COM3021 - RDFA WEB SPIDER

http://www.rdfas.com

Title: RDFa Web Spider
Name: Paul Ridgway
Supervisor: Fabio Ciravegna
Module Code: COM3021
Date: 5th May 2010

This report is submitted in partial fulfilment 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 dissertation 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 dissertation have been used with the explicit permission of the originator (where possible) and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this dissertation and the degree examination as a whole.

Name: Paul Ridgway

Signature:

Date: 5th May 2010

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 XHTML 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 and providing more research crawling and indexing large numbers of pages and enormous amounts of data.

ACKNOWLEDGEMENTS

Most importantly I would like to thank **Fabio Ciravegna**, my supervisor, for finding a home for my 24 servers, giving me advice on sematic web-related issues and guiding me through the structure of the report. I would also like to thank:

The DCS Support Staff

Specifically Dave Abbott for agreeing house my servers in the DCS Server room and Chris Stoddart for allowing me access to the servers when things needed fixing.

Amanda Taylor

My girlfriend, for putting up with my constant rambling about memory leaks, data structures, network protocols, efficiency and all the other strange issues that had to be dealt with during the project.

James Gilbert, David Gill and Daniel Hough

My Housemates, for providing a constant source of distraction when writing got tiresome.

CONTENTS

FIGURES

TABLES

APPENDICES

GLOSSARY

AJAX

Asynchronous JavaScript and XML – a group of related web development technologies used in creating interactive client-side applications.

CALL BACK

A reference to a piece of executable code that is passed as an argument to other code.

CHECKSUM

A fixed size 'signature' computed for some data for detecting accidental errors potentially introduced during its transmission.

CRAWLING

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

DNS

A hierarchical naming system for internet resources.

ESCAPE CHARACTERS

Escape characters identify the start of a character sequence which should be interpreted differently from if the same characters occurred without the escape character.

FUSE (LINUX)

A kernel module for Unix-like operating systems that lets users create file systems without editing kernel code.

HTTP

Hyper Text Transfer Protocol – the application layer protocol for interacting with web servers.

(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.

MUTEX

A mutual exclusion system used to ensure exclusive access to a resource on a concurrent or multithreaded system.

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 XHTML web pages.

REPOSITORY

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

REVERSE INDEX

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

REWRITE RULES

Used in web servers to map virtual URLs to actual resources.

SEGMENTATION FAULT

Also known as an access violation, can occur when a program attempts to access a memory location that it is not allowed to.

SESAME

A Java Servlet/Tomcat based Triple Store.

TCP

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

TRIPLE

In the context of RDF a triple is a collection of three pieces of data – a subject, a predicate and an object.

TRIPLE STORE

A purpose built database for storing and querying RDF data

THE WEB

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

USER AGENT

In the context of this report a User Agent is a string used to identify a HTTP client.

CHAPTER 1: INTRODUCTION

The Web has grown exponentially since its conception and it is now extremely large, impossible to quantify accurately. 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 an application called 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 where 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 known as RDF and RDFa 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 RDFa data to potentially allow for a detailed search index and interface.

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

Furthermore there is not much published research on Spiders and Web Crawlers which tackles in depth the more specific issues such as identifying unique URLs and ordering a URL queue to diversify it as much as possible. Hopefully this paper will make a viable contribution to Spider research and provide a good starting point for others to build upon.

The details are broken down in to several sections in this report. Chapter 2 reviews current practises and background material used in creating a spider. Chapter 3 outlines the requirements of this project. Chapter 4 maps out a plan for the design and architecture of the overall system. Chapter 5 discusses actual implementation of the system covering problems encountered. Chapter 6 details the results of various tests run when developing the system and results from running the crawler and chapter 7 concludes the main points in this report.

CHAPTER 2: LITERATURE REVIEW

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 WORLD WIDE 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 INTERNET

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 from Web Servers.

The Internet uses a numerical addressing system which allows computers to connect directly to each other. The system currently in use (called IPv4) uses addresses of the format a. b. c. d where a, b c or d are integers between 0 and 255 (with some restrictions), and is basically a 32 bit address. This means the max number of available addresses is $2^{32} = 4{,}294{,}967{,}296$ without restrictions. In practice there are fewer due to reserved blocks and unusable broadcast addresses.

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 $2^{128} = 3.4 \times 10^{38}$ – which should last much longer than IPv4.

2.1.2: THE WEB

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, (more) authoritative one 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. But these changes apply to all users and services that access the Internet, which is why transitions are often slow.

2.2: Accessing The Web

2.2.1: Accessing Pages

Retrieving a web page from the Internet is a process that requires multiple steps. Here 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

Uniform Resource Locators

Uniform Resource Locators (or URLs) are used to identify the location of a specific page on the internet. Web Page URLs are of the format:

$$http://server[:port]/folder/page/?this = querystring\&more = ok\#fragment$$

The *http* prefix, otherwise known as the protocol, 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, the *http* protocol 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.

The substring *?this=querystring&more=ok* is the querystring, which allows parameters to be passed to the page. Anything before the ? (or #) is considered the page address, anything after is not. Querystrings are name and value pairs in the form name=value and several can be used separated by an ampersand symbol. The limit on the querystring size is the same as imposed by the URL length, less the rest of the URL. This URL length limit is not fixed; different browsers and webs servers have their own restrictions.

Finally the section *#fragment* is the fragment. Traditionally it identifies a section of the page for the browser to ensure is visible. The sections are named and defined in anchor tags in the html and referenced by using the name as the fragment. More recently they have also been used in pages using AJAX to add a form of persistence using it to store a path or parameters so that if the page is refreshed the page's customized content does not reset. Facebook and Google Mail are two big sites which use both AJAX and fragment parameters.

A URL can only contain specific characters, any others must be escaped. The server (or domain) must contain only a-z, 0-9 and dashes some UTF-8 characters can be escaped for other languages; any capital letters can be used but will be reduced to lower case when normalized. Folder and page names can only contain valid file system name characters unless the server can translate them, slashes are used to denote file system hierarchy. Querystrings often need to contain data that is outside of the allowed characters range, in which case they are escaped using percent encoding (Berners-Lee, 2005). Fragments are not often sent to the

server, so they can contain characters that can occur in the page mark up. Unreserved characters consist of A-Z, a-z, 0-9,_, -, . and ~; domains, folders and querystrings can only contain these characters. There are a small set of reserved characters including % (used for percent encoding), ? used to denote the start of a querystring section, & to delimit querystrings and # to identify the fragment section.

URL NORMALIZATION

A URL for a specific page can take, in theory, an infinite number of forms. A resource called *demo.htm* in the folder *test* on the server at *www.example.com* could have the URL of *http://www.example.com/test/demo.htm*. However, the following URLs (including an infinite number of others) would also resolve to that resource:

1. *http://www.example.com/test/nothing/../demo.htm*
2. *http://www.example.com/nothing/../test/demo.htm*
3. *http://www.example.com/./test/demo. htm*
4. *http://www.example.com/././test/demo. htm*
5. *http://www.example.com/test/./demo. htm*
6. *http://www.example.com/./test/./demo. htm*

On a page, a hyperlink may specify a URL relative to the page, or an absolute URL, which is mainly why ".." and "." are used in URLs. The ".." notation specifies the parent folder, so example 1 goes to the parent of */test/nothing/* which is */test/* which is why */test/nothing/../demo.htm* is the same as */test/demo.htm*. The "." denotes the current folder, for example, if a link was on */test/demo.htm* and linked relatively to "." then it would simply link to */test/,* a link to *./another.htm would link to /test/another.htm. H*owever, in this case the "." prefix is unnecessary as linking to *another.htm* would give the same result. Hence "/./" is the same as "/" so */test/./demo.htm* is identical to */test/demo.htm*.

In practise "." does not come up much, but ".." does. This poses a problem for crawlers, as it is best practise to avoid crawling a page more than once in a certain window of time. If a crawler discovers several variations of a URL and cannot determine that they are all equal it could crawl the same page many times needlessly. The solution is to normalize (or canonicalize) the URL, which is the process of modifying the URL to standardize it in a consistent manner.

Normalization is used by web crawlers and search engines to avoid crawling and indexing the same page more than desired, however browsers may also normalize URLs to determine if a link has been visited or cached (Wikipedia, 2010). Browsers may also normalize URLs so that when the resource is requested from the server it is not needlessly redirected to the expected (normalized) URL. This is because many clients will only accept a finite number of redirects to avoid getting stuck in a redirect loop, so if the server redirects the client it could count against the number of redirects carried out.

There is a standard normalization procedure defined in (Berners-Lee, 2005) which is made up of many steps. In theory they are all optional, however most should be used for the best results. They are as follows:

- Convert the scheme (protocol) and host (server) to lowercase
  HTTP://WWW.Example.Com/ becomes *http://www.example.com/*
- Add a trailing slash if there is no path specified to indicate the root directory.
  http://www.example.com becomes *http://www.example.com/*
- Remove directory indexes
  http://www.example.com/index.html becomes *http://www.example.com/*

  However deciding what is and is not an index is potentially difficult as each server can have its own list of 'default (or index) documents'. Even though index.html may be an index page on one server, it may not be on another, resulting in a document not found (404) error if the URL is requested after normalization.
- Capitalize letters in escape sequences
  http://www.example.com/%3a becomes *http://www.example.com/%3A*
- Remove the fragment
  http://www.site.it/page.htm#section becomes *http://www.site.it/page.htm*
- Remove the default port
  http://www.example.com:80/ becomes *http://www.example.com/*
- Remove dot-segments ("." and "..")
  http://www.site.com/../1/../2/3 becomes *http://www.site.com/2/3*
- Remove "www" as the first domain label
  http://www.example.com becomes *http://example.com* or vice versa

  This is only useful if example.com and www.example.com return the same content, which is often hard to test, for even if the home page does obey this rule, other pages may not. It can also be problematic as the same page requested at two slightly different times could return different content, for example if the page includes the date and time or generation time.
- Sorting querystrings
  http://www.site.com/page?name=test&id=10 becomes

  http://www.site.com/page?id=10&name=test

  The order of querystrings usually does not matter to the server processing the request, so when a hyperlink is created the order naturally does not matter so ordering them consistently during normalization is suggested.
- Removing arbitrary querystrings
  http://www.site.com/page?name=test&this=pointless becomes

  http://www.site.com/page?name=test

  Most pages using querystrings only use a few select names, so unused ones could be removed. However a client is not likely to know in advance which are needed and which are not, so this is not common, but may be done server-side through rewrite rules.
- Removing default querystring variables

http://www.example.com/sort=asc becomes *http://www.example.com/*

If asc is the default value for sort then the page will render the same whether it is there or not. However once again a client is not likely to know this.

