COM3010 - RDFa Web Spider

Title:	RDFa Web Spider
Name:	Paul Ridgway
Supervisor:	Fabio Ciravegna
Module Code:	COM3010
Date:	4th December 2009

This report is submitted in partial fulfillment of the requirement for the degree of Master of Software Engineering in Computer Science by Paul Ridgway.

SIGNED DECLARATION

All sentences or passages quoted in this report from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations which are not the work of the author of this report have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this project and the degree examination as a whole.

Name:	Paul Ridgway
Signature:	
Date:	4th December 2009

Abstract

Current web 'standards' formalize formatting and provision of information on the Web, but little of this information can be put into context by a machine without heavy analysis. A proposed html extension called RDFa allows the content creator to specify the type of data on a web page which implies or specifies the context and relationship of this data. This allows automated processes to potentially discern the meaning of the information. There are many search engines for several different types of media, but most commonly they allow the user to search content on the Web, return results based on a relevance match which is often done by the frequency in which the search term appears in the document. The aim of this project is to index pages which contain RDFa data for searching, tackling issues involved with crawling and indexing large numbers of pages and enormous amounts of data.

CONTENTS

FIGURES

Glossary

Crawling
Crawling the Web is the process of automatically and methodically browsing the web.

(Forward) Index
An index is an ordered list mapping an identifier to some data.

Indexing
Indexing is the process of parsing data and creating an index from it.

The Internet
The Internet is a global system of interconnected computers and networks.

IP (Internet Protocol)
The Internet Protocol (IP) is a protocol used for communicating data across a packet-switched internetwork using the Internet Protocol Suite, also referred to as TCP/IP.

RDF
The Resource Description Framework is used to model information used in web resources.

RDFa
The W3C Resource Description Framework - in – attributes recommendation adds RDF attribute extensions to web pages.

Repository
In the context of this paper a repository is a data store for web pages.

Reverse Index
A reverse index is a list mapping data tokens to the original set of data.

TCP
The Transmission Control Protocol (TCP) is one of the core protocols of the Internet Protocol Suite.

The Web
The Web is a system of interconnected hypertext documents on the Internet.

CHAPTER 1: INTRODUCTION

The Web has grown exponentially since its conception and it is now extremely large. Finding information on The Web without any assistance is near impossible unless you have prior knowledge of its location, and this has ensured that search engines will always be well used and make an enormous contribution to the usefulness of The Web. A search engine must first (and repeatedly) collect data for its index, as the index is searched when a search engine is asked to find information.

This collection process has two parts, first The Web is crawled, and then the data retrieved by the Crawl is Indexed. Crawling is a process where a crawler repeatedly downloads a page, identifies all links on it, downloads those pages, identifies all their links, and repeatedly harvests pages until it has acquired every linked page it can find. These pages are all stored so that they can be Indexed.

The pages downloaded are then Indexed, each page is parsed and the visible content is located. All individual words in the page are identified and counted so that a forward index can be created where each page has a list of words and counts. There is also a reverse index for each word which is a list of all the pages in which that word appears. Different search engines will vary their Crawling and Indexing procedures so that the resultant data is tailored to match the features of the search service provided.

The majority of search engines create their index from visible page content and little more, however there are new standards emerging, called RDF that allow the context of information on the Web to be specified. The aim of this project is to index pages that are annotated with RDF data to potentially allow for a detailed search index and interface.

Crawling for RDF data will require crawling every page, but only storing some of them, namely the ones containing RDF mark-up. There are several key issues with crawling and indexing amount of data, in this case the Web. This project will attempt to tackle these issues with practical scalable solutions.

CHAPTER 2: LITERATURE SURVEY

This literature survey examines the infrastructure and topology of the World Wide Web, drawing particular reference to finding, indexing and searching for information with and without contextual enhancements.

2.1: THE STRUCTURE OF THE WEB

The World Wide Web (the Web) is an enormous collection of interlinked documents which reside on servers connected to the Internet. There are several different services that allow the Web to exist on The Internet.

2.1.1: THE BASICS

In the crudest sense, The Internet is a very big network of computers. In reality it is lots of networks linked together to make The Internet. The words "web" and "internet" are often mistakenly used in everyday language to refer to the "Web" but they are not the same thing. The Internet is a global network, whereas the Web is the collection of web pages that are accessible over the Internet for a Web Server.

The Internet uses a numerical addressing system which allows computers to connect directly to each other. The system currently in use (called IPv4) which is of the format $\mathrm{a.b.c.d}$ where a, b c or d are integers between 0 and 255 (with some restrictions), it is basically a 32 bit address. This means the max number of available addresses is $256^4 = 4{,}294{,}967{,}296$ without restrictions, in practice there are fewer.

The Internet has now become so big that about 10 years ago a specification for a new IP Protocol (IPv6) was proposed (Network Working Group, 1998). An IPv6 address is of the format $aabb{:}ccdd{:}eeff{:}gghh{:}iijj{:}kkll{:}mmnn{:}oopp$ where each pair is a hexadecimal representation of an 8 bit number making the IPv6 a 128 bit address, giving an address space of $256^{16} = 3.4 \times 10^{38}$ – which should last much longer than IPv4.

The IP protocol is merely one of the technologies of the giant infrastructure that is The Internet and the Web. The Web (viewing content, at least) relies on two main types of server, HTTP and DNS. DNS stands for Domain Name System and it is a mechanism for resolving domain names (used for memorability, structure and order) to IP addresses. For example, when a user tries to browse the page at www.google.com the web browser asks the ISP's name servers to resolve www.google.com and it will look it up and return an IP so the computer can make a direct connection. DNS is a hierarchical system and one Name Server often requests information from another to resolve a query, but the detail on how this is carried out is beyond the scope of this paper.

Revisions and updates in the network technology behind the Internet and the Web could provide problems for web spiders and indexers if they are not able to keep up with these changes, and gracefully operate during transition periods.

2.1.1: HOW BIG CAN IT BE?

There are currently well *over* 1.25 trillion (1,250,000,000,000) unique linked URLs on the Internet (Google, 2008) (Majestic-12, 2009). There are bound to be more pages on the

Internet as some will not be linked to others, making them hard to find, and others will be behind password protected areas, or prohibited by spider politeness rules such as robots files (which are discussed later).

The web contains a truly vast amount of data, for example (Building Nutch: Open Source Search, 2004) assumes that a single web page is on average 10 KB in size, based on this assumption 1.25 trillion pages would take up 11.3 PB if they were stored in an uncompressed format. That paper is 5 years old, with advances in network technology, the average internet connection is now much faster and it is clear that web pages are more content rich now with images and other included files such as JavaScript and CSS includes, making the overall average size of web pages larger.

The Google homepage, which is renowned and now patented (Lardinois, 2009) as a very simple user interface requires 50 KB of bandwidth, a Google search result is 75 KB and many other reasonably simple pages require well over 100 KB. This could put the disk cost of storing the pages at 113 PB, but this is still an estimate as it excludes the space needed for all the streaming video, image hosting, content distribution and all the other rich content providing sites.

In 1998 Google had an index of 24 million pages (Brin, et al., 1998) and in the space of 10 years it has risen to well over 1 trillion. Google store both copies of the pages, and reverse indexes (discussed later) for all words in the page. Their 24 million page index of 1988, with a compressed repository of all pages downloaded was 108.7 GB, assuming the data is still stored in a similar format with the same level of compression, their new index of today's web would be around 41,000 times larger, which would be 4.25 PB. This estimation does not compensate for the increase in internet connection speed and page size. Their paper acknowledges that as computing performance increases they could easily use heavier and more intensive compression to reduce their index without having to worry about the performance overhead.

These statistics are based on research and are not definitively accurate but they are educated estimates. They are also used later and their use implies this warning.

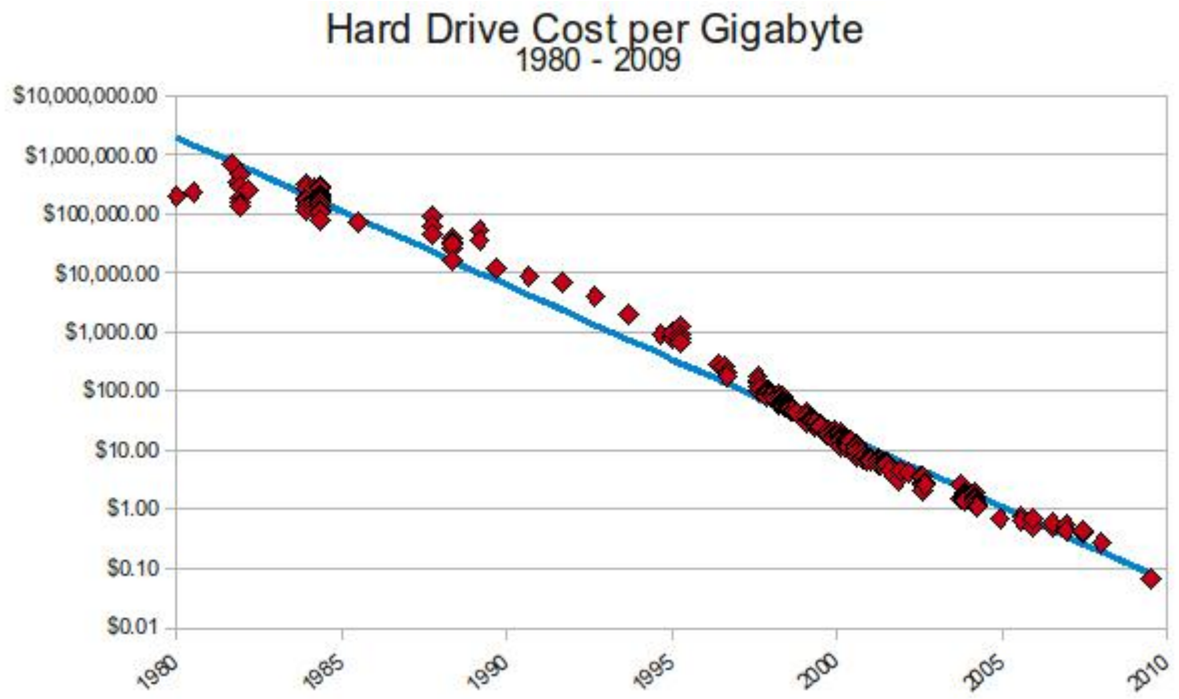


Figure 2.1: A graph of the effective cost per GB of hard drive storage against time (Matthew Komorowski, 2009). Permission acquired.

There are two main factors that have contributed to the exponential growth of the Web. Firstly, the number of internet users has grown by almost 5 times than it was in 2000, and many of these users will be contributing to the Internet in one way or another, for example by making websites, setting up businesses and participating in forums. The other factor is the cost of disk space. When Google released their first big index in 1998, disk space cost about $50 per GB, whereas today the rate is about $0.07 per GB (Figure 2.1) (Matthew Komorowski, 2009). With disk space getting exponentially cheaper there is less pressure on service providers to clean up old content to save space; instead many sites now have archives of older versions of pages or documents. A prime example of archiving is archive.org (also known as The Wayback Machine) who have been archiving copies of public web pages since 1996, and they claim to have over 3 PB of storage for this task (Internet Archive, 2009).

The size of the Web as a whole is a factor that must be taken into consideration when attempting to crawl all or part of it. If a front-end search style interface is being provided then storage of the data for processing and referencing would require serious consideration, but even if this is not the plan, downloading and parsing the data will still require lots of bandwidth, time, processing and power. All of these issues will be addressed later on.

2.1.2: Browsing it all

The estimated number of internet users is just under 2 billion (Internet World Stats, 2009) and the majority of those users will be trying to find information in one way or another. If each user was assigned an equal portion of individual URLs they would have over 625 each, and manually searching those pages would still take about a day if the user spent a couple of minutes on each. Furthermore, if their search was completed, that small one two billionth of the Web may not have contained the information they were looking for, rendering their effort useless.

2.1.3: Search Engines

It is a fact that the Web is huge, and that no one user could easily find information on it unless they had prior knowledge as to its location. That is why search engines are essential to the everyday use of the Web and why almost half of the top 20 websites ranked by popularity are search engines (Alexa, 2009). Search engines provide a crucial gateway to the Internet, allowing users to enter a short query and frequently find the information for which they were looking.

Search engines can only truly be considered useful if the user is able to find the information or site that they are looking for fairly quickly. It is a common belief that most users of a search engine will only look at the first page or two of results, which is usually the first 10 or 20 results. This means that the algorithms used to sort the results must be very adept at ranking the entries in their indexes in terms of relevance to the user's query. This has always been a problem for the operators of search engines as for as long as search engines have been around, there have been people trying to mislead them and distort the results by using various tactics to promote their sites for specific queries that they may not actually be related to, to increase traffic, sales or to capitalize from advertising.

2.1.4: Getting a page from the web

Retrieving a web page from the Internet is a process that requires multiple steps, only application layer protocols and interactions will be considered:

1. Parse the URL
2. Resolve and IP Address for the domain
3. Connect to the web server
4. Send the Page Request
5. Wait for/accept the Response

Parsing the URL

Web Page URLs are of the format

$$http://server[:port]/folder/page$$

The *http* prefix indicates that the resource is to be retrieved from a web server (other examples are *ftp*). The *server* element can be an IP address or a resolvable domain name. The *:port* section is optional, a *http* prefix implies a default port of 80 but it can be specified that the web server is running on a different port.

A URL can omit the */folder/page* section, if this is the case a trailing forward slash will be added as / is the location of the default page at the root of the site. The */folder/page* section is the path to the page.

Resolve the IP Address for the Domain

If the *server* value is an IP address then this step can be skipped, otherwise the computer performing the request must contact the local name server (which is usually specified by the IP configuration of that machine) and ask for the domain to be translated to an IP address.

Connect to the Web Server
The computer now needs to connect to the server using the IP address and port specified (or 80 if there is no port specified). The connection is done using the TCP protocol. Upon successful connection there is no 'welcome message' as with some protocols, the client is free to send the request.

Send the Page Request
The client now needs to format and send the page request. This tells the server the domain name requested, and the page. Optionally, other information can be sent like form variables, cookie settings or restrictions on content type or language.

A basic request is formatted as follows (Network Working Group, 1999):

```
GET /folder/page HTTP/1.1
Host: www.domain.com
```

This simple request merely asks for a page, specifying no restrictions and without any cookie data. Without cookie data advanced features such as sessions cannot be used. Cookies and other restrictions are conveyed and specified in the similar means to the 'Host' attribute, in the format:

```
Property: value
```

The first line of the query first states the method, in this (and many cases) 'GET' followed by the path to the document, relative to the server root, and finally the HTTP version expected of the format of the exchange. The server can reject unsupported protocol versions. The request is finished with a blank line, technically a 'character return, line feed'. Some methods allow or require data after the blank line, as a spider will upload no data this does not need to be considered.

After a request the connection can remain open if requested and supported by the server. A property called 'Connection' with a value set to 'close' will cause the server to close the connection after a response. This is often used in simpler request mechanisms, especially if pages are not requested sequentially and repeatedly from the same site.

Wait for/accept the Response
The client must now wait for the response from the server, if the connection was successful then it is more likely a response will be received, however if the server is inundated with requests there could be a long, or indefinite delay if the request goes astray. This may cause a timeout to occur and pass an error message or page to the user. The following is a very basic request and response after a request for http://www.google.com/robots.txt (the response has been truncated from a longer list of robot control statements).

Request:

```
GET /robots.txt HTTP/1.1
Host: www.google.com
```

Response:

```
HTTP/1.1 200 OK
Content-Type: text/plain
Last-Modified: Wed, 18 Nov 2009 01:25:08 GMT
Set-Cookie:
PREF=ID=7b3a862b4b1b0006:TM=1258926213:LM=1258926213:S=RrIw8wOm
t0iMrPbT; expires=Tue, 22-Nov-2011 21:43:33 GMT; path=/;
domain=.google.com
Date: Sun, 22 Nov 2009 21:43:33 GMT
Server: gws
Cache-Control: private, x-gzip-ok=""
X-XSS-Protection: 0
Expires: Sun, 22 Nov 2009 21:43:33 GMT
Transfer-Encoding: chunked

User-agent: *
Disallow: /search
```

The content of the response starts with the line ‘User-agent: *’, and it always starts after the blank line which follows the header – regardless of the type of data requested.

2.2: The web, in context

There is a lot of ambiguity in language, for example the word "close" can refer to both proximity (those cars are close to each other) or state (close and open, in reference to electronics for example gates and switches, or something as common as a door) but it is the context in which the word is used that often determines its meaning. As the Web is and must be indexed automatically due to its size, computers are left to analyze the content and though there is much research into the analysis and processing of text it is still far from perfect and can be a very resource intensive process. An extension to the XHTML markup language called RDFa has been developed to allow machines to easily 'read' web pages, giving them the ability to look at the data between HTML tags and determine the meaning of the data. For example, whether it refers to a person, or a place, or any number of other things (W3C, 2008). RDFa allows the representation of RDF data as XHTML attributes.

2.2.1: RDF and RDFa

RDF (the Resource Description Framework) is a language for providing information about resources on the Web (W3C, 2004). The RDF specification is built on the XML syntax. The intention for RDF is that it can be used in situations where the information is to be processed by computers and not individuals, for example data mining, or comparisons. RDF is based on the idea that web resources are identified using URIs (uniform resource identifiers) and these URI’s can be described with properties and values.

RDFa allows RDF data to be embedded into an XHTML page as tag attributes:

```
<tag attribute="value"></tag>
```

This allows a content creator to say

This page was created by Some Person.

The code behind this page that represents the link will also state that the link points to the page of a person. This page may then have elements that verify this, like the person's name, or sites they have worked on. Identifying people's details is just one of many uses of RDFa, and it is already an existing ontology but there are many including those for books, products and images.

An ontology is a description of entities and their relationships, which is written in XML and designed to be read by computers and not humans.

2.2.2: Searching content vs. context

When currently searching the Web there is little context analysis, for example some names are ambiguous in the sense that they are made up of words which have another meaning in language. With contextual information available a user could then search and specify in which context they were searching, for example they could specify that they are (or not) looking for a person.

2.2.3: Storing RDFa Data

A system has been developed for storing RDF called a triple store. It stores identities that are constructed from triplex collections of strings. The triplex collections represent a relationship between a subject, predicate and object (Jack Rusher). Storing the data is the less technically challenging part, the feature of many RDFa triple stores is that they allow for logical querying, in a prolog style syntax which allows for logical relationships to be created.

2.3: Crawling and Indexing

There are numerous projects in place to index the Web for different purposes. A very common reason is to provide data for a search service, for example Google, Yahoo and Bing (formerly Live Search). But there are other reasons, as previously mentioned Archive.org indexes data so that it can keep a historical record, and the company Majestic-12 provide linking relationship statistics to companies and individuals who carry out Search Engine Optimization services (used to improve search engine rankings).

'Indexing' the Web as a whole has two major parts, *crawling* the Web to find and download all the pages and then *indexing* those pages by parsing them and creating a searchable index structure.

2.3.1: General infrastructure

The simplest logical process for crawling the Web is as follows, it assumes that we have a list of URLs to crawl and that it keeps track of the URLs that have already been called:

1. Add a ‘seed URL’ (a URL to start with) to the crawl list
2. Download the next entry on the crawl list if has not already been downloaded before
3. Parse the HTML extracting URLs.
4. Save the HTML for indexing
5. Add those URLs to the crawl list
6. While the crawl list is not empty go to step 2.

This basic process has issues and limitations that are considered later on.

This simplest logical process for indexing the crawled data is as follows, it assumes (crudely) that we have a big folder with all pages in them and that they are deleted after being indexed and that for each word we have a list of pages in which they occur:

1. Load the next page in the folder
2. Make a set (no duplicates) of all the words that occur in the page
3. For each *word* in the set
    a. Add the URL of this page to the list for *word*
4. While there are still pages in the folder go to step 1

Once again, this basic process has issues and limitations which do not consider the advanced structure of web pages and this will also be addressed later on.

2.3.2: Crawling and Indexing in Detail

Crawling is essential to indexing the Web as without the data from the crawl there would be nothing to index. The procedure of crawling can be a very intricate and delicate one for, if any one component of the crawler process does not perform as expected it could cause it to slow down and perform inefficiently or behave impolitely and be banned from many web servers.

2.3.3 Overall Performance

Google (Brin, et al., 1998) acknowledged that hardware performance is a very serious consideration when processing large amounts of data, and that slightly optimizing one area of code can shift a bottleneck from where it was to somewhere else (usually the next slowest part of the system) rapidly. For the initial big crawl that Google carried out, of 24 million pages, they ensured that the indexer was optimised just enough to run faster than the spider so that it would not be the bottle neck and the spider was the limiting factor on performance at the time. It was also observed that disk access is a significant area of their systems that hindered performance, with disk seeks taking around 10ms the data structures and systems were designed to minimise the number of disk seeks, either keeping indexes in memory or optimizing data structures so that sequential access was possible in most cases.

There will always be bottlenecks in large scale web spiders and indexers, and they are likely to occur where less has been invested in a certain resource. Some can be dealt with or tolerated, others may cause the system to stop working, for example a lack of available disk space will stop a spider but bandwidth limitations on an internet connection may cause the spider to run slower, but will not stop it entirely. The ideal situation would be where the Web could be crawled and indexed so quickly due to an abundance of resources that the system could either wait until, of finish, just as the next crawl was due. If the system finishes quickly it will still have bottlenecks, they will just be negligible as they do not affect the planned use of the system.

2.3.4: Bandwidth

Even if not all pages are to be indexed when crawling the Web most must be downloaded so that the links on the pages can be followed as these or subsequent links may lead to a page that does need to be indexed. Based on the previous assumptions after examining reliable sources (Google, 2008) (Majestic-12, 2009) that there are at least 1.3 trillion pages on the Internet the following could be reasonably assumed.

(Building Nutch: Open Source Search, 2004) suggests that on average a web page would be about 10 KB in size, as noted earlier, this estimate is several years out of date, but it is sufficient for this example. The request and data header add about 0.5 KB (based on the earlier HTTP example in section 2.1.4) so it will be assumed (very conservatively) that the total bandwidth to download a page including headers and overhead is 10.5 KB (which is 86 Kbits). So the bandwidth for 1.3 trillion pages would be:

- 13,977,600,000,000,000 Bytes or
- 111,820,800,000,000,000 Bits or
- 12.4 PBytes or
- 111.82 PBits

Currently, one of the fastest home internet connections is about 50 Mbit/sec (Broadband.org, 2009). If it is assumed a connection of this speed is used and that it will always access pages at full speed with no delays, and disregarding the fact that home internet connections have a slower upload rate than download rate, the following calculation holds:

$$111{,}820{,}800{,}000{,}000{,}000_{(data\ to\ download)} \div 50{,}000{,}000_{(connection\ speed)} = 2{,}236{,}416{,}000_{(time\ in\ seconds)}$$

2,236,416,000 seconds is 37,273,600 minutes, or 621,226 hours, or 25,884 days, 70 years.

This figure in reality would be greater because removing the assumptions, and introducing reality would add many time delays. Another factor ignored is the bandwidth taken to find out if a URL contains an image, binary file or other non-html content, as they would need to be ignored too. Only the header of the response needs be retrieved but this is another 0.5KB per request which will add more delays.

Corporate broadband services run much faster, so a faster connection would speed up the process as more concurrent requests could be made to different sites simultaneously, but network and server delays would not be improved.

2.3.5: Storage

Crawling and Indexing all of the Web requires lots of disk space (as well as bandwidth) and there are several considerations to be addressed. The conservative estimate made in section 2.1.1 suggested that 113PB may be required to store all pages for indexing. Other lists and indexes involved may a large volume of space (uncompressed). A good compression scheme could reduce it by up to 75% (Brin, et al., 1998) cutting the space required down to 28.25PB.

No physical disk is currently that big, so one way or another, the data would need to be split up. It is possible that some indexes may be bigger than one physical disk too.

The data collected as well as being vast is quite expensive in terms of resources required to gather it, as it takes a lot of CPU time and bandwidth, so it could be very detrimental if all or part of it were lost. Unfortunately backing up data requires up to double the disk space used to make one whole backup, or less if the overhead of checksums are introduced (AC & NC, 2009).

The Google File System (Ghemawat, et al., 2003) tackles both these issues simultaneously whilst also trying to maximise performance and concurrent use. There are several assumptions made based on how they store data, but it works as follows:

In each cluster there is one master server and multiple data (chunk) servers. The data, saved as files are divided in to 64Mb chunks. Each chunk has a replication count (with a default of 3) and these chunks are stored on at least that number of chunk servers. The master server keeps track of where files exist, and manages locks and access. The concept is highly detailed and the intricate detail of how it works is more than needs to be covered here.

2.3.6: Data structures

Google (Brin, et al., 1998) took much care, even for their first major index to use very carefully designed data structures to store information, they acknowledge that generally the performance of computers improves but that disk seek time is still around 10ms, and for that reason they optimize their structures to avoid disk seeks if possible. Testing whether a URL is new or not could be quite resource intensive, for example if a simple list was created of all URLs seen so far it would then have to be searched each time a new URL was found and as the list got bigger the search would get slower. For text processing and storage there are several common tree based data structures which could prove beneficial for a fast lookup, but if stored on disk they are expensive in terms of space as overhead is added in the form of pointers (Goodrich, et al., 2004).

Google (Brin, et al., 1998) uniquely checksum their URLs which shortens them and allows for a quicker comparison and binary search is used to link the checksum to an ID which can then be used to find details about the URL. Larger data structures which will not need rapidl searching or iterating are compressed to save on resources (disk space), which is a calculated plan as the cost of performance has been traded against disk space gain.

2.3.7: Politeness

Politeness is a term used to describe how a spider behaves when it crawls the Internet. It generally takes into account whether the spider obeys limitation rules of the site (or not) and how aggressive the spider is towards an individual web server – in other words, how often does it try and access pages from that site and considering that it could cause a Denial of Service error for other, *real* users. A polite spider will obey all limitation rules and will not query any individual site too frequently.

(Re)crawling

Assuming a spider uses the basic crawling logic described in section 2.3.1, it instructs the spider to use the next URL off the list to crawl. If the crawler is set to start on the homepage of a fairly large site, there will probably be at least a handful of links on that site that point to other internal pages, and when those pages are followed there will likely be a few more links on each page which are unique and lead to other pages on the site. Before long the crawler will spend most of its time on this one site `until it has visited every URL at which point it will then probably get hung up on another large site.

Consideration must be taken when processing the 'crawl list' so that it is **not** done sequentially, unless the process of adding to this list is not sequential. Though this extra consideration will require processing time, and add general overhead – it is essential to prevent the IP(s) of the spider from being banned by vigilant webmasters who are annoyed by handful of impolite spiders preventing their real users from gaining access (Cody, 2001).

If there is a need to maintain a reasonably up-to-date index then the rate at which the site is recrawled must also be determined and set to a sensible frequency. For a small crawler project, the resources available for the system may force this to be several months or more, but large search engine providers can afford to recrawl at least once a month, if not more frequently. The robots restrictions (discussed later) does allow a crawl and recrawl delay to be specified, however it is not an official extension to the robots specification and for that reason many spiders ignore it (Wikipedia, 2009).

ROBOTS

There is a two part system in place used to explicitly inform spiders (otherwise known as Robots) of what they are **not** allowed to access and to crawl on a particular domain.

One part of this system is known as the Robots Exclusion Standard. There is no official standard or RFC for the Robots Exclusion Standard specification; it was created by consensus in June 1994 (Pages, 1994). It consists of a text file that resides in the root of the website, called 'robots.txt'. The address of the robots file for www.google.com would be http://www.google.com/robots.txt for example.

Robots files can specify instructions for all robots and specific robots. An example would be as follows:

```
#This is a valid comment
User-agent: * #This line says these rules apply to all robots
Disallow: /paths
Disallow: /path/

User-agent: Googlebot
Disallow:
```

The first line is comment, denoted by the # prefix. The comment on the second line is also valid and does not affect the text preceding it. The first user agent specifies a wildcard; this means any spiders that are not otherwise explicitly addressed should use these rules, that means Googlebot will ignore this section as Googlebot has its own specification.

Googlebot is allowed everywhere as the disallow statement is blank, in other words it is **not** disallowed anywhere.

Any other search engine is not allowed to access any path prefixed by the disallow entries:

• /path/file	is not allowed
• /paths/file	is not allowed
• /pathscanbelong	is not allowed
• /pathtest	is allowed

The other part of the system can be implemented in the *head* section of a web page in the form of meta tags. It can tell the spider whether or not the links can be followed or indexed (Wikipedia, 2009). Common values can specify that the page is not indexed at all, the page is allowed to be indexed but no links are followed and that the page is not to be cached. Unlike robots.txt, some spiders do ignore meta tags to preserve the integrity of their results.

2.4: Distributed Computing

Several serious attempts to crawl the Web (many successful) have used distributed computing to achieve this (Majestic-12, 2009) (Brin, et al., 1998) (Building Nutch: Open Source Search, 2004). Due to the vast amount of data collected from a crawl distributed storage is required. Network Attached Storage is still distributed as it spans the data (distributes) over many drives. Google distribute the data and workload over many computers, exact figures are not known as Google keep their current workings very secret but it is clear how much data they must store (as a minimum) and it can be reasonably be assumed that there are lots (possible 800,000 plus (Dean, 2008)).

Another very good reason for using distributed computing (with cheap hardware) is reliability (Dean, 2008). Jeff Dean of Google outlined the failure rates and servers affected that occur within a year of installing 1000 servers:

- Power distribution failures (500-1000 servers)
- 20 rack failures (40-80 servers)
- 12 router reloads (takes down network services)
- 3 router failures (networks become disconnected instantly)
- 1000 individual machine failures
- 1000's hard drive failures

High cost machines could easily have similar specifications however have a greater cost due to more reliable hardware. Cheaper machines mean permanent failures cost lest to rectify, and unlike supercomputers or mainframes, they can be slowly built up. Distributed computing allows for greater redundancy if the infrastructure design is well planned as key points of failure can be avoided by spreading services on systems or geographically, unlike bigger non-distributed systems (such as mainframes).

Chapter 3: Requirements and Analysis

The main aim for this project is to create a well-structured and efficient web spider that is capable of crawling the Web and indexing pages that contain RDFa data. Optionally a front-end will be created which will be in the form of a search service (for example a web based search engine), however this feature will be time permitting and not essential. Comprehensive result ranking will not be considered as this in itself presents an enormous challenge which many organizations have been working on for years and they are continually tweaking and perfecting the methods used.

3.1 Overall Structure

As crawling and indexing the Web can take a while, and may not finish in the time available, there is the need for the data to be accessible before a complete crawl has finished so that it can the completeness and validity of the data can be collected, the following proposes to tackle this problem.

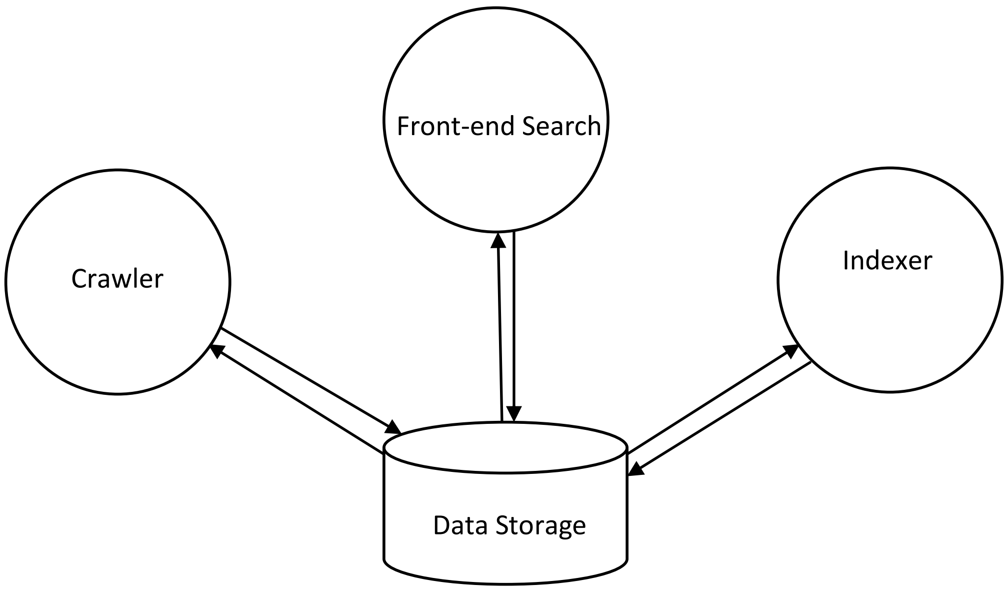


Figure 3.1 The basic structure of the RDFa Spider

The crawler system, indexer system and front-end search provider will all run simultaneously. Due to the current resources available a large crawl could take months, running the systems at the same time will allow nearly immediate access to the data gained providing the indexer is at least as fast as the crawler.

3.2: Crawler Structure

The crawling system needs a distributed structure so that isolated failures do not affect the system as a whole. Based on research it is clear that unexpected failures are frequent, and the crawler needs to be able to cope with different types of failure.

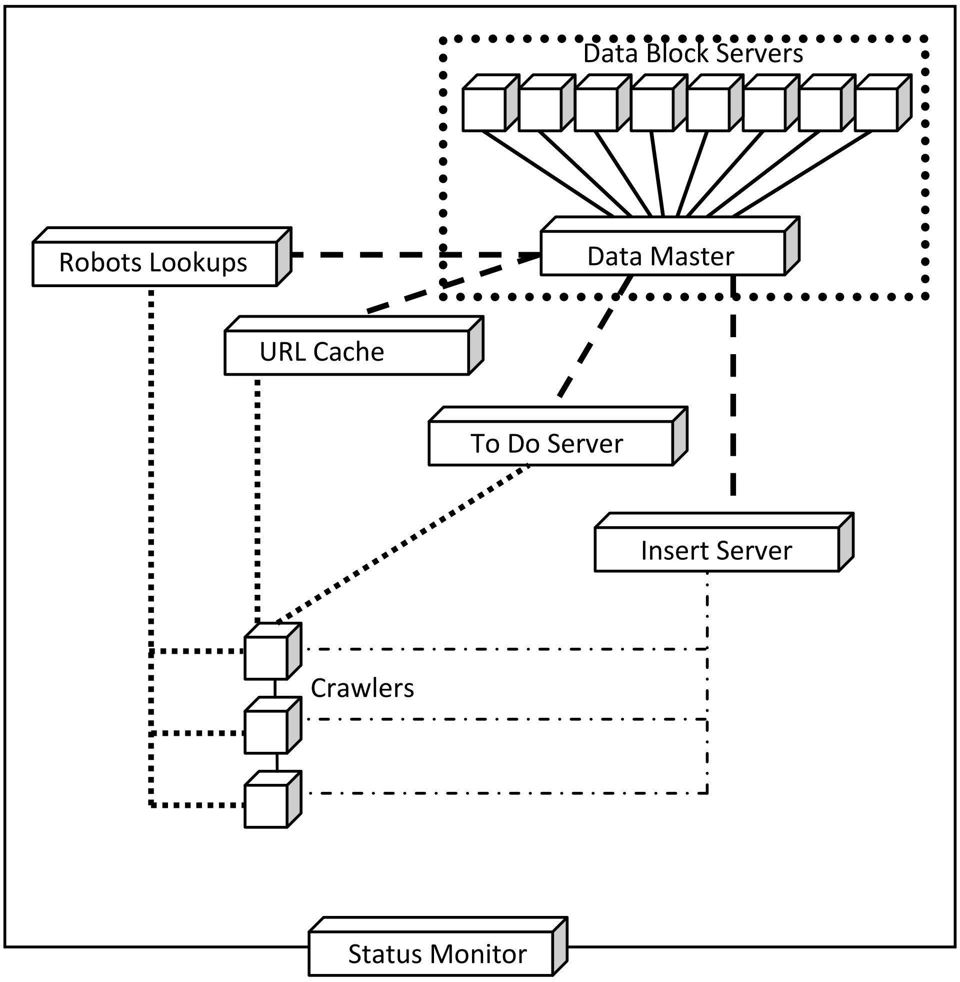


Figure 3.2 – Crawler Connectivity

Figure 3.2 shows the connectivity structure of the crawler system; however the data servers are likely to be a series of flat files on one or more drives unless time permits the implementation of a more advanced storage system.

3.2.1: Crawlers

The crawler process is the one that does the majority of the work, and it relies on all the other Crawler Services. (None of the other services are reliant on each other).

The order of logic will be roughly as follows:

1. While (crawler running)
    a. Get next URL from ToDo Server
    b. Download URL
    c. Parse URLS to Vector<string>
    d. For each URL

            i. Check robots from memory or server
                1. Cache robot in memory if not cached
            ii. Check url cache local or remote
                1. Add to local url cache (last *n* entries) if not cached
            iii. If not cached and not restricted by robots send to Insert
        e. If page contains RDF data save to Repository (data storage)
        f. Clean Expired Caches
2. Go to step 1

This logic is based on the previous system but it has extra local caching steps added as to reduce network and core server load on other services. Extra compensation will be needed to cope with network or external service failures, causing the crawler to suspend its work until connections have been restored.

To speed up the crawling process as much as possible, the method which downloads pages will be enhanced to support common server compression standards so that less bandwidth and time is required to request a page.

3.2.2: Robots Lookup

The crawler needs to be polite in the way it conducts itself around the Internet so that it does not cause webmasters to defend their sites by blocking the crawler, and even though obeying robots is good, the need to check robots can cause problems too. These are addressed below:

Every time a link is discovered by the crawler, a check needs to be performed to ensure that this URL is allowed to be crawled. This must be done by checking the robots file for the domain of the URL, however it would be very inefficient and in itself, impolite, to check the robots file for a domain every time a URL is discovered. Instead, the crawler will need to request the robots data from the Robots server, which will then check its own cache of past lookups. If it has cached the data and the cache entry is not too old then the data will be returned from the cache, otherwise it will be looked up from the domain requested and cached for future use.

If a robots file is not available, either due to not existing or a temporary error, this situation must also be recorded and an appropriate expiry time must be set. For example, if the error is that the file cannot be found on the server then it should be assumed the robots file does not currently exist, but should be rechecked after a while, whereas if the error indicates the file is temporarily unavailable it should be checked more frequently.

To maintain the ability to quickly look up data, several indexes will be used as it can be reasonably assumed that the robots data cache could become very large very rapidly. An on-disk index of all the data will be stored as well as a smaller in-memory index of key separations in the data, for example alphabetic boundaries.

3.2.3: Insert and To Do

A list must be maintained of the URLs that still need crawling, the changes to the list must be atomic, and the retrieval of entries from the list should *not* be sequential so that the crawler avoids getting hung up on one big site.

The Insert and To Do are two separate services, though this is only to assist in serialization of data transactions, in reality they form one system as they manage one list, the list of URLs still to be crawled. New URLs found are added to the end of the list whereas the next URL to be crawled is selected at random, this is to prevent over-crawling one site in a short space of time. It is not a flawless solution as by chance it could select the same site repeatedly, but the random selection has minimal processing overhead compared to calculating an ideal next crawl. When the next URL has been retrieved from the list it is removed so that it cannot be crawled again.

3.2.4: The URL Cache

The crawler needs to know which URLs it has seen, either processed or added to the crawl queue. This cache needs to be able to insert entries and check if they already exist very rapidly with minimal overhead.

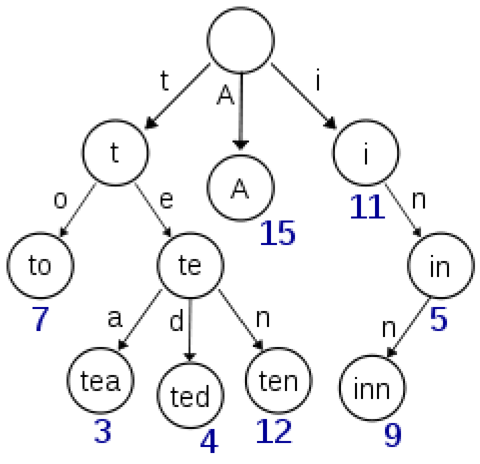


Figure 3.3 – Trie Structure

The URL cache will be based on a trie structure (figure 3.3), however the current implementation has several unnecessary pointers as the current trie is unsorted, being able to iterate in both directions is not required. To aid in faster lookups the strings will also be compressed as heavily as possible using a lossless compression, this will give less tree depth to search and will save space. The URL strings will not need to be decompressed in part, just as a whole.

3.2.5: Data Structures

The crawler will process a considerable volume of data, much of which must be stored. To optimize the operations involved, logical data structures must be used so that mutating and reading the data is as efficient as possible. This section considers the structures required to achieve this.

There are four main types of data that need to be stored, the cache entries, the robots data, the pages crawled (the repository) and the RDFa data extracted. It is possible, if a search interface is implemented, that reverse indexes may also need to be stored.

URL Cache

The trie entry structure will be as follow and each level of the tree will be sorted.

```
nnnnnnnn pppppppp cccccccc iiiiiiii d
```

nnnnnnnn	Pointer to Next	64bit integer
pppppppp	Pointer to Previous	64bit integer
cccccccc	Pointer to Child	64bit integer
iiiiiiii	ID	64bit integer
d	Data	1 character/byte

This structure is 33 bytes per tree entry. It is yet to be seen whether or not this will save space in the long run, but if it does not then it will have to be redesigned. The ID will allow the URL cache to serve as a lookup table linking URLs to other data which can then simply be represented by the URL's ID, for example the URL's page content. During the indexing process this will save disk space as most URLs will be longer than 8 bytes (a 64 bit integer).

Robots

The data extracted from robots files will vary in length and cannot be stored in a static length structure. The indexes will also have dynamic entries as domain length will vary.

The robot data entry itself will be as follows:

lll d_0…d_{lll}

lll	The length of the proceeding data, limited to 16 Mb (3 bytes)
d…	The data itself

As the entry implies, the cache will hold no more than 16 Mb of robots data, in reality this is likely to be more than required, but a two-byte length definition limits to 65Kb which may prove to be too little.

The robots entries themselves do not have any information of which domain they correspond to, this is all saved in the on-disk index, which has the following entry format:

m n_0…n_m oooooooo

m	The length of the domain (256k limit)
n…	The domain
oooooooo	Data offset in main data file

Repository

The pages which have been crawled and contain RDF data need to be stored for indexing.

The repository entry will be stored in the following format:

iiiiiiii lll d_0…d_{lll}

iiiiiiii	The URL ID	64 bit integer
lll	The length of the data (limited to 16Mb)	24 bit integer
d…	The data itself	

The structure deign does not allow for pages greater than 16Mb in size to be stored, and this is not likely to cause a problem as most pages are small, the rich content (such as images etc) is often the bulkier set of data and that would not be collected.

RDF (Triple Store)

The RDF data collected will need to be stored, and there is a commonly used system called an RDF Triple Store, for this project such a system needs to be utilized.

Compression

Resources for this project are limited, specifically disk space – it is therefore necessary that the system is able to compress its assets if disk space becomes limited.

The disk space available for this project will be limited, and currently it is estimated to be just over 1Tb, due to this it is quite likely that indexes and data may need to be compressed due to resource limitations. Indexes will be compressed using Huffman coding as this allows sections to be decompressed without having to decompress all the data. Stored pages will be compressed to a higher degree as they will only be decompressed as a whole. Compressing data will also benefit other areas of the system, such as the network, as there will be less data to shift around if it decoded upon receipt.

3:3: Indexer Structure

The indexer is likely to be a resource intensive process, especially with regards to CPU time. For this reason the indexer needs to be capable of running on several systems concurrently and in such a way that the instances do not interfere with each other.

The indexer will have a less distributed structure compared to the Crawler; it will run as a single service accessing the data store. It will be capable of running concurrently on several machines to improve overall processing time.

It will need to process all pages in the repository, collecting all the RDFa tags used on the pages and sending them to a triple store.

3.4: Storage

The volume of data to be stored could be large and filling a disk could stall the system if the design of the system does not cope with spanning data over multiple disks and servers so it is essential that data can be reliably be divided and stored.

There are various different types of data to be stored, and this data will be stored in appropriate files on some of the servers. The need to split up data is probably going to be mandatory as the crawling and indexing will produce lots of output data, probably more than will fit on one disk, so the data will have to be split over many drives and servers. Logically splitting indexes and repositories will allow them to be search simultaneously and hence save time.

If time permits a basic distributed file system will be implemented. For easy implementation it will behave like a standard C++ stream, but when saving or reading data it will access the server in which the data is stored in (as determined by a master index) and relay the data from the storage node to the client.

3.5: Searching

The search feature will be implemented if time permits (it is not a *requirement* however it would provide a user friendly interface with the data). It may require the creation of other systems, and changes to existing ones. The indexer will need to create reverse indexes that are rapidly searchable. The search interface will be a web site where the user can input a query, with the option to search data from specific ontologies.

3.6 Evaluation

The final system will need to be evaluated to determine whether it has become a suitable solution for the problem. It will be compared against the original requirements, where an analysis will be made comparing the similarities and differences, why they exist and how they have or have not benefitted the project.

CHAPTER 4: PROGRESS

Considerable progress has been made so far, the system has been able to run a simple distributed spider on a cluster of 10 computers and crawled about 400,000 URLs during a brief test. The programming language used so far and will continue to be used is C++, the main motivation for using it is because it is very portable between common platforms, but this is strongly backed up by the high performance of well written code due to it not being interpreted or running in a virtual environment like many popular languages today (Java and C# are two examples). There is currently ongoing consideration for using Python for part of the crawler as natively it supports all the functions required whereas C++ requires many custom or external libraries; fortunately Python can also run on both Windows and Linux. In this project the systems and services will be run on Linux so that software licensing costs are not an issue and because it is heavily customizable for both flexibility and performance.

4.1: HARDWARE AND INFRASTRUCTURE

At this point, a small cluster of 24 computers are available which can be customized and used for running this system. They have been set up carefully so that they can easily be reconfigured and reinstalled if the need arises from a local software repository, a task which completes in less than 5 minutes. The CentOS linux operating system installed has been customized to allow custom tasks to run when the system starts. They are connected via a 100Mbit 48 port managed switch with dual gigabit uplink. The plan is to install them at the University of Sheffield as their internet connection's ability is far greater and they have a server room suitable for cooling, powering and connecting the cluster. On boot the machines are configured to download a script (the *boot script*) from a master server, that script then copies any new files to the machine and executes any applications listed for that node.

4.2: SOFTWARE AND TESTS

The data backend is currently MySQL which is running on a powerful machine. All the data is currently stored on one disk. This server holds a database which stores all data from the crawl and a few configuration tables, except the 'list of URLs seen'.

Currently there are several test applications and common classes for various parts of the system which perform a required function, but they do so crudely and without optimization. The existing applications are detailed below:

4.2.2: RWS_COMMON

The common superclass contains many classes which are provided by higher level managed languages, including:

• download_file	Downloads a file to a string, and supports timeouts
• global_settings	Allows reading of settings from the database
• file_pointer_operations	Converts arrays to 64 bit integers
• logging	Writes log entries to the database
• mysql_client	Wraps the c mysql library
• robot	A structure to load and check robot data

• string_functions	Common string functions like replace and split
• tcp_client	A synchronous tcp client library
• tcp_server	An asynchronous tcp server with overridable processing function
• unix_environment	Provides access to unix OS statistics and data
• url	A structure to break down a url and analyze parts of it

4.2.3: RWS_HEARTBEAT_SERVICE
The heartbeat service is started by the boot script; it reads from the database the host and IP of the system manager and connects to it. It maintains a connection persistently, if the connection is lost it retries repeatedly until the connection is reestablished. Every second it sends the server time, free, used and total memory, CPU usage, CPU speed and the processes running. This allows all the servers to be monitored from one place.

4.2.4: RWS_CACHE
The cache service implements a trie structure and it listens on a TCP server. It responds to requests to check if a URL has already been seen, it checks the trie and sends the result. If not seen it adds it to the try as it assumes that a request implies it has now been seen. The crawler checks every URL it discovers against the cache.

4.2.5: RWS_ROBOTS
The robots service acts like a form of caching proxy. The crawler asks the robots service for robot data for a specific domain, for which it then queries the database for. If the data exists it returns it, otherwise it requests the data from the domain's webserver and stores it. All stored data expires after a predetermined amount of time.

4.2.6: RWS_SQL_INSERT
The SQL insert server listens on a TCP server for URLs to be crawled. It listens asynchronously then inserts the URLs in a serialized fashion to the 'To Crawl' list in the database. The crawler sends the URL to this server after checking the cache and robots.

4.2.7: RWS_TODO
The ToDo server returns the next URL to be crawled via a TCP server. All the crawlers call this when they have finished with one URL and needs another. The ToDo server requests a random URL from the 'To Crawl' list, removes it from the list and sends it to the client.

4.2.8: RWS_CRAWLER
The crawler ties all the other services together. It currently runs on 20 threads, and each thread does the following:

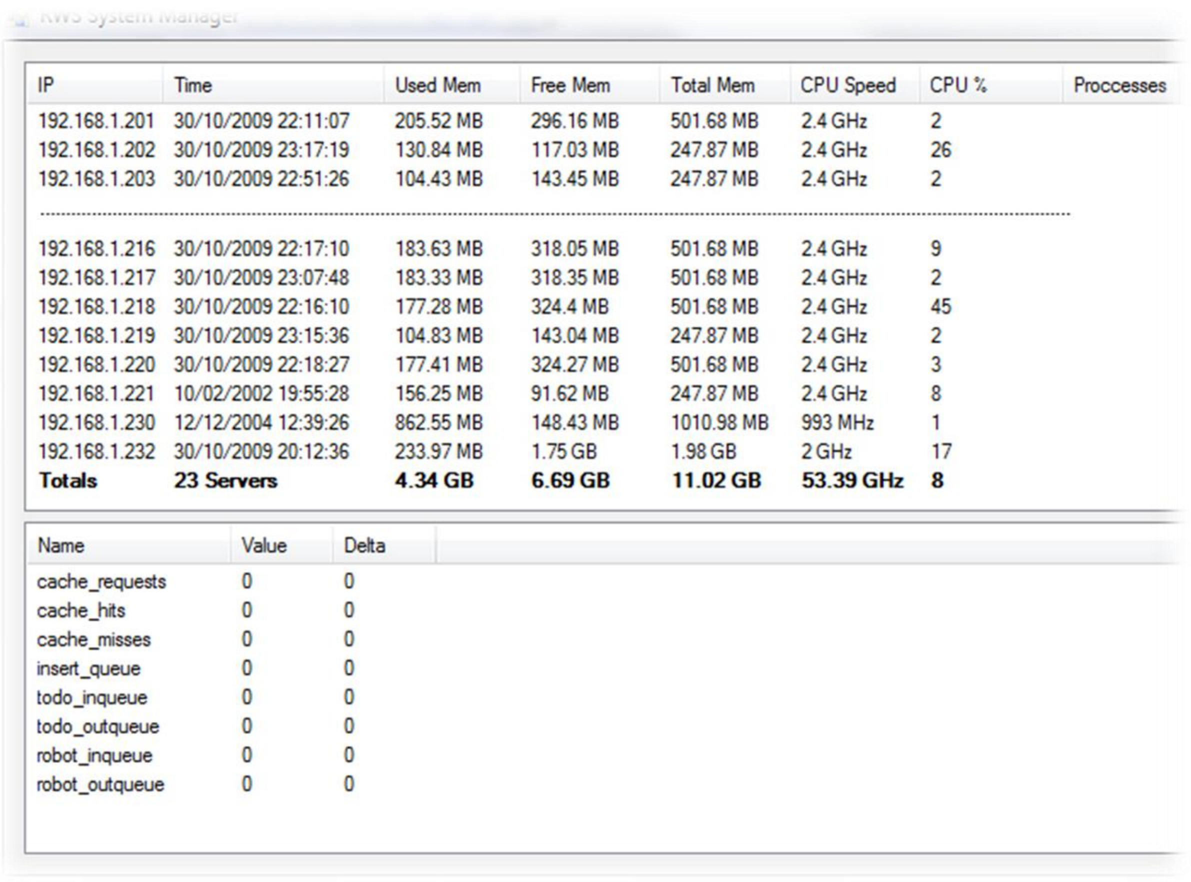
1. Request the next URL from the ToDo Server
2. Download the URL
3. Parse the URL
4. For each link:
    a. Check its valid
    b. Check robots for the link
    c. Check the cache

   d. If b and c are ok then send to SQL Insert
5. Go to step 1

Unfortunately it currently gets stuck at some point, it seems to be caused by the robots server but this is something that needs investigation.

4.2.9: System Manager

The system monitor runs a TCP server and waits for connections from the heartbeat service. It displays the data received in a detailed list with a total or average at the bottom. It also shows the common variables which are pushed to the database by many services such as cache size, URLs crawled and more.



IP	Time	Used Mem	Free Mem	Total Mem	CPU Speed	CPU %	Proccesses
192.168.1.201	30/10/2009 22:11:07	205.52 MB	296.16 MB	501.68 MB	2.4 GHz	2	
192.168.1.202	30/10/2009 23:17:19	130.84 MB	117.03 MB	247.87 MB	2.4 GHz	26	
192.168.1.203	30/10/2009 22:51:26	104.43 MB	143.45 MB	247.87 MB	2.4 GHz	2	
192.168.1.216	30/10/2009 22:17:10	183.63 MB	318.05 MB	501.68 MB	2.4 GHz	9	
192.168.1.217	30/10/2009 23:07:48	183.33 MB	318.35 MB	501.68 MB	2.4 GHz	2	
192.168.1.218	30/10/2009 22:16:10	177.28 MB	324.4 MB	501.68 MB	2.4 GHz	45	
192.168.1.219	30/10/2009 23:15:36	104.83 MB	143.04 MB	247.87 MB	2.4 GHz	2	
192.168.1.220	30/10/2009 22:18:27	177.41 MB	324.27 MB	501.68 MB	2.4 GHz	3	
192.168.1.221	10/02/2002 19:55:28	156.25 MB	91.62 MB	247.87 MB	2.4 GHz	8	
192.168.1.230	12/12/2004 12:39:26	862.55 MB	148.43 MB	1010.98 MB	993 MHz	1	
192.168.1.232	30/10/2009 20:12:36	233.97 MB	1.75 GB	1.98 GB	2 GHz	17	
Totals	**23 Servers**	**4.34 GB**	**6.69 GB**	**11.02 GB**	**53.39 GHz**	**8**	

Name	Value	Delta
cache_requests	0	0
cache_hits	0	0
cache_misses	0	0
insert_queue	0	0
todo_inqueue	0	0
todo_outqueue	0	0
robot_inqueue	0	0
robot_outqueue	0	0

Figure 4.2: System Manager

CHAPTER 5: CONCLUSIONS AND PROJECT PLAN

At this point there is a very basic spider (crawler) which is capable of running several crawlers on a distributed computer system. It has several performance and reliability related issues which need addressing. Research indicates that there are several ways that the current infrastructure can be improved, thus increasing reliability and performance. The basic applications that make up the spider will be rewritten so that they can be improved from the bottom up, this will add much of a time delay because they mainly tie together common classes and functions.

A web based interface may be created to allow access to the data (a search engine) if time and resources permit.

5.1: PROGRESSION PLAN

	Task	Jan 2010				Feb 2010				Mar 2010					Apr 2010			
		4	11	18	25	1	8	15	22	1	8	15	22	29	5	12	19	26
1	Revise Heartbeat System	■	■															
2	Improve Robots Cache	■	■															
3	Improve URL Cache		■	■														
4	Replace MySQL 'URL List' with flat files			■	■													
5	Revise Crawler			■	■													
6	Create Repository			■	■													
7	Begin Crawling				■	■	■	■	■	■	■	■	■	■				
8	Create Indexer	■	■															
9	Begin Indexing					■	■	■	■	■	■	■	■	■				
10	Update System Monitor		■															
11	Evaluate System Performance									■	■							
12	Performance Enhancements											■	■					
13	Implement Search Front End *								■	■	■	■	■	■				
14	Evaluate Project												■	■				
15	Write Final Report													■	■	■	■	■

*Not compulsory and time permitting

REFERENCES

AC & NC. 2009. Raid Levels. *AC & NC.* [Online] 1 1, 2009. [Cited: 11 21, 2009.] http://www.raid.com/04_01_0_1.html.

Alexa. 2009. Top Sites. *Alexa.* [Online] 18 11, 2009. [Cited: 18 11, 2009.] http://www.alexa.com/topsites.

Brin, Sergey and Page, Lawrence. 1998. *The Anatomy of a Large-Scale Hypertextual Web Search Engine.* s.l. : Standford University, 1998.

Broadband.org. 2009. Fastest Broadband Speeds - Top 3 Speeds. *Broadband.org.* [Online] 11 1, 2009. [Cited: 11 25, 2009.] http://www.broadband.org/fastest_broadband.html.

Building Nutch: Open Source Search. **Cafarella, Mike and Cutting, Doug. 2004.** 2, s.l. : ACM, 2004, Vol. 2. 1542-7730.

Cody, Daniel. 2001. Using Apache to stop bad Robots. *Evolt.org.* [Online] 08 22, 2001. [Cited: 11 25, 2009.] http://evolt.org/article/Using_Apache_to_stop_bad_robots/18/15126/.

Ghemawat, Sanjay, Gobioff, Howard and Leung, Shun-Tak. 2003. *The Google File System.* s.l. : ACM, 2003.

Goodrich, Michael T. and Ramassia, Roberto. 2004. Tries. *Data Structures and Algorithms in Java.* s.l. : Wiley, 2004.

Google. 2009. Introducing Rich Snippets. *Google Webmaster Central Blog.* [Online] 5 12, 2009. [Cited: 11 19, 20009.] http://googlewebmastercentral.blogspot.com/2009/05/introducing-rich-snippets.html.

—. 2008. We knew the web was big. *The Official Google Blog.* [Online] Google, 7 25, 2008. [Cited: 11 18, 2009.] http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html.

Internet Archive. 2009. PetaBox. *Internet Archive.* [Online] 11 20, 2009. [Cited: 11 20, 2009.] http://www.archive.org/web/petabox.php.

Internet World Stats. 2009. Internet Usage Statistics. *Internet World Stats.* [Online] 11 18, 2009. [Cited: 11 18, 2009.] http://www.internetworldstats.com/stats.htm.

Lardinois, Frederic. 2009. Google Patents Its Homepage. *ReadWriteWeb.* [Online] 9 2, 2009. [Cited: 11 20, 2009.] http://www.readwriteweb.com/archives/google_patents_its_homepage.php.

Majestic-12. 2009. Majestic SEO. *Majestic SEO.* [Online] 11 18, 2009. [Cited: 11 18, 2009.] http://www.majesticseo.com/.

Matthew Komorowski. 2009. A History of Storage Cost. *mkomo.com.* [Online] 07 01, 2009. [Cited: 11 20, 2009.] http://www.mkomo.com/cost-per-gigabyte.

Network Working Group. 1999. Hypertext Transfer Protocol -- HTTP/1.1. *IETF.* [Online] 5 1, 1999. [Cited: 11 22, 2009.] http://tools.ietf.org/html/rfc2616.

—. 1998. RFC2460. *IETF Tools.* [Online] 12 1, 1998. [Cited: 11 20, 2009.] http://tools.ietf.org/html/rfc2460.

Pages, The Web Robots. 1994. A Standard for Robot Exclusion. *The Web Robots Pages.* [Online] 5 1, 1994. [Cited: 11 25, 2009.] http://www.robotstxt.org/orig.html.

W3C. 2008. RDFa Primer. *W3C.* [Online] 10 14, 2008. [Cited: 11 19, 2009.] http://www.w3.org/TR/xhtml-rdfa-primer/.

Wikipedia. 2009. Meta Element. *Wikipedia.* [Online] 11 14, 2009. [Cited: 11 25, 2009.] http://en.wikipedia.org/wiki/Meta_element#The_robots_attribute.

—. 2009. Robots exclusion standard. *Wikipedia.* [Online] 11 12, 2009. [Cited: 11 23, 2009.] http://en.wikipedia.org/wiki/Robots_Exclusion_Standard.