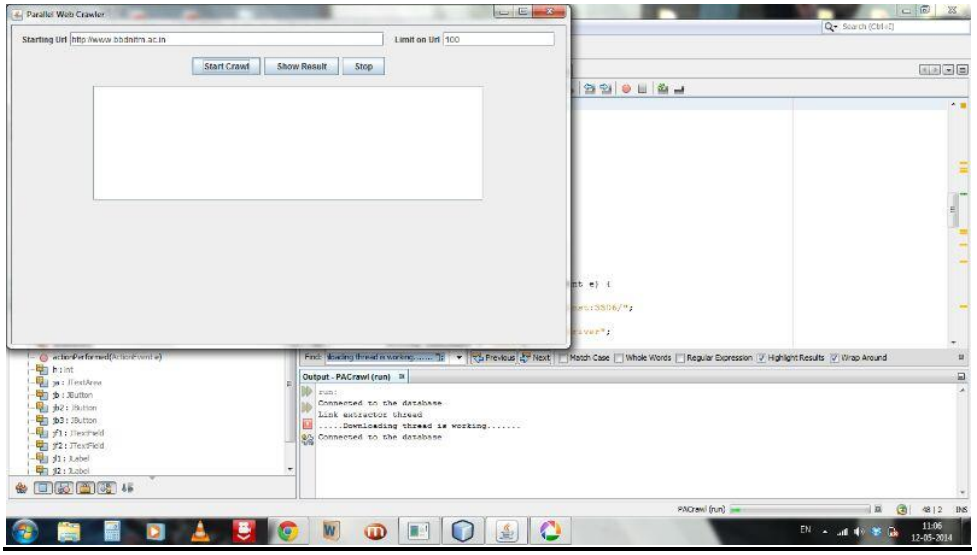
SCREENSHOT 2: STARTING URL



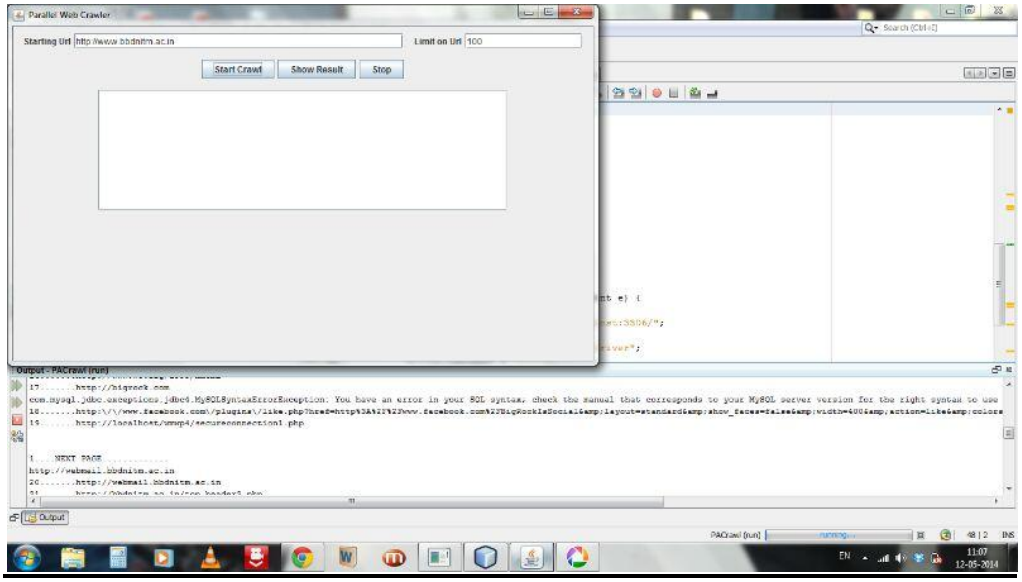
SCREENSHOT 3: PUTTING THE LIMITS ON URLs



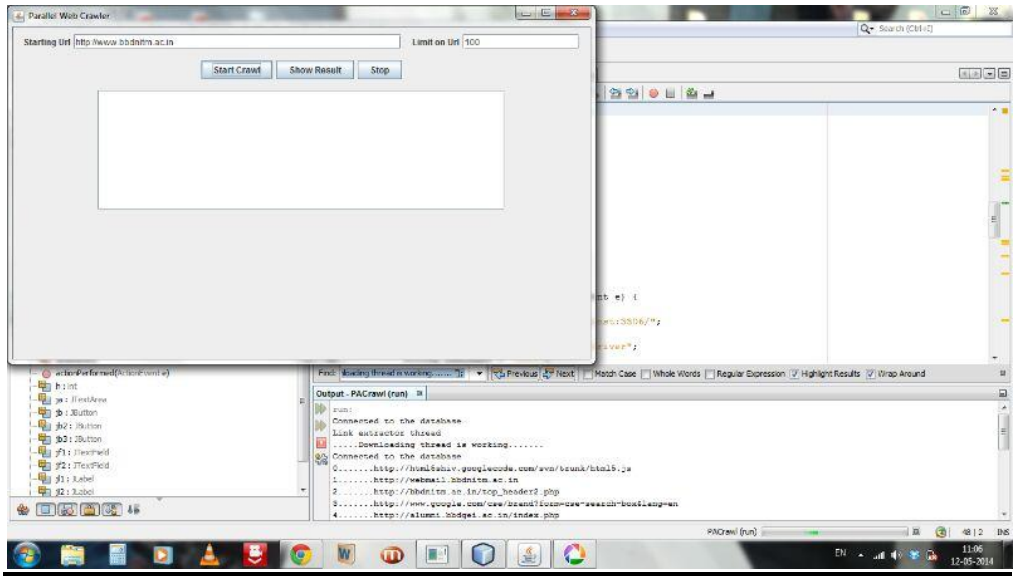
SCREENSHOT 4: INITIATED WITH THE PROCESS OF CRAWLING



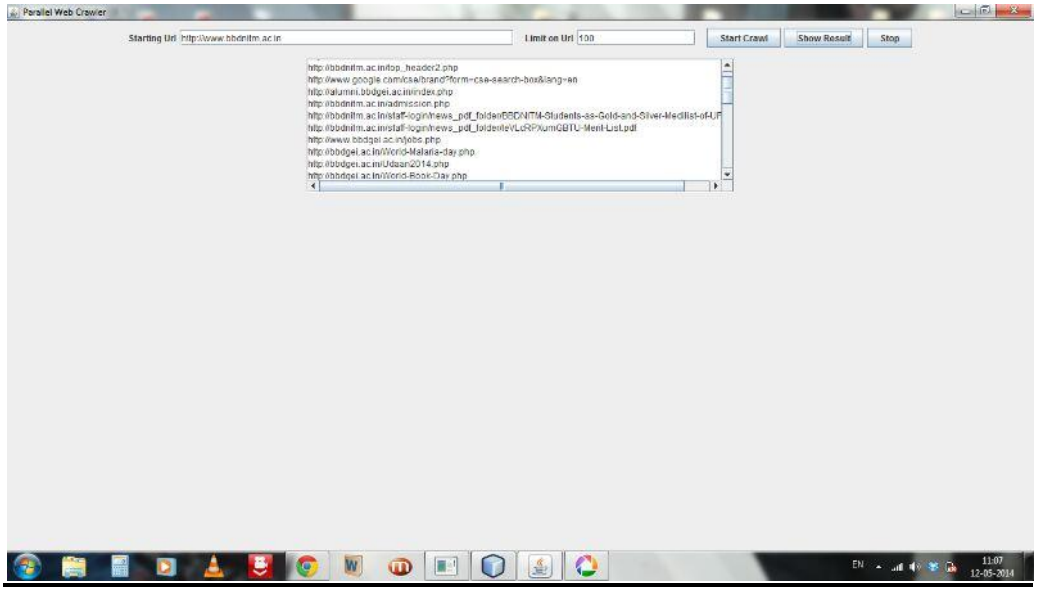
SCREENSHOT 5: CRAWLING PROCESS STATRTED [LINKS ARE BEING EXTRACTED]



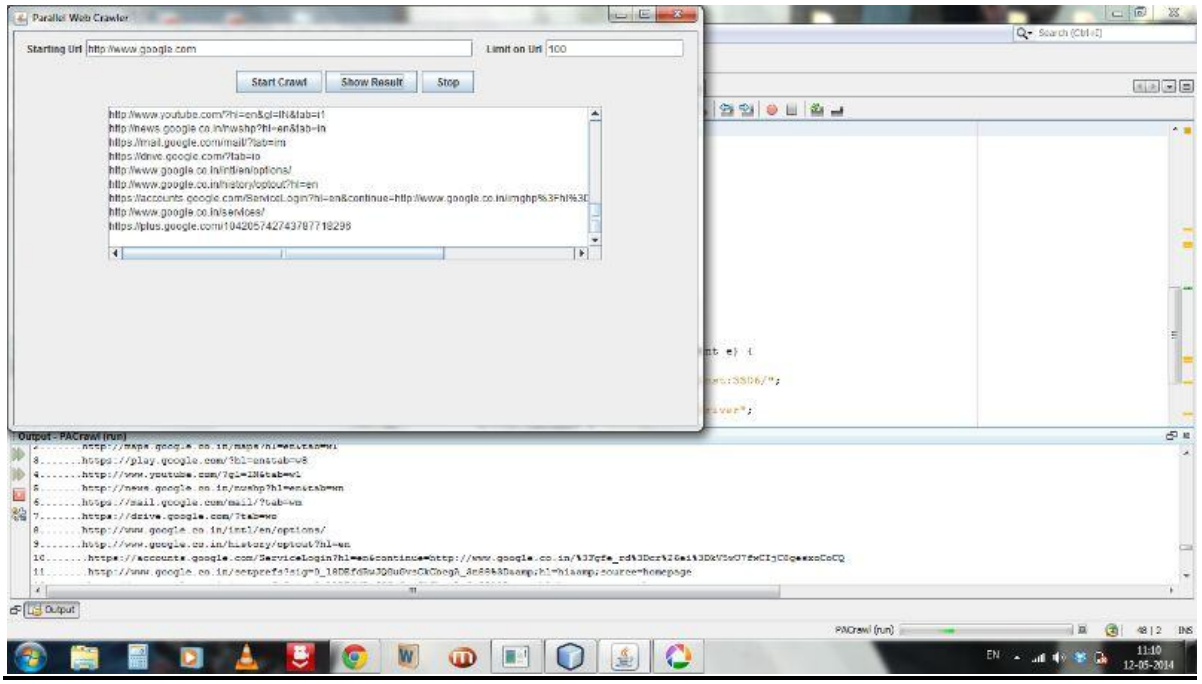
SCREENSHOT 6: LINKS ARE EXTRACTED ON NEXT PAGE



SCREENSHOT 7: OUTPUT [RESULT OF CRAWLING]



SCREENSHOT 8: WEB CRAWLING RESULT FOR OTHER ULRs



4.5 PHASE 5 : TESTING

- This phase aims at bringing all the pieces together into a special testing environment , then checks for errors , bugs and interoperability.
- For testing purpose we run the crawler again and again, fetch the URLs and download the related URLs and pages.
- We check whether the fetched URLs and pages are getting stored in the database or not.
- We successfully test our software and hence ensure the proper functioning of our window application.

4.5.1 Current Problem

Since size and change rate of the Web is very high, hence, the crawler needs to address many important challenges, including the following:

- **Overlap:** When multiple processes running parallel to download pages, it is possible that different processes download the same page multiple times. Such multiple downloads should be minimized to save network bandwidth and increase the crawler's effectiveness[1].
- **Quality:** A crawler wants to download "important"pages first. However, in a parallel crawler, each process may not be aware of the whole image of the Web that theyhave collectively downloadedso far[5].
- **Page Freshness**: Once the crawler has downloaded a significant number of pages, it has to start revisiting the downloaded pages in order todetect changes and refresh the downloaded collection. The crawler nneeds to carefully decide which pages to revisit and which pages to skip in order to achievehigh "freshness"of pages.
- **Network Load Reduction:** When the crawler collects pages from the Web, it consumes resources belonging to other organizations. Therefore,the crawler should minimize its impact on these resources[1].
- **Communication Bandwidth:** In order to prevent overlap, or to improve the quality of the downloaded pages,crawling processes need to periodically communicate and coordinate with each other. However, this communication may grow significantly as the number of crawling processes increases[5].

- **Scalability**: Due to enormous size of the Web, it is often imperative to run a parallel crawler. A single-process crawler simply cannot achieve the required download rate in certain cases[5].
- **Network-Load Dispersion**: Multiple crawling processes of a parallel crawler may run at geographically distant locations, each downloading "geographically-adjacent" pages .According to this dispersion might be necessary when asingle network cannot handle the heavy load from a large scale crawl[5]

4.5.2 Area of Improvement

In this section we give a brief solutions to the most important challenges and issues which a Parallel Crawler can have.

- **Scalability:** Due to the fullydistributed architectureof Crawler, its performance can be scaled by adding extra machine. Inaddition, data structures of Crawler use a limited amount of main emory,regardless of the size of the Web. Therefore, Crawler can manage to handle the rapidly growing web.
- **Network Load Distribution and Dispersion:** Multiple crawling processes of parallel crawler mayrunat geographically distant locations, each downloading "geographically-adjacent"pages.For example, a process in Germany maydownload all European pages, while another one in Japan crawls all Asian pages. In this way, we can *disperse* the network load to multiple regions. In particular, this dispersion

  might be necessary when a single network cannot handle the heavyload from a large- scale crawl.

- **We identify the three following goals:**

- Any time, each URL should be assigned to a specific agent, which is solely responsible for it.
- For any given URL,the knowledge of its responsible agent should be locally available. In other words, every agent should have the capability to compute the identifier of the a gentresponsible for aURL, without communicating.
- The distribution of URLs should be balanced, that is, each agent should be responsible for approximately the same number of URLs.

 Extensibility: Due to the fully distributed architecture of Crawler, its performance can be scaled by adding extra machine. In addition, data structures of SCrawler use a limited amount of main memory,regardless of the size of the Web. Therefore, SCrawler can manage to handle the rapidly growing web.

 Portability: SCrawler has been designed in a modular way. For a particular crawling environment, SCrawler can be reconfigured by plugging in appropriate modules

 Politeness: Crawler is written entire lyin Java to achieve platform independence, thus runs on any platforms for which there exists a Javavirtual machine.

 High-Throughput Crawling: By harnessingthepower ofalargenumberof nodes, the crawlingserviceis morescalable than centralized systems. without modifying its core components.

PHASE 6 : MAINTAINENCE

MAINTENANCE:
Various maintenance activities are carried out after the software release, to ensure that it runs perfectly.

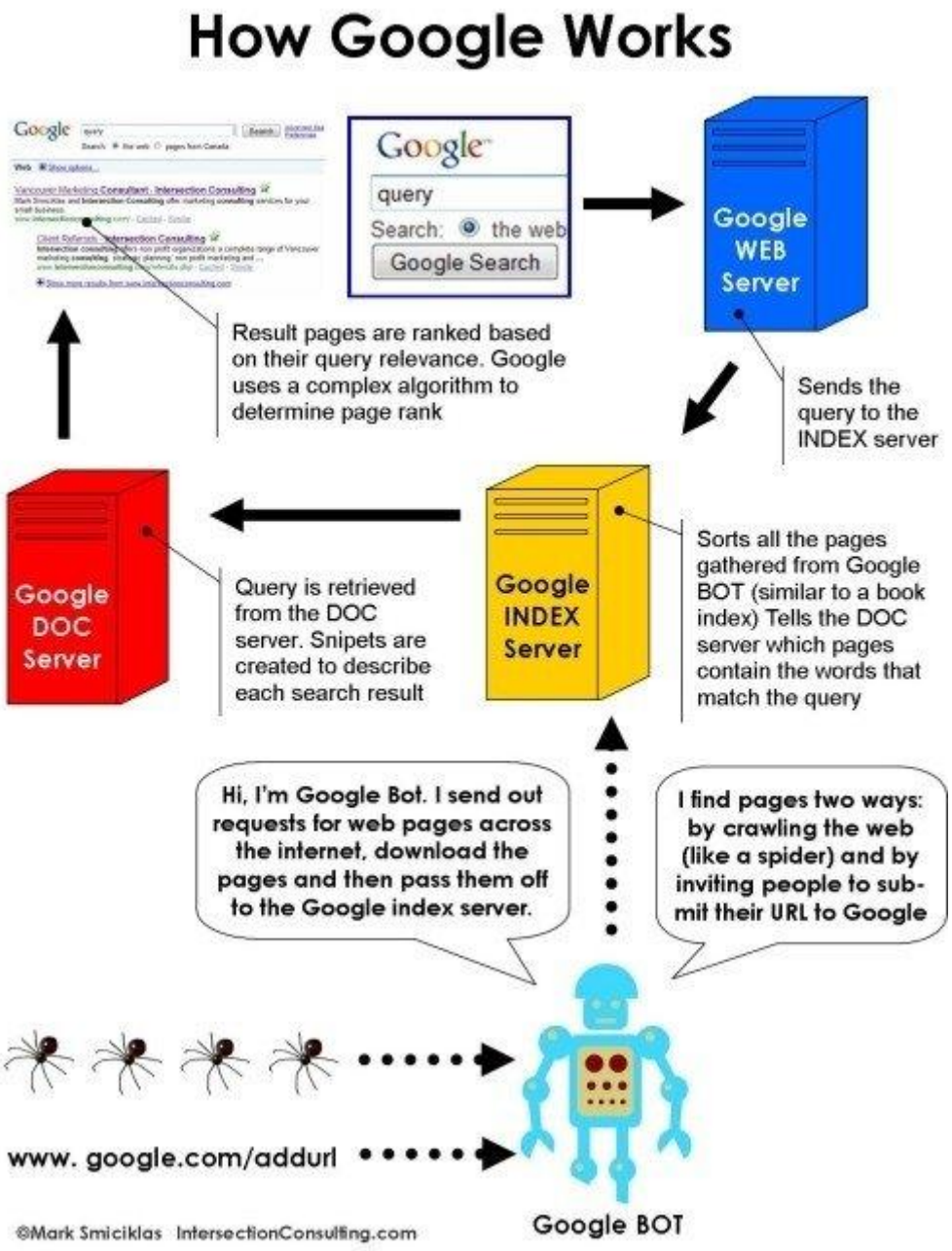
- During the maintenance stage of SDLC, the system is assessed to ensure it doesn't become obsolete.
- It involves continuous evaluation of the system in terms of its performance.
- This phase is carried out when the software has been developed completely.
- Our main aim is to improve the quality of the crawler by-

1. reducing redundancy
2. increasing the speed and the crawling capacity of the crawler

The various sets of maintenance processes can be categorized as:

- Corrective maintenance - Reactive modification of the software product performed after delivery to correct discovered problems.
- Adaptive Maintenance - Modification of a software product performed after delivery to keep the software product performed after delivery to keep the software product usable with the change in the changing environment.
- Perfective maintenance – modification of the software product after delivery to improve the performance or maintainability.
- Preventive maintenance – Modification of the software product after delivery to detect and correct the latent faults in the software product before they become effective faults.

Example : Google Crawler



Examples of Web crawlers :

1)World Wide Web Worm
2)Yahoo!slurp-Yahoo search crawler
3)Msnbot-Microsoftbing web crawler
4)FAST Crawler
5)Googlebot
6)PolyBot

Future Scope of Project

Benchmarking Measuring the efficiency of a Web crawler –considering only short-term scheduling –is not an easy task. We need a framework for comparing crawlers that accounts for the network usage, the processing power required, and the memory usage. It would be good to have a measure that allows us to compare, e.g., 30 pages per second in a 1Ghz processor with 1Gb RAM with 20 pages per second in a 800MHz processor with 640Mb RAM. A more important problem is that network conditions vary so for testing different Web crawlers we must consider the time of the day, network capacity, etc.

Finding combined strategies Specifically the best parameters for the manager program, in terms of the importance that should be given to intrinsic quality, representational quality and freshness. We also consider that the scheduling policies should adapt to different Web sites, and our model provides a framework for that kind of adaptability.

Exploiting query logs for Web crawling The usage of user query logs in a search engine for guiding a Web crawling is an important step in integrating the collection and search processes. The query logs can be used to refresh frequently returned pages faster, so the crawler refreshes the active set of the search engine more often. Moreover, the query terms used in Web search can be given priority, so the Web crawler scheduling could be biased toward pages that contains the query terms that are being queried more frequently by the search engine's users.

Parsing of non-HTML files:
During the crawl, the crawler looks for content-type of the page to be crawled. It crawls pages with a content-type of text/html and ignores PDFs, Javascript, CSS and other text or application files. If one can come up with a parser for such documents, it can be plugged into the code and various documents can be crawled and information extracted from various documents. Right now, extract_links is the function to parse HTML documents. One can easily add functions to crawl and parse different types of pages .

Processing of a fetched document:

The crawler chooses a server to fetch next URL to crawl. This depends on the configuration switch of the courtesy pause to be observed between two fetches from the same server. This is done to avoid overloading one particular server with multiple requests in a short time.

The current method selects a server and crawls a URL hosted on that server. But the URL can be unreachable or might have errors. This leads to a delay of period of time-out set for the User Agent object. Currently, the time-out is set to 30 seconds and the courtesy pause is set to 10 seconds. Thus, the crawler can try to fetch just one where it could fetch three pages. Moreover, due to large amount of links on a page and checking each link with the database to see if it is seen before or not; some pages require more than a minute to be processed .

Context based sorting of URLs in the frontier:

The sorting of URLs is a dynamic function. The URLs in the frontier are sorted in the order of their frequency of discovery in already fetched documents. A rank is already provided to each URL and a URL with the highest rank is crawled first. A function which calculates rank according to the context of the link or its position from the seed page or some parameter based on link structure of the URLs and add it the current rank can classify more accurately how important a URL is and what its position is in the frontier queue .

CONCLUSION

A parallel crawler is implemented and many design issues related to a distributed environment are learnt. They are an important aspect of search engines. Data manipulation by these crawlers cover a wide area.It is crucial to preserve a good balance between random access and disk access.

Almost all of the issues are resolved but there are some of them where there exists an opportunity for improvement or enhancement. Key problems faced are mentioned in challenges and issues. These are also the issues when resolved; can enhance the performance and accuracy of the crawling task.

- We have successfully built Page Downloader & Parser and tested it successfully.
- We have successfully built Analyzer, link extract or and the database.
- We have studied various research papers related to the project.
- We are studying page ranking techniques and performance evaluation techniques.
- We have successfully developed the crawling mechanism by using parallelism which extracts the information from the web.
- We have developed an attracting Graphical user interface for the crawler.
- Our Comparison with other web crawlers how that significant improvement can be made if the technique applied on alarge scale

REFERENCES

1. **The Deep Web: Surfacing Hidden Value.** ***http://www.completeplanet.com/Tutorials/DeepWeb/***
2. **S. Lawrence and C. L. Giles. Searching the World Wide Web. *Science*, 280(5360):98, 1998.**
3. **S. Lawrence and C. L. Giles. Accessibility of information on the Web. *Nature*, 400:107-109, 1999.**
4. **S. Raghavan and H. Garcia-Molina. Crawling the Hidden Web. Technical Report 2000-36, Computer Science Department, Stanford University, December 2000. Available at *http://dbpubs.stanford.edu/pubs/2000-36***