- Removing "?" if there is no querystring
  http://www.example.com/? becomes *http://www.example.com*

THE HYPERTEXT TRANSFER PROTOCOL

Most systems that rely or utilise communications over a network follow a protocol. The Hypertext Transfer Protocol (HTTP) is an Application Layer protocol which is the primary mechanism used to retrieve web pages, amongst other things. HTTP uses a request-response principal which is common in client-server network computing. The Hypertext Transfer Protocol determines the format and parameters the client and server can use when forming and processing requests and responses (Fielding et al., 1999)

A HTTP REQUEST

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 (default is 80 if no port is 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 (or verb), 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' or

CRLF. 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 by the client and if the server supports it. 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 server is inundated with requests there could be a long delay, or possibly an 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

```

The request ends in a double CRLF – the blank line signifies the end of the request

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=RrIw8wOmt0iMrP
bT; 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
Length: 1234

User-agent: *
Disallow: /search

```

The response ends in a double CRLF.

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.

2.2.2: SIZE

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, (Cafarella and Cutting, 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 6 years old, with advances in network technology, the average internet connection is currently much faster and it is clear that web pages are more content rich now, with images and other included files such as JavaScript and Cascading Style Sheets (CSS), 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 around 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 a vast underestimate 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 and Page, 1998) and in the space of 10 years it has risen to well over 1 trillion. Google stores 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.

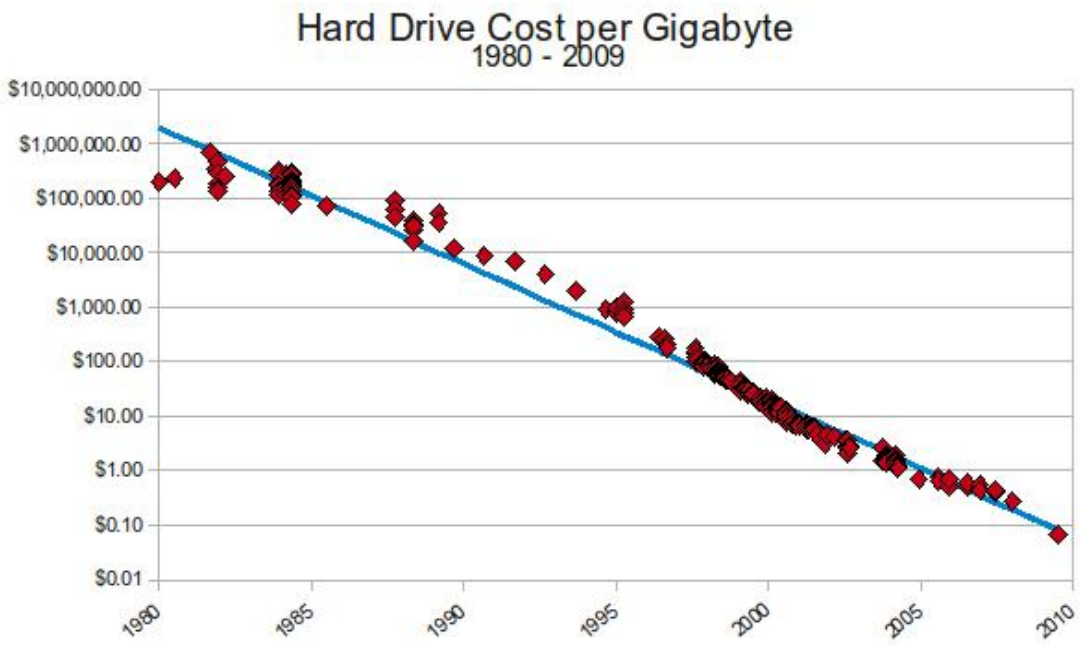


FIGURE 1: A GRAPH OF THE EFFECTIVE COST PER GB OF HARD DRIVE STORAGE AGAINST TIME (MATTHER KOMOROWSKI, 2009). PERMISSION AQUIRED.

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 since 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 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), however they do only store html copies of pages, no other media such as images, video, style sheets etc.

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 (electrical) power. All of these issues will be addressed later on.

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. This consideration helps illustrate how useful search services like Google are, especially when they can produce results in well under one second.

2.2.3: 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. Research from a survey (iProspect, 2006) carried out in 2006 states that the majority users (62%) of a search engine will only look at the first page of results (usually the first ten) results. The survey was also conducted in 2004 and 2002 and found that as time went on, more people were only looking at the first page of results and less people (from 19% down to 10%) were prepared to go beyond the third page of 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 because 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 to which they may not actually be related in order to increase traffic, sales or to capitalize from advertising.

2.3: 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 must be indexed automatically due to its size, computers are left to analyse 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 tag's content; 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.

2.3.1: 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.

2.3.2: 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:

```
<div class="right" about="http://www.ivan-herman.net/foaf#me"
typeof="v:Person foaf:Person">
```

This code, taken from Ivan Herman's page at W3C (*http://www.w3.org/People/Ivan/*) indicates that the div specified and all content in it is a type of "v:Person" and "foaf:Person" entry. The fact that they contain the word person does not mean that they are actually about a Person. Their definition is in the namespaces identified earlier in the page (shown below) which link to URLs and the content at these URLs helps define the relations.

```
xmlns:foaf="http://xmlns.com/foaf/0.1/"
xmlns:v="http://rdf.data-vocabulary.org/#"
```

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 designed to be read by computers and not humans.

2.3.3: PARSING RDFA AND STORING RDF DATA

A system has emerged 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, n.d.). There is sometimes a fourth element, the context – technically a system that can store these is called a Quad Store, though many support the fourth element they are still known as Triple Stores. Storing the data is the less technically challenging part; the feature of many RDF Triple Stores is that they allow for logical querying in a Prolog/SQL style syntax call SPARQL. SPARQL allows for logical relationships to be created as graphs. To be able to store RDFa in a Triple Store its containing page needs to be parsed and the RDFa converted to RDF.

2.4: 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. 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 because 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.

2.4.1: BASIC CRAWLER AND INDEXER

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.

2.4.2: 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 it tries 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 re-crawled 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 re-crawl at least once a month, if not more frequently. The robots restrictions (discussed later) does allow a crawl and re-crawl 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.

2.4.3: DATA STRUCTURES

Google (Brin and Page, 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 and Ramassia, 2004).

Google (Brin and Page, 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 rapidly 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.

2.4.4: 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.

(Cafarella and Cutting, 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:

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

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.

2.4.5: Storage

Crawling and Indexing the entire 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 and Page, 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, Gobioff and Leung, 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 intricacies are more than needs to be covered here.

2.4.6: Overall Performance

Google (Brin and Page, 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 rapidly shift a bottleneck from where it was to somewhere else (usually the next slowest part of the system). 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 crawler so that it would not be the bottle neck and the crawler 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, or 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.

2.5: DISTRIBUTED COMPUTING

Several serious attempts to crawl the Web (many successful) have used distributed computing to achieve this (Majestic-12, 2009) (Brin and Page, 1998) (Cafarella and Cutting, 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 less 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

3.1: OVERALL STRUCTURE

As with any complicated and/or large scale project or challenge there are many potential solutions and at first it may be hard to determine the long term implications or differences. This means that some preliminary design choices may become crucial to the project's success later on. This project has three main areas of focus, a web crawler, an indexer and scalability as a whole. These core items can be subdivided into many smaller areas as detailed in the following sections.

3.1.1: CRAWLING

Crawling can be an ambiguous term as it is often used to refer to the whole process of crawling, indexing, and sorting web pages, or more specifically it is used in reference to downloading pages from a queue. In this paper, when referring to the system as a whole the aim is to use the term Spider. "A [Web] crawler is a computer program that browses the World Wide Web in a methodical, automated manner." (Wikipedia, 2010) - Wikipedia's definition is unclear as *browsing* involves some form of parsing, sometimes done in the indexer. The Crawler, as talked about here, is a system which goes from page to page discovering links and finding data to index; it encompasses subsystems like the URL queue and duplicate URL checking.

3.1.2: INDEXING

Data from pages retrieved as part of the crawling process will need to have information deemed useful extracted and stored. This step can be broken down in to two parts, parsing the page to locate and extract the data, and then storing the useful information extracted. Parsing is a task specific activity, for example parsing for a search engine would require forward and reverse indexes to be generated. This also applies to storage, search engines will often store the whole page so that search results can reference points in the page where the search terms occur. The data being indexed can define when indexing occurs in the overall process, this is discussed later.

3.1.3: SCALABILITY AND SPEED

There are many delays that the crawler may encounter and resource limits could also impose limits on the crawler such as limited disk space or bandwidth. As any one server has limited resources, even if it is fully upgraded, a Spider needs to be scalable so that it can run across a number of servers allowing it to go about its task as quickly and efficiently as possible.

3.2: CRAWLING

3.2.1: TRACKING LISTS

Several aspects of a Crawler need to keep track of large lists of data, most commonly URLs. The various aspects of a crawler require the use of lists for different purposes and therefore the lists in each area require specific features or enhancements. The term list is used here as an umbrella term, the ideal solution may not eventually be a list; it may be a tree or some other data structure.

The implementation of these lists will need to be done carefully with respect to their required features, with special attention given to performance as a poorly implement list may prove to be a bottleneck.

3.2.2: Queue URLs to Crawl

URLs that are discovered while crawling a page also eventually need to be crawled. A list of these URLs would need to be maintained; ideally removing entries once they have been crawled. As discussed in the next section, organization of the list can be very important for preserving crawler politeness.

3.2.3: Being polite

A crawler must avoid crawling a particular site (or server) too often (over-crawling) as well as not trying to access restricted areas. Failure to be polite is likely to cause a crawler to be banned from accessing sites - if the crawler does not obey rules correctly then it is probable that the IP(s) of the crawler will be blocked.

When a page is crawled, unless it is a link directory it is quite likely that most links on that page will link back to other pages on the same domain. This means that as a list of URLs to crawl is generated it will likely be dominated by this domain. This is because while processing the list it will more often crawl pages from that domain. Processing the list of URLs to-crawl (known as the crawl queue) sequentially often means that over-crawling will occur, one domain at a time.

The Crawler's architecture will essentially determine where it deals with over-crawling a site; however the method used can be implemented wherever this is. When the decision is made as to which URL to crawl next, checks need to be made to ensure that this URL has not been crawled previously, or recently.

3.2.4: Downloading a Page

A crawler needs to download web pages (which are technically files), it is realistically the only way to find more links on the Web and continue its task. The basic procedure is outlined in chapter 2, using the HTTP request-response system, but there is more to be considered. The method used to download files would need specific features:

- It almost goes without saying that it should fully support the HTTP 1.1 protocol. Since its introduction in 1996 it was the majority protocol in use within 6 months and it is almost certainly standard today (Wikipedia, 2010).
- It must be capable of timing out. A crawler cannot afford to wait too long for a response; for example if a server is heavily overloaded it may start sending a response but not complete it, or perhaps not even respond at all. In this case the crawler should move on after a specified delay and ideally queue the URL to be tried again at a later time.
- For use in a crawler it is likely to be used in several concurrent threads within an application, therefore which ever method is used needs to be able to run simultaneously on several threads without any cross-thread interference.

3.2.5: Extracting Links

So that the crawler can explore further it must be able to find links on the pages it downloads and then crawl them. HTML pages are usually represented as long strings of text and they contain hierarchical mark-up tags which can have attributes, values and inner content. The hierarchy allows the page and elements in it to be represented as a hierarchical collection of data. These representations allow for various methods of finding links on pages. A suitably fast and flexible method must be used to ensure the maximum number of valid links can be discovered.

3.2.6: Check for Duplicates

At some stage the Crawler needs to determine whether a URL has already been crawled, either as they are discovered or as they are taken from the crawl queue. This is partly associated with the politeness of the crawler as it does prevent over-crawling a page or domain, but it also boosts efficiency. It is not feasible to crawl duplicates as they come up, as pages with many links to them may clog up the crawl queue meaning other pages never get crawled.

3.2.7: Queue for Indexing

Depending on how indexing is handled, more specifically if it is done separately from the Crawler; then the pages discovered and downloaded need to be queued and stored to be indexed. Systems like Google use the crawler to download the page and then store it to be indexed (Brin and Page, 1998). Separate indexer processes work though the store counting terms to create inverted and forward indexes as well as discovering links which are sent to the URL queue that feeds the Crawlers. Intermediate page storage may only be needed if the Indexer and Crawler work at different rates, which is probably quite likely as synchronizing them would mean one is just wasting time sleeping. The Crawler may be forced to wait if the Indexer is slower and the intermediate storage becomes full.

3.2.8: Deciding when to stop

Though it is theoretically possible for a crawler to continue crawling until every linked page it can discover has been visited; due to limitations such as resources or time it could be more practical to impose some sort of limit. There are various ways such a limit could be imposed. If any limit is applied there may be a need to analyse content during Indexing and prioritize certain URLs if it is thought they may be more relevant or useful.

3.3: Indexing RDFa

Indexing in a Spider is completely task specific; in this case the indexer needs to focus on parsing RDFa. As discussed previously and later, the ideal way to store RDFa is to convert it to RDF so this is what the indexer will need to do.

3.4: Storing RDF and RDFa content – Triple Store

The RDF extracted from web pages that have been crawled needs to be stored; it would be ideal to store it in a manner that allows it to be easily searchable. A database system would easily allow vast amounts of text to be stored and indexed so that it easily searchable. *(Note: database indexes and indexing is not the same as Spider indexing)*. However a system that can be based on database engines has been developed called a Triple Store.

Triple Stores are aimed specifically at storing RDF triples; they also index the triples so that the triples are searchable using a specific query language called SPARQL. There are a myriad of Triple Stores available in a range of languages and they all scale differently, some to millions of triples, and others to billions. Unfortunately the more powerful Triple Stores require vast resources such as 16 GB of memory per storage node. A suitable Triple Store will need to be chosen for storing RDFa.

3.4.1: Data Input, Output and Queries

The means of adding and extracting data is very much dependent on the method of storage used, however the most suitable methods (being Triple Stores and Database Engines) often offer similar mechanisms. A web interface is a common way to control data; and most Triple Stores and database engines have one. Data can be uploaded though the HTTP POST mechanism, and retrieved with HTTP GET requests (which is the standard request used to access a page). The data retrieved can be restricted using specific queries. Alternatively some Triple Stores and most database engines offer client libraries for most common languages which sometimes offer more control than a web interface, and more direct interaction with the server. The crawler will somehow need to interact with the Triple Store to insert data. The ability to query the Triple Store will be a non-essential and low priority requirement.

3.5: Scalability

Crawlers can be difficult to create and run as there are tasks they must perform which rely on external systems which are entirely out of the control of those running the crawler. There are also areas of the system which could prove to be very resource intensive; these factors must be accounted for and coped with as much as possible.

3.5.1: Hardware

Crawling and Indexing can be very resource intensive; lists needs to be managed, reshuffled regularly and lots of data needs to be stored. Using just one mediocre machine may prove to be very slow. Resources needed by a Spider are hard disk space, internet bandwidth, CPU time and memory. The cost of most of these resources scale exponentially with size, which means it might be more cost effective to distribute load among many average machines rather than few very powerful ones. Having many machines also improves redundancy because if one machine fails it proportionally makes up less of the cluster and unless it provides a key role this should have less of an impact.

3.5.2: Storing Data

A Spider will potentially acquire and generate a lot of data, some of which will be temporary such as queues and some will be permanent, like the data in the Triple Store. All of this data needs to be stored in such a way that is easily accessible, for example it all appears on the same logical device; it also needs to be in some way protected against failure. If a disk fails which contains a section of the data it may well render the whole set of data useless which would mean starting the crawl again!

The current limit for storing data in one physical device (such as a disk) is around 2 Tb which implies that for more than 2Tb to appear as a logical device (i.e. a mount point in Linux), a mechanism to span the data across several devices would definitely need to be used.

3.5.3: SPEED

As part of the crawler, pages need to be downloaded from external servers and this process may suffer delays due to many factors. The first step is the DNS lookup. This sets a chain of requests going between the hierarchy of DNS servers, and some of the servers may be too busy or unavailable to answer a request, meaning it could be delayed for some time or eventually timeout altogether.

If the DNS lookup is successful the crawler can then proceed to connect to the server and download the page. The connection process is also susceptible to delays which can be caused by network issues or server load, or just by the server not being available at all. Once a connection is made the server can also delay the response if it is overwhelmed.

Regardless of the cause of any delays, they must still be tolerated in the hope that the page is successfully retrieved. So that this does not become a bottleneck one solution would be to make several requests simultaneously; it means that while potentially waiting for some page requests others will finish and new ones will start, so the crawler is always busy doing something.

The ideal situation would be where the crawler is going so fast that its limiting factors are environmental conditions that are unchangeable. For example, if the crawler is downloading pages so fast that the bandwidth limit is reached, or it runs out of disk space when absolutely no more is available.

3.6: EVALUATION

The final system and its components will need to be evaluated to determine whether it has become a suitable solution for the project. It will be compared against the original requirements set out here, where an analysis will be made comparing the similarities and differences, why they exist and how they have or have not benefitted the project. A site or sites with a known number of triples will also be crawled to ascertain whether the Crawler is capable of finding RDFa data and this will show the crawlers accuracy. Other scalability and performance tests on components of the system and the system as a whole will be carried out during the development and evaluation.

CHAPTER 4: DESIGN

4.1: OVERVIEW

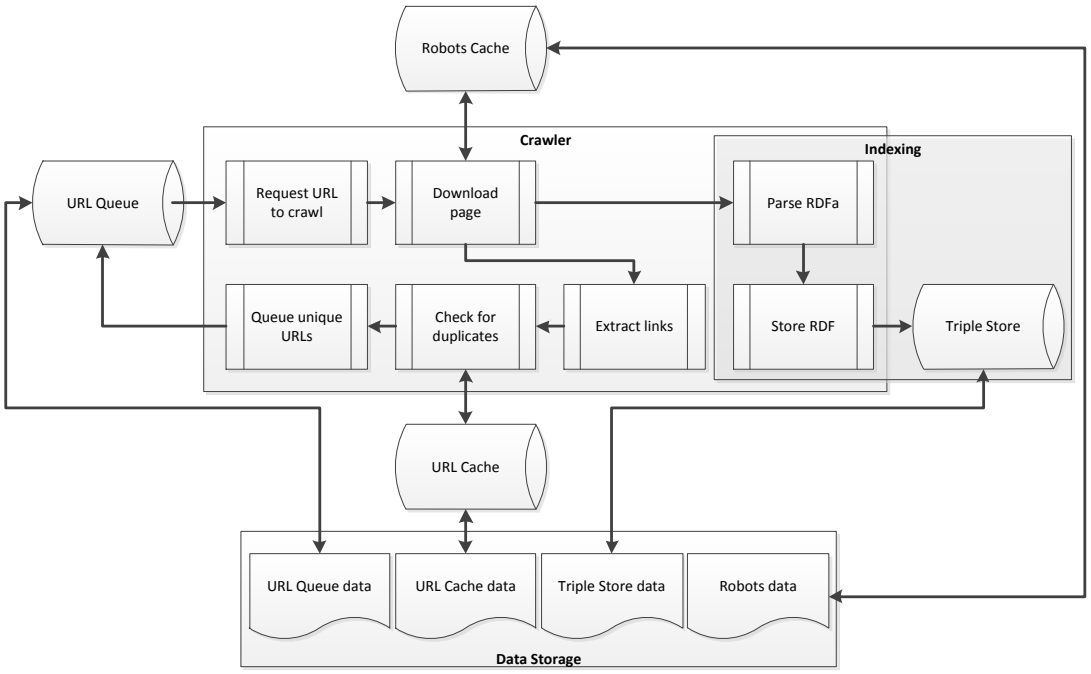


FIGURE 2: AN OVERVIEW OF THE SPIDERS ARCHITECHTURE AND INTERACTIONS

Figure 2 shows an overall structure for the Spider as a whole. The URL Queue, Robots Cache, URL Cache and the Triple Store are all separate applications (or services). All applications will run as single instances except the crawler itself which will run on several servers. It is likely a few other service applications will be created for tracking running applications and for logging purposes.

4.1.1: UNITY: CRAWLING AND INDEXING TOGETHER

In a system which both crawls and indexes data, there are two main ways to approach the need for both processes. The first approach is to use a crawler to take URLs from a queue, download the pages and then send them to a store. A separate process, the indexer, retrieves them from the store. Once the pages are retrieved they are parsed to extract links and other desired data; the links are then sent to the queue and any other data is stored as appropriate for the specific system.

Alternatively indexing could be done as part of the crawling process. If each document needs to be stored with the indexed data, for example for referencing, then the first method is likely to be best. However, if only the indexed data needs to be stored then indexing after crawling in the same process can minimize network bandwidth and potentially improve performance. As the data of a page does not need to be sent to another service, server or process it can be processed on the same machine in the same process simply passing the memory reference to the parser. This also means no intermediate storage needs to be used. As it is not necessary to index separately for this project it will be done as part of the crawling process.

4.2: LANGUAGE

There are many programming languages around today all of which have pros, cons and specific abilities. The choice of language could prove to be a key factor but on the other hand it may well cause restrictions too. The available languages could be restricted by the systems the code is going to be run on, for example Microsoft's .NET languages do not run natively on Linux (some, mainly C# can be run now under Mono (Mono Project, 2010), however some features of the .NET framework are not implemented). Languages like Java and those of the .NET framework have vast component libraries and memory management which can mean when used, programming tasks are faster to complete and easier to debug, however they can suffer in raw performance because of the safety checks and garbage collection that goes on while the program is running as well as lacking explicit use of pointers.

On the other hand, languages like C and C++ can be harder to use, and require much custom code or 3rd party libraries to perform 'simple' tasks that can otherwise be done with the built in libraries included in Java and .NET. But beneficially they also allow complete control over the allocation of resources. Like other decisions to be made, there is no *correct* choice out of the many potential solutions, each language will have benefits and cause issues.

The language used to implement the bulk of the system will be C++. The motivation for this choice is primarily due to two implicit requirements; the first is that the system will be running on Linux, mainly to avoid licensing costs, which essentially discounts any Microsoft languages such as ones using the .NET Framework like C#. However, Linux file systems generally do not suffer from fragmentation on the same level that Windows ones do which is a performance benefit. Secondly, there is substantial emphasis on performance, with the web being so vast it is beneficial for the crawler to run as quickly as possible. There are many data structures which along with being carefully designed, need to be accessed quickly, C++ allows much tighter and yet more flexible control over low level resources such as memory which may aid performance.

4.3: DATA STORAGE

There is potentially the need to store a vast amount of data, possibly more than can fit on one physical disk. There are four common ways of storing data across the boundaries of disks and partitions; in this section they are given a critical review.

4.3.1: RAID

A "Redundant Array of Inexpensive (or Independent) Disks" (RAID) is a very common storage solution, as the name suggests several cheap disks can be used to create a larger array which appears to the host system as one drive. Contrary to the name's suggestion, RAID arrays are not always redundant. There are several levels of RAID often identified using a numbering system. The most common RAID levels are detailed in Table 1.

Raid Level	Minimum Disks	Space Available (of *n* disks)	Description
0	2	n	Data is 'striped' across the disks in the array. Access is fast as sequential stripes are stored on separate disks. There is no parity or mirroring and hence no redundancy
1	2	n / 2	Data is mirrored between pairs of disks, for example with 4 disks (1-4) 1 may be mirrored to 2 and 3 mirrored to 4. One of each pair can fail without affecting the array.
5	3	n – 1	Data is striped with distributed parity; this means that if one drive fails access can still be maintained as reads can be calculated from the distributed parity. With one failure the data is at risk as a second failure will render the array useless.
6	4	n – 2	As with RAID 5 the data is striped but with dual distributed parity. Up to two drives can fail while still allowing access. This means that if one drive fails the array is still redundant which is useful as larger drives take longer to re-sync with existing data, so while the failed drive is being replaced and re-synced the original data is still protected.

TABLE 1: RAID LEVEL COMPARSIONS

Traditionally RAID systems run locally, that is on one machine and a limiting factor is often the number of drives you can connect to a machine or RAID Controller. It also means that the machine itself could be a bottleneck to accessing the data. The SATA II Bus has a data rate of 3 Gbit/s but the PCI bus has, at best a throughput of about 4 Gbit/s (on 64 bit systems) so if the drives are connected to a SATA-PCI host adapter then the full throughput of all four drives cannot be utilized simultaneously. RAID, when striping, generally uses all drives simultaneously, so the concept can be fundamentally flawed in terms of performance.

The redundancy benefits of RAID are often oversold. It should not be overlooked that RAID operates at a low level; the whole array appears to the operating system as a disk which means the raw data is protected. The downside is that if the system performs an operation that corrupts a key area of the file system, for example the File Table these data changes are sent to the storage device (RAID) and then copied to the disks (possibly with parities being calculated). In this situation the data has become corrupted but the RAID system functioned perfectly. RAID should not be used to replace conventional backups; it provides reasonable real-time protection against failures but not system or file system errors.

4.3.2: Basic Network Attached Storage

To take RAID access a step further is to use it as Network Attached Storage (NAS) whereby the array of disks is accessible as a mounted shared device. This may seem practical in the sense that many machines can access it simultaneously, however most NAS devices are just computers running basic Linux with hard disks and a network card and they are often no better than running RAID from a server together with a file server. A 'normal' server would actually give an administrator more control and flexibility.

4.3.3: High Performance Network Attached Storage

There is a hybrid model above RAID and basic NAS (+RAID) which is commonly used at an enterprise level where high performance and/or high availability is needed; it is also generally known as NAS. It combines the benefits of RAID while attempting to remove performance bottlenecks of certain host technologies (such as PCI) and in turn, as a necessary implication improve network access performance. A dedicated hardware system (disk array host) provides RAID services with an array of hard drives directly connected to it; the hardware is designed to operate each disk at full speed. This device can only be connected to by one machine, but at a very fast rate, often by fibre-channel or Infiniband which can offer speeds of up to 96 Gbit/sec (more commonly 8 Gbit/sec), possibly well above the speed at which the RAID array can operate.

So that many clients can access high speed disk arrays they are often connected to a server running a file server. Network speeds at best are currently about 10Gbit/sec but more commonly 1 Gbit/sec, which in high performance environments would appear to be a bottleneck. The solution is to have several high speed network cards in a machine working together to serve clients on the network; this is known as link aggregation. It does require the server to have very fast powerful processors, fast RAM and a fast BUS. Although this system can perform very well, all the requirements to achieve a high data throughput make this solution very expensive.

4.3.4: Distributed File Systems

Distributed File Systems allow many servers to store data but to appear to clients as one disk, folder or mount point. Unlike NAS RAID systems, access to the data is not reliant on the reliability of one single server or disk array; furthermore they allow for more cost effective scalability. As high performance NAS systems require disk array hosts and a server, once a disk array host is filled to add just one more disk would require another disk array host, and disk array hosts *can* cost as much as a decent server. Distributed File Systems can store data on many or few storage nodes (servers with hard drives); each node can have any number of disks connected to it which means to increase capacity either a single disk could be added to an existing node, or to a new node.

In a way, Distributed File Systems can offer a form of RAID 0 and RAID 1 where data stripes (often called chunks in Distributed File Systems) are stored on different disks (and servers) and these stripes are also mirrored on to other servers. The striping allows for high performance; so that many clients can access the same chunk simultaneously on different nodes which reduces the chances of overloading a specific node. This effectively acts like link aggregation.

Some Distributed File Systems employ load balancing and fault tolerance. If a file or chunk is being accessed heavily it will be replicated on to more nodes to spread out the load. To deal with fault tolerance chunks or files often have a replication count or goal, and if a node goes down the files or chunks are replicated to other servers to maintain the goal. Network switch failures can be compensated for by using Link Aggregation though different switches.

Google's GFS is an example of a distributed file system where the data is split into chunks and distributed many times over many servers offering high availability and fault tolerance. Like

several other Distributed File Systems, the data storage nodes can consist of commodity hardware. GFS does however require a master server to manage access, file locks and hold metadata such as the file system hierarchy and map of chunks on the storage nodes. This master server usually requires slightly better hardware – often more RAM so that the metadata does not need to be stored on disk to improve performance.

4.3.5: CONCLUSION

With RAID and NAS data storage solutions often being expensive a distributed file system will be used to store data on the assumption that the capacity of the disk(s) in one server will not be enough to store information gathered in a large crawl. There are many distributed file system solutions available for at no extra cost for Linux. During the Implementation the most promising of these will be tested for suitability and one will be chosen and configured on the cluster.

4.4: SCALABILITY

So that the Spider can run as quickly as possible it will need to be scalable so that it can be run on many servers simultaneously. This requires separate applications that can communicate with each other as well as common practices like multithreading and asynchronous input and output.

4.4.1: THREADING, SYNCHRONICITY AND ASYNCHRONOUS IO

There are often periods of waiting which a crawler must endure, but so that this doesn't waste time it should also be crawling other pages simultaneously. There are three ways to do this; the first and crudest is to run many processes each crawling as a single thread. This may consume resources unnecessarily as many processes could duplicate identical structures in memory. A way to improve resource efficiency is to use one process and either multithreading or use asynchronous IO. With threading, in theory each thread can request pages from servers; if one thread is forced to wait the others can still continue without being delayed. Some synchronization may be required to ensure all resources used are thread-safe.

With asynchronous IO, waiting for IO results will not block code execution, so several requests can be made and slow requests just continue to run in the background. However, unlike threading, if several requests come back at once they will then form a queue and be processed synchronously. A hybrid method using both ASIO and threading would be most beneficial.

4.4.2: COMMUNICATION BETWEEN APPLICATIONS

The Crawler will need to talk to the other services which are often going to be running on a separate server to the Crawler process itself. The only realistic way to do this is over a network. There are two main protocols used for communicating on an IP network: UDP and TCP.

The UDP (or User Datagram Protocol) uses stateless queries to provide a mechanism for answering large numbers of small queries from clients (one of the most common uses of UDP on the Internet is in DNS services). UDP is considered unreliable as the protocol itself does

not support message acknowledgement and therefore it cannot be easily known if the message has reached the recipient.

The TCP (or Transmission Control Protocol) is almost the complete opposite to UDP. It uses state based connections and a handshaking procedure to open a connection. Packets that are sent, are verified, check summed and ordered by the recipient so that any missed packets can be re-sent. TCP is considered reliable and the protocol's requirements ensure that the recipient has accepted the connection and is able to receive data, as well as ensuring messages have been received successfully.

Communication will be done using the TCP/IP network protocol as its protocol definition and state based connections provide a reliable communication system. This will require two main libraries to be created so that they can be integrated easily in to several services, a TCP Server and a TCP Client. The interactions between server and client will be simple request/response transactions, where every request expects a response.

Using a request/response model makes the creation of the TCP Server and Client easier, however complexity is not a significant factor as most interactions require a response, which will be described later. The TCP Client will have a method called *write* which will take a request string, send it to the server, wait for a response then return that as a string. The constructor will take connection parameters such as a host and a port, and it will automatically open the connection. The destructor will close the connection and clean up.

The TCP Server will listen on a specified port for TCP connections. When a connection request comes in it will be accepted and an asynchronous loop will begin. The server will read asynchronously until a (Windows) new line, represented as "\r\n" or Character Return Line Feed (CRLF) is received. As the read is asynchronous it should not block any new or existing connections. When a new line is received any text preceding it is considered to be the request string. That request is processed on a new thread so that any other asynchronous reads that come through are not blocked by the current one being processed. As with asynchronous IO the process of waiting is non-blocking but otherwise it still only runs on one thread. An asynchronous write is executed after the processing, sending back the result, and the call back from the write starts an asynchronous read, completing the loop process. The TCP Server will take a pointer to a Processor class so the processing method can be customized to suit the implementation.

4.5: URL CACHE

URLs must not be crawled too often; the Spider must keep track of which URLs have been crawled so that they are not re-crawled too frequently. For example, they could have a time associated with them for when they can next be re-crawled.

A simple solution would be to store a list of all URLs that have been crawled and for which the re-crawl delay has elapsed. When the URL is first crawled it can be added to this list and once the delay has elapsed it can be removed during a periodic purging process. Every time the crawler encounters a URL it can check it against this list; if the URL is not in the list it is crawled and added to the list, otherwise the URL is ignored and the crawler moves on to the

next one. Ideally the URLs would be normalized so that different variations of the same one are not crawled again.

Maintaining large lists of URLs (or any data for that matter) can be resource intensive, especially if duplicates need to be looked up based on the URL name and old entries need to be purged based on their date/time stamp. Alternatively, re-ordering lists as data comes and goes can also be very time consuming. If a list is stored in memory, operations are often very rapid as jumping between different areas in the memory (seeking) is very fast, however the limitation is that the list size is restricted to the amount of available memory. In memory data structures are lost as soon as the program terminates, or if unforeseen circumstances cause the system they are running on to crash, power off or otherwise loose the contents of the memory. Storing the lists on a conventionally more cost effective device, like a hard drive means the list can grow much larger but disk seek time is much slower than memory seek time making an on-disk list much slower. As keeping track of duplicates is likely to be very important, storing them in memory may be risky.

A more efficient system regularly used in databases is to use an index to quickly locate a section of a list where an entry exists or would exist. The index and data list will still change rapidly and both still need to be ordered in some way, so there are still performance related overheads but the ability to locate entries quickly is useful, especially when using such a method to identify duplicates.

Indexes usually partially duplicate data that already exists in the main data list, so it is not particularly space efficient; however there is a trade-off between space efficiency and performance efficiency. For rapid look-ups an index could be used without a list, storing the full entry in a rapidly searchable structure. Common indexes used a B-Trees and B+-Trees however they may not be best suited to storing whole values. In many cases their implementations are aimed at partial values, such as the prefix of a data list entry.

A basic option would be to implement an index using a MySQL database as a backend. With the different data types available for columns, and the ability to add indexes, look-ups should be suitably fast and all data changes, such as additions and removals are managed automatically by the database engine. With a quick implementation of tracking lists work on the crawler, which will rely on these, can begin.

A possibly more powerful option, instead of a database engine is to use a purpose built index. The system which keeps track of duplicate URLs will need to be capable of checking rapidly if it already knows of a URL or not. The data structure used to store these will be a Trie. Figure 3 shows a Trie.

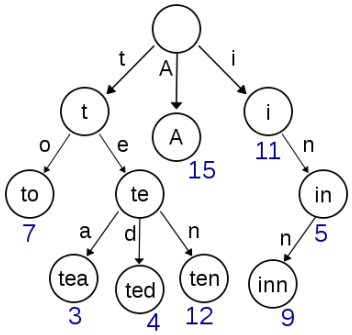


FIGURE 3: A VISUAL REPRESENTATION OF A TRIE FROM WIKIPEDIA, PUBLIC DOMAIN AND CAN BE USED FOR ANY PURPOSE

In Figure 3 the arrows contain key characters and the nodes represent the implied key of the path. The numbers by the nodes represent values associated with the key name implied by the path. Though there are several data structures often used for indexes, Tries have several benefits for this application.

Duplicate prefixes do not waste space in Tries. To ensure this is a benefit, URLs will be stored in a specific format. This is necessary as they are made of two *main* parts, the domain and the path; roots of both meet in the middle.

Domain roots are at the end of the string, a domain technically should end with a dot – where most people would write www.rdfas.com it would usually be automatically (internally) corrected to www.rdfas.com. with the extra dot at the end. The dot is effectively the root node of a domain which means domains have a root-last hierarchy; com is one of the first level nodes (otherwise known as a top-level domain). Figure 4 shows a tree of domains would be represented.

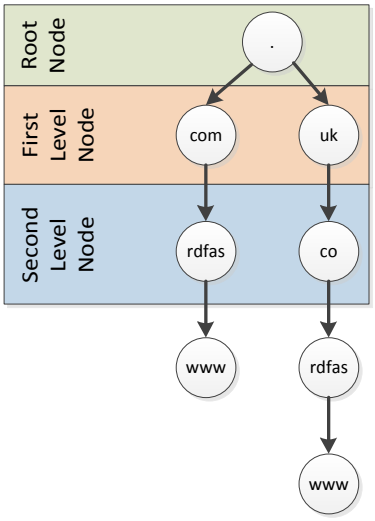


FIGURE 4: A TREE SHOWING SEVERAL DOMAINS AS A HIERARCHY

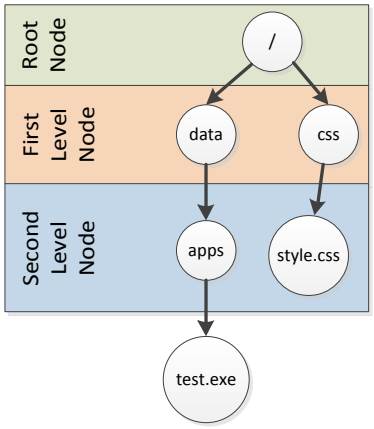


FIGURE 5: A TREE SHOWING SEVERAL FOLDERS AS A HIERARCHY

As previously stated, the *dot* is the root, *com* and *uk* are first level nodes (top level domains), *rdfas* and *co* are second level nodes (co is a second level domain, though *rdfas* is not as it is not a common suffix), and so on with both *www*s being leaf nodes.

Paths start with the root. In the case of domains, the root is the slash, which is why as part of URL normalization www.rdfas.com would usually be corrected to www.rdfas.com/ (or www.rdfas.com./ though the dot is usually an automatic internal correction as part of the DNS lookup and not URL normalization). Figure 5 shows a folder hierarchy is.

Domains have many common prefixes as the list of top (and second) level domains is very much restricted, so for example, it can be guaranteed that there will be many domains ending in *com* but none (currently valid) that end in *test*. Using this fact and the knowledge that Tries eliminate duplicate prefixes, the domain string will be reversed so that the *dot* root is at the beginning, making the whole path a root-first hierarchy. www.rdfas.com/test/page.html will become .moc.safdr.www/test/page.html which means the root is at the beginning and the hierarchy progresses from left to right. Note the http prefix is dropped as the Spider will only support http resources so it is unnecessary to store it. As a Trie indicates a character at each level and not a chunk of a string the whole domain is reversed, not just the structure meaning that domains sharing a common suffix will end up sharing a common prefix, which will then not be duplicated. An example is shown in Figure 6.

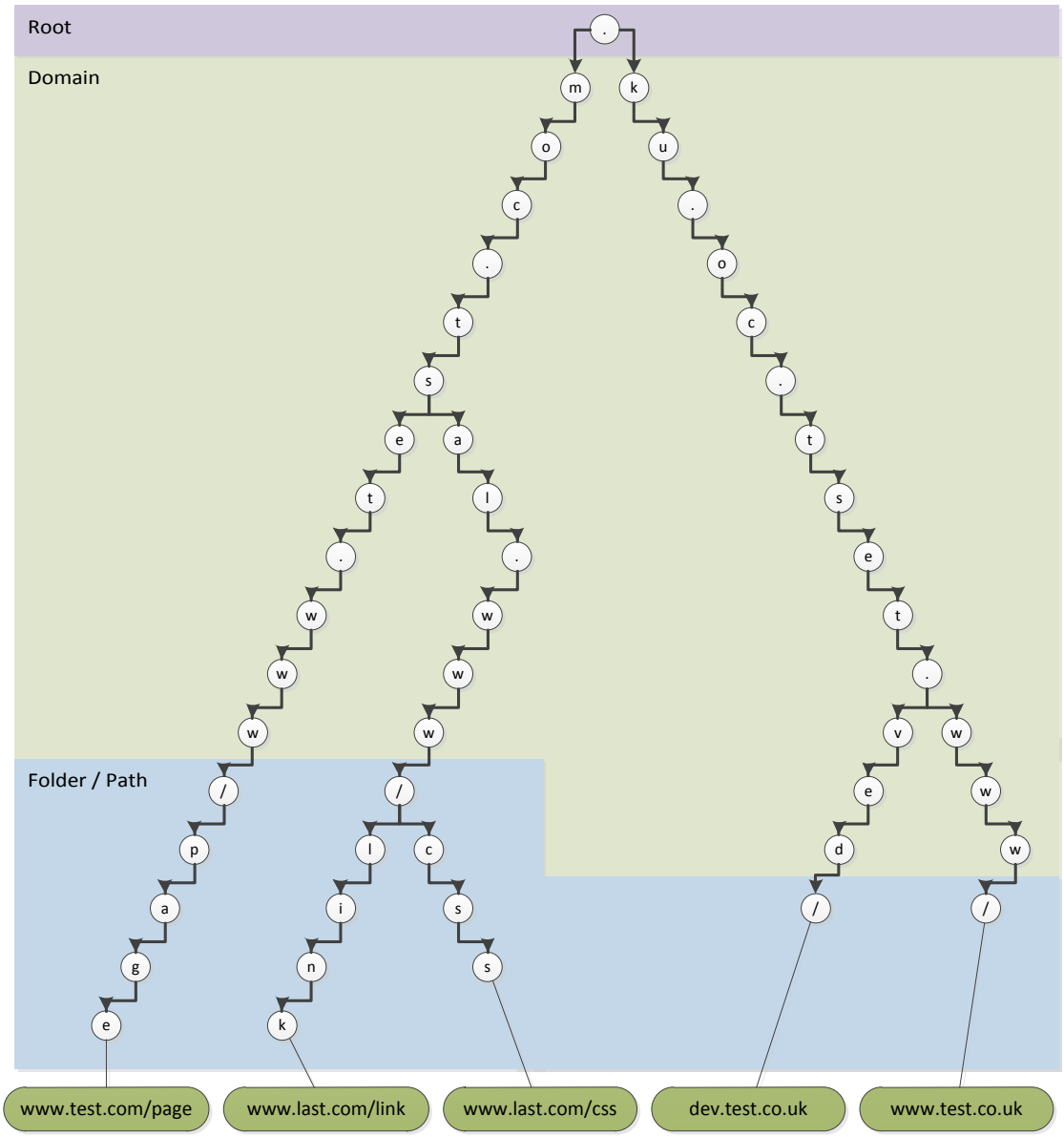


FIGURE 6: SEVERAL URLS SHOWN AS A FULL HIERARCHY WITH REVERSED DOMAINS

Clearly the tree structure grows quickly, and even short domains take up lots of space when visualized. Each node represents an element in the Trie. It has a parent and it potentially has peers and children. In this implementation there is no need to recurse up the tree, so parent elements do not need to be stored. Each node will have an ID, starting from 1; it will point to its *first child* and its *next peer*. A pointer to 0 will represent a null node, in other words no node.

Nodes at each level will not be sorted; there are several reasons for this. For sorting to be in any way effective the number of nodes at the level would need to be stored. However even knowing the number of nodes on each level does not indicate the distribution if they are sorted, for example if there are 10 nodes, they could be *a to j* continuously, or *q to z*, or some other set and if looking for *m* for example, jumping halfway through (as m is halfway through the alphabet) then searching left or right based on the value at that point will take just as long in both cases. Scanning in both directions will require each node to know its *next peer*

and *last peer* and the ability to jump to a point will mean the parent node will have to know all the *ID*s of its children. This poses several issues which are discussed below. If there is no list of children then the nodes will have to be scanned to find the middle, following the *next* or *last peer* pointers, which is no faster than just searching for that node in an unsorted list.

Node entries need to be stored in blocks of a fixed size so that there is no need to expand or shrink the block. This is because making extra space contiguously or making use of free space can require data to be shifted which is extremely costly in hard drive operations. Storing a list of child nodes, as mentioned in the previous paragraph would either require space to be reserved for the list, or for the node's entry block to be expanded. There are two ways to expand the block, either all data after the block would need to be shifted down to make space, or the data would be added at the end of the file and it would need a pointer from the original part to the part at the end. This fragmentation of data will severely degrade performance as many users of a Microsoft file system will be happy to complain about.

In summary a Node needs to store the character itself, its *next peer*, its *first child* and whether or not it is *terminal*. Terminality indicates that from the root to the terminal node it is a full URL that has been seen before. As the blocks are of fixed size and are appended to the data stream their ID can be inferred from the position and intuitively the position can be calculated from the ID. Each pointer will be 64 bits giving a limit of 18,446,744,073,709,551,616 entries which is likely to be far more than needed or even feasibly storable. 64 bits is 8 bytes, a terminality Boolean is 1 and a character is 1 making a node block 18 bytes. Compressed pointers will not be used as they will require data shifting which will cause the same issues discussed in the previous paragraph.

Each Trie entry block will be structured as follows:

```
nnnnnnnncccccccvt
```

Where *nnnnnnnn* is the *next peer* ID, *cccccccc* is the *first child* ID, *v* is the value (character) and *t* is the terminal indicator. The position of the block itself implies its own ID, as they start at 1 a block at offset 0 would have ID 1, a block at 18 the ID would be 2, and so on.

For the following example short, unrealistic (or invalid) domains will be used so that the visualizations are smaller and more readable. As all domains end with a dot it means it is a root that will be common to *all* Trie entries after they have been rearranged. The URLs http://abc/1, http://edc/2, http://edc/3 and http://xyz/4 would be represented in a shown in Figure 7.

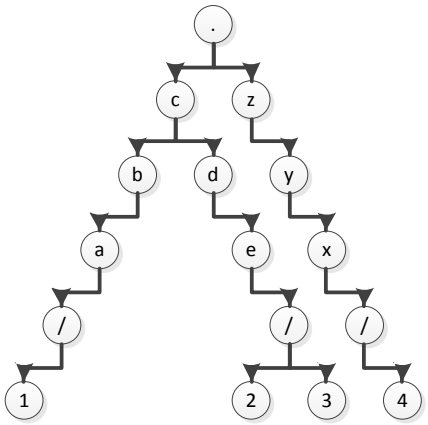


FIGURE 7: FOUR URLS SHOWN AS STORED IN A TRIE

The data structure, comprised of node entry blocks could be visualized in progressive steps as shown in Figure 8; however the order and arrangement of nodes may vary based on the order added.

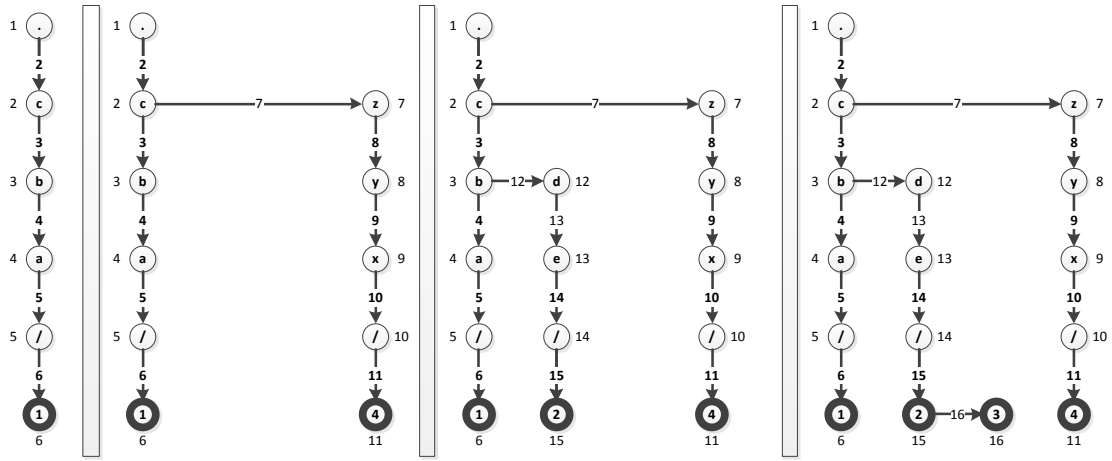


FIGURE 8: PROGRESSIVE STEPS SHOWING HOW FOUR URLS WOULD BE STORED IN A TRIE DATA STRUCTURE

Each circle represents a node entry, the value in the circle is the node value (character), the numbers adjacent to the nodes are the inferred IDs and the numbers on the arrows are the pointer values. A downwards arrow represents a *first child* pointer, and an arrow across represents a *next peer* pointer. All nodes have both pointers but they will simply point to 0 if there is no child or peer and so they are not shown in the diagram. The bold circles represent terminal nodes.

4.6: URL QUEUE

A system is needed to tell the crawler processes which URL to crawl next. There are several ways to store such a list; a quick implementation would be to use a database system which would allow specific data types to be associated with columns. A performance increase may be offered by a more customized solution, tailored to utilize storage in memory, on disk or both

As outlined in the requirements, the URL queue affects the politeness of the crawler. If the queue is sending out too many URLs on the same server (or with the same domain) that server or site will get over-crawled. Essentially the solution is not to processes the list of URLs to-crawl sequentially, and analyse the URLs to ensure that the domains are also not crawled sequentially. The most effective method is to use a round-robin approach when considering the URL's domain.

For example a list containing:

- http://www.1.com/page1
- http://www.1.com/page2
- http://www.1.com/page3
- http://www.2.com/page1
- http://www.2.com/page2
- http://www.2.com/page3
- http://www.3.com/page1
- http://www.3.com/page2
- http://www.3.com/page3

Would be best parsed in the order:

- http://www.1.com/page1
- http://www.2.com/page1
- http://www.3.com/page1
- http://www.1.com/page2
- http://www.2.com/page2
- http://www.3.com/page2
- http://www.1.com/page3
- http://www.2.com/page3
- http://www.3.com/page3

This is so that all domains are crawled as infrequently as possible. Every time a URL is added to or removed from the list the order would need calculating to maintain the round-robin layout to prevent over-crawling. In practice this may not be possible as it could become very resource intensive as the list grows.

The standard Trie (discussed in the previous section) can be taken further and can be used as a round-robin based URL queue, this variation has been given the name XTrie (Extreme Trie) to differentiate it from the original. Each XTrie entry block will require two more data values to be stored, a 64 bit pointer called the *round robin* pointer and the *terminal* byte will now store more than just terminality. If a node is *terminal* it can also have the state '*to be done'*. Any node can have the state '*a child needs to be done'*. All this information is encoded in the terminality character as previously only 1 of 8 bits were used. The XTrie could now be used as both a URL Cache and a URL Queue.

Every time a URL is added, the last node again will be *terminal* to identify that it is a full URL but it will also be marked as *to-do* (or to be done). The recursive algorithm will work up the tree making sure all parent nodes have the state *inherited to-do* (or a child needs to be done). Once this is done a quick check at any level of the XTrie can tell whether or not a branch has *to-do* URLs on it. The *round-robin* pointer is similar to the *first child* pointer, except it does not point to the *first* child, just the *next* child to be used.

The following pseudo code describes how finding a URL *to be done* from the XTrie is carried out:

```
//Starting Point
CurrentNode = RootNode
NextNode = CurrentNode.RoundRobin

//Loop until a Terminal Node with To-Do is found
While NextNode Not (To-Do And Terminal)
      If (NextNode == Null)
            NextNode = CurrentNode.FirstChild
      Else
            NextNode = NextNode.NextPeer
      End If
      If (NextNode == CurrentNode.RoundRobin)
            Error "This situation should not occur!"
      End If

      //If Inherited To-Do Move down a level in the XTrie
      If NextNode Has Inherited To-Do
            CurrentNode.RoundRobin = NextNode.NextPeer
            CurrentNode = NextNode
            NextNode = CurrentNode.RoundRobin
      End If
End While
```

Upon leaving the While loop NextNode should be a *terminal* node which needs *to be done*. The URL can be found by logging the nodes values as the algorithm recurses down the XTrie. Each time it goes down a level in the XTrie the *round-robin* pointer is updated to the *next child* node for the next time it is used.

Once a node is found that is *terminal* and *to-do* then the *to-do* flag must be cleared which may require *inherited to-do* flags to be cleared too. If the node had *inherited to-do* and *to-do* it means it still has children which need doing, in this case no recursion is needed to change parent nodes as this also implies they have children that need doing. Otherwise, if de-flagging the node will potentially cause the parent to change, all peers have to be checked for explicit or inherited *to-do.* If the result of this check results in a change of state for the parent then the parent is the current node and the check starts all over again, potentially until the root node is reached. This ensures all *inherited to-do* flags are correct, and if they are the ERROR in the pseudo code should never be reached.

The same (unrealistic) example URLs described in the Trie would be built up in an XTrie in steps as in Figure 9.

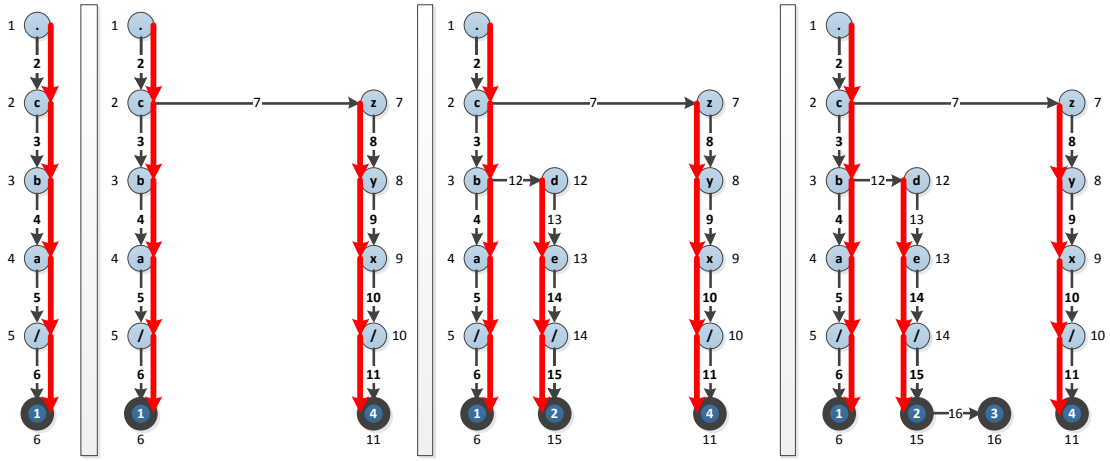


FIGURE 9: PROGRESSIVE STEPS SHOWING HOW FOUR URLS WOULD BE STORED IN AN XTRIE DATA STRUCTURE.

As before, each circle represents a node entry, the value in the circle is the node value (character), the numbers adjacent to the nodes are the inferred IDs and the numbers on the arrows are the pointer values. A downwards arrow represents a *first child* pointer, an arrow across represents a *next peer* pointer and a thick red arrow indicates the *round-robin* pointer. All nodes have all three pointers, they will simply point to 0 if there is no child or peer and so they are not shown in the diagram. The bold circles represent terminal nodes. Dark shaded nodes have a *to-do* flag and light shaded nodes have *inherited to-do*.

A URL requested would follow the path of red arrows in the previous diagram (green and dashed in Figure 10) from the root at ID 1 to ID 6. *Inherited to-do* and *to-do* markers would be removed and the *round-robin* pointers would increment to the next child where they are encountered and if another child exists. Light grey arrows are *round-robin* pointers that still exist but will never be followed, unless a new URL is added down that path (see Figure 10).

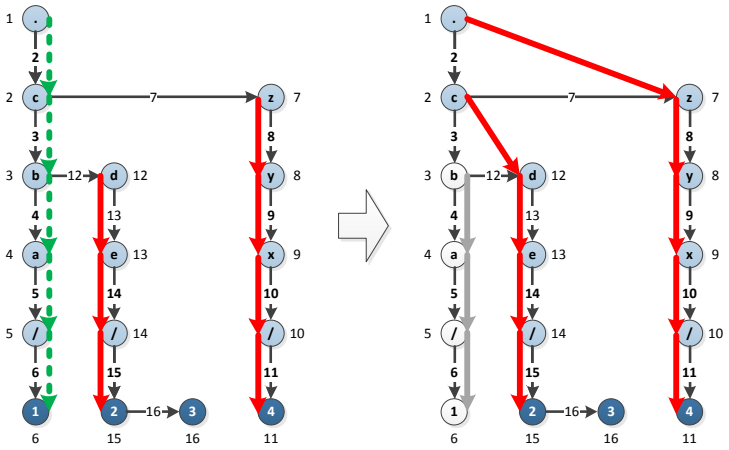


FIGURE 10: STEPS SHOWING HOW URLS ARE QUEUED AND EXTRACTED FROM AN XTRIE

Now for the next URL request a completely different path will be taken, ensuring no two same domains, or any part of the path where this is a branch, is returned in successive requests. The following two URL requests are illustrated in Figure 11 in the same format:

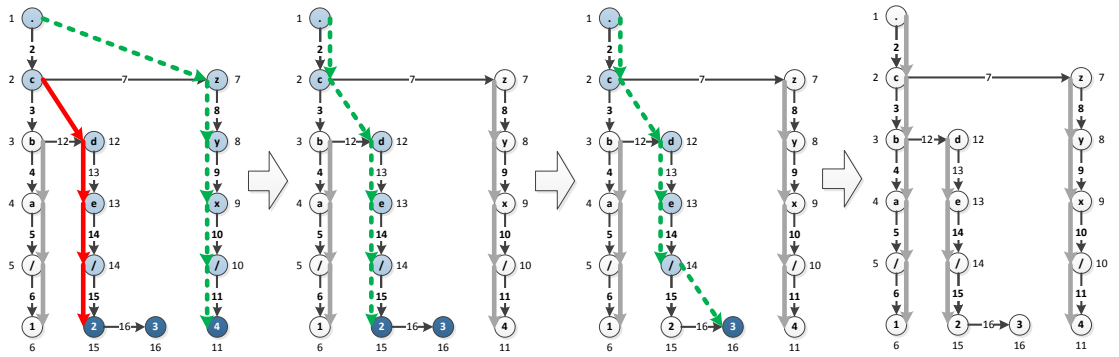


FIGURE 11: FURTHER STEPS SHOW HOW URLS ARE QUEUED AND EXTRACTED FROM AN XTRIE

At some points the *round-robin* pointers would have incremented then moved back to the start, these are not shown. Though this system may seem complicated, it should be very successful at ensuring that over-crawling does not exist when there are other domains available in the queue. This service will actually run as two TCP servers, one to accept incoming URLs and one to accept requests for a new URL to crawl.

4.7: Robots Cache

Staying out of restricted areas is a very big part of a crawler's politeness and was discussed in the requirements. It is relatively easy to deal with as there is a system in place for informing crawlers of where they are and are not allowed to go. The robots file, found at the root of the site. This can be parsed the first time the domain is accessed and a list of inaccessible URLs can be generated for that domain, then when the crawler is instructed to access a page on that domain it can check whether to proceed or not. To minimize bandwidth overhead the robots file could be cached for a certain period of time, but care would need to be taken to make sure it is not cached for too long.

There are three parts needed for the robots system, a parser for the robots file to determine which rules apply to the Spider and then an interpreter to decide whether a specific URL is blocked, and finally a cache so that the robots file does not need to be retrieved for every page, just once every so often.

The robots parser will take the raw robots data and the User Agent of the Spider. It will then calculate which rules apply to its User Agent, and represent them as a compressed format string. This string can then be stored in the cache and passed to the robots parser in the future, meaning that minimal data is cached and the robots file is only parsed once per fresh retrieval. Once the robots parser has the relevant data it will be able to calculate which URLs are allowed to be crawled and which are not.

The robots parser will handle the Disallow command, as specified by the robots exclusion standard, but also the Allow command. When there is ambiguity about which rule applies the most specific or longest match will be used.

The Robots Cache will run as a TCP Server. When the crawler needs to check a robots rule it will request the compressed robots string from the server and if needed, the server will fetch the file and process it to extract the string which will be cached and sent. However if a cached

version already exists it will send the cached version. When the Crawler receives the data it will use the parser again to check whether the URL it is about to crawl is allowed. The Robots Cache will store data in a MySQL database primarily, however if time permits a dedicated structure will supersede it.

Web masters often like to know why their site is being visited, to identify the crawler it will have a descriptive User-Agent which will state that is the RDFaS Crawler, with a link to a page describing the Spider and its politeness policy.

4.8: CRAWLER

The crawler will implement the design outlined in the overview section (4.1). The crawling stages in the overview will be called in a loop; however some stages (namely indexing) could block the process for a substantial amount of time. To allow each machine to crawl rapidly each crawler process will start many threads.

At some point the crawler might have to stop, either by choice (a stopping condition) or because of resource limitations (such as disk space). If the desire is to simply crawl as much as possible before resources are exhausted then the crawler stops when it runs out of the most limited resource, for example disk space. In this case it should not be too difficult to predict roughly how many pages will be included in the crawl and how long it might take. Another such method would be to stop after a specific number of pages have been crawled, or after a certain amount of time.

An indirect limit on pages crawled is to restrict the depth. The seed page has depth *n*. Every page found from the seed page has a depth of *n – 1.* Every page found from those pages have a depth of (*n – 1) – 1*, *n – 1* being the depth of the page they were discovered on, and so on. Once the depth of a page reaches zero it is still indexed, but any links found on it are ignored and are not added to the crawl queue. Depth limits will be used so that sites can be explored a specific distance from the seed site(s).

4.8.1: REQUEST URL TO CRAWL

The crawler's main loop starts by determining which URL it should crawl next. To do this it makes a request to the URL Queue over a TCP connection. The URL Queue then sends back a URL, if one exists and the depth of that URL, so that when subsequent URLs are discovered their new depth can be calculated.

4.8.2: DOWNLOAD PAGE

There are three areas of the Spider which require pages to be downloaded. The crawler itself needs to download pages, as does the robots cache sever (described in detail in section 4.7) and submitting to the Triple Store uses a HTTP POST method which submits the data as a request and downloads the response, this is discussed in section 4.9.

C++ does not have any built in libraries to handle downloading content from web servers, but it can manage sockets on a low level, or alternatively the 3rd party Boost library could be used for its ASIO (Asynchronous IO) classes. There is also a C implementation of a class to download files called Curl which has C++ wrappers.

Curl is a commonly used library for downloading files and generally making HTTP related requests, it is implemented in C but can easily be used in C++ directly or by using the wrapper library called Curl++ (curlpp). As specified by the requirements Curl supports timeouts, custom headers and POST, therefore it will be used to download pages.

4.8.3: EXTRACT LINKS

The crawler needs to extract links from pages it has downloaded so that it can discover other un-crawled pages. A simple way to extract links would be to parse the string, looking for start link (anchor) tags "<a" and matching end link tags "" and then to search the content in between for a link or "href=" followed by a link. Crude string parsing can often be slow, restrictive and difficult to modify; a more efficient method would be to use a regular expression.

Regular expressions are one of the most common techniques used for parsing text, amongst other things. A regular expression could be used to find all tags starting with "<a" that contain "href=" and extracting as a group the content within quotes after the "href=" section. It should be case insensitive and lenient about whitespace appearing in different sections, for example between < and A.

Web Browsers parse web pages and use their hierarchical structure to create a DOM or Document Object Model, where the HTML elements on the pages (mainly represented as HTML tags, e.g. "<a … /a>") are represented hierarchically as objects. Thus allowing the selection of all elements of a type, such as the "<a" tags that are used for links. This type of interpretation of a web page may be over the top as processing all the elements just to extract links could be unnecessary.

C++ does not contain any built-in methods for dealing with Regular Expressions or representing HTML pages as Document Object Models, but fortunately the invaluable Boost library has a set of classes for using Regular Expressions. To parse and extract links a Regular Expression will be compiled using:

```
<\s*A\s+[^>]*href\s*=\s*"([^"]*)"
```

This Regular Expression will find anchor tags which contain links, and extract the link portion as a group. It can easily be modified to make it more restrictive, for example to crawl only one domain. The matches will then be iterated over to create a list of URLs found on the page. So that the crawl does not grow too large only the first 100 links will be collected from any page.

4.8.4: CHECKING FOR DUPLICATES

To ensure that duplicate URLs are correctly identified they will first be normalized using a URL class. The URL class will take the raw URL string and break it up into protocol, domain and path; these sections will then be normalized according to RFC 3986. As with the Robots Server, the URL Cache will also run as a TCP Server, which stores URLs that have been Crawled in a Trie as described in section 4.6.1 so that all crawlers can access the cache.

4.8.5: QUEUE UNIQUE URLS

Discovered URLs that are unique will then need to be queued to be crawled. The URLs are sent to the URL Queue over a TCP connection along with their depth, which is 1 less than the depth of the URL they were discovered on.

4.9: INDEXING

Indexing will be done in the crawler to eliminate the need to store the pages, as they are not needed later for references or extra processing. Storing them temporarily will only take up unnecessary disk space and bandwidth.

4.9.1: TRIPLE STORE

Sesame can be accessed by two methods; a Java API allows direct access through the storage backend or remote access via the web interface, and a standard 'web client' can commit and retrieve data through the Apache TomCat interface. Non-Java remote access of Sesame is only available using a web client, which is again where the file downloading class will be used. Sesame through TomCat makes use of HTTP verbs; GET requests are queries, POST requests with *request data* are used to submit triples and DELETE requests can be used to remove data.

INPUT

The parser used by the crawler (pyRDFa) can output to several formats of RDF: RDF+XML, Turtle and N-Triples. Fortunately, Sesame can accept all these formats and more. To submit data a POST request is made to the Sesame server, in this request a querystring is used to specify the BaseURI and Context, both of which have to be URL Encoded. The request has a body which contains the RDF data, this body must either have its length specified in the request headers or be sent using chunked encoding. The request header must also specify content type so that Sesame knows the format of the incoming triples. If all goes well a 204 "No Content "response is returned which means the request was OK but the server has nothing to return.

QUERIES AND OUTPUT

As a low priority the ability to query the Sesame database may be made available through a web interface. Queries will be made using the SPARQL query language and sent to Sesame as a HTTP GET request. The return response will then be interpreted and the results displayed.

4.9.2: PARSE RDFA

As the requirements stated the indexer needs to parse RDFa. There are two main approaches to this requirement, either create a parser or use a third-party one. Creating a parser may be an exercise in futility as there are many fully featured third-party parsers available in most common languages (RDFa, 2010). Creating one would only be necessary if it is either not available in the language of choice, or if it would provide a feature that none of the currently available ones do. Notably RDFa Distiller (also known as pyRDFa), which is W3C's official implementation, is currently in public use as part of their semantic tools (W3C, 2010).

The RDFa parser and convertor provided by W3C is open source and freely available for developers to use in custom implementations. The public W3C version called RDFa Distiller is based on pyRDFa, the aforementioned open source parser made by Ivan Herman. pyRDFa is

mainly a library; however it contains a few sample implementations; one of which can take a file which contains RDFa and convert it to RDF. The crawler will use such a method to parse html pages.

Once a page has been downloaded by the crawler and it has completed extracting links it will be saved to a temporary file on disk. A Python script utilizing pyRDFa (which itself is written in Python) will then be called by the crawler and instructed to parse the temporary file. Executing the parser is a blocking call so that thread of the crawler waits for the parser to finish. The resultant RDF is returned through the output stream. The crawler pre-emptively tries to determine whether a page should be parsed by checking if the page contains RDF(a) related keywords or XML namespace declarations which often imply RDFa markup.

Invoking external applications is often risky but as pyRDFa seems to be one of the best, stable parsers then it is a situation that will have to be tolerated.

4.9.3: STORING RDF

Data will be stored using a Triple Store called Sesame. There are many Triple Stores available, however some have requirements that are not realistic for this project, for example 4Store recommends 64 bit hardware and 16 GB of RAM per storage node. Sesame has been recommended for this project by several individuals who are very experienced in the area of the semantic web and the technologies associated with it, so it will be the Triple Store used.

CHAPTER 5: IMPLEMENTATION AND TESTING

5.1: IMPLEMENTATION

5.1.1: OVERVIEW

The shared nature of many of the components developed for this project meant that a lot of code was developed as C++ header-only libraries which were then imported into the applications that required them. The overall structure provoked a standard library and application naming convention, where each class library project was prefixed with rdfas_lib and each application (server service) was prefixed with rdfas_app. The interactions and services are summarized in Figure 12:

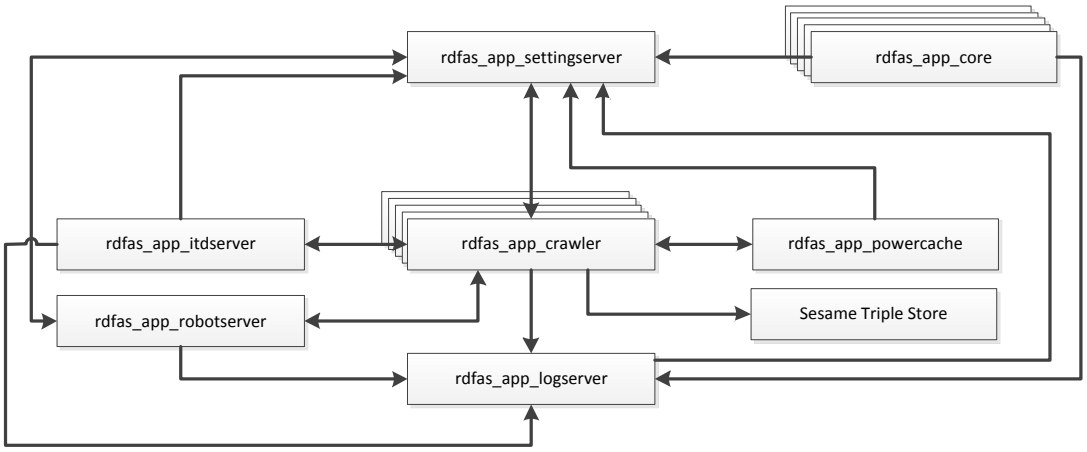


FIGURE 12: THE IMPLEMENTED ARCHITECTURE OF THE SPIDER

These applications and how they have been implemented will be described in detail in this chapter. The applications themselves do little but tie powerful libraries together, so the libraries are also discussed.

The design called for a several main components which have been created as applications:

- Robots Cache ➔ rdfas_app_robotserver
- URL Queue ➔ rdfas_app_itdserver
- Crawler ➔ rdfas_app_crawler
- URL Cache ➔ rdfas_app_powercache

Some of the other applications which do not link with the original design were created to make the system easier to control and monitor. The final implementation consisted of 23 C++ projects, some applications and some libraries which in total contained 50 classes.

5.1.2: LIBRARIES

COMMON

As the name suggests, the common library implements functions that are utilized by most of the other parts of the system. Classes in the library include support for processing strings, interacting with the file system, retrieving system information from Linux, URL representation

and normalization, Pointer conversion, managing Log and Settings files and a thread-safe integer.

CACHE

The Cache library implements the Trie structure described in section 4.6.1. Several steps were made to improve the performance of the Trie as the basic implementations method of storing data using a file stream only allows one request at a time. Locking was implemented to allow multithreaded access as a block cannot be read while it is being written by another thread; each block could be locked for read or write access. Multiple read locks can occur concurrently but a write lock prevents read locks until it is complete. Though in theory this would work efficiently, the overhead of the locking system slowed the system down more than just queuing requests and handling them one at a time.

As locking did not provide a suitable performance increase the next logical step was to speed up each Trie interaction. This was done with block caching; the last ten million blocks (approximately 180 Mb) are cached in memory. When they are read they are stored in a map which links their ID to the data, and when they are saved, the cache is updated as file stream data is updated. As updates apply to both the cache and the file stream during data access the cache only needs to check the file stream if the block is not cached because it can otherwise guarantee that the file stream entry has not changed since last use. Performance tests were carried out with block caching and different file system accesses methods, the results are in Chapter 6.

LOCKING

The locking library was created to provide a central method for locking resources. It was intended to be used by the Trie implementations to synchronize the Trie entry blocks under multithreaded access. The locking library has two methods of determining whether a resource is free, it either maintains a map of ID's to Mutexes so each resource can be directly locked, or there is a list of locks and a Mutex allowing access to the list. The locking system is not currently enabled in the Trie implementations as the overhead for Mutexes was more than the performance gain.

QUEUE

The XTrie design did not work out as planned, the heavy level of recursion sometimes required to update inheritance flags caused it to be very slow as it had to potentially look up hundreds of nodes to check their flag values. The system used to replace the XTrie was called the Paged Queue.

Instead of maintaining a potentially massive list of entries which are continuously sorted to avoid over crawling, the data is broken up into 'Pages', a concept abstracted from database and memory data management. There are two types of Page used, a Queue Page and a Temporary Page; each page has its own file. Files are used for two reasons, so that queues can grow large and so that the data is persistent if the application using it crashes.

A Queue Page takes all entries upon its construction and saves them to the associated file. Then when entries are requested, they are returned in order, and as they are a marker is written in the file by that entry to indicate that it has already been processed. Then if the

application crashes it can resume from where it left off. Alternatively Temporary Pages start empty and fill up as entries are added to them, once the page is deemed full all the entries can be dumped in one go as a vector of strings.

The library's main interface provides two methods - add and get; and as the name suggests the overall library behaves like a queue. As entries are added they are stored in a Temporary Page, when the Temporary Page becomes full (determined by a maximum size) the queue is emptied into memory, sorted and saved in a Queue Page. When an entry is requested it is taken from the current queue page.

Two Page Managers are used, one for the Queue Pages and one for Temporary Pages. They keep track of available free pages, used pages and the current page in use. Every time the current Queue Page is emptied the Page Manager selects the next Queue Page to be used as the current Queue Page and removes the empty one. A separate thread is used to process temporary pages so that it does not block the "add" or "get" processes. If a Temporary Page is filled the Page Manager creates a new one to be used while the old one is being processed. If lots of entries are being added rapidly then Temporary Pages are repeatedly created as they fill up and the full ones are processed one at a time on the other thread.

Temporary Pages are processed and saved as Queue Pages in an attempt to avoid over-crawling of a domain. Each entry is known to be a URL and so the first step extracts all the domains and counts them. A value called the diversity threshold determines how common the most common domain can be, for example a diversity threshold of 0.5 means that at most the most common domain can make up 50% of the whole set. Any entries with domains that violate the diversity threshold are removed and added to the current Temporary Page in the hope that they can be filtered out with new entries. The diversity threshold is automatically tuned, this is done by looking at how many requests were made since the last Temporary Page was processed and divided by the time since the last processing to give an estimated number of "gets" per second. The diversity is set to the reciprocal of this number so that in theory, if the crawl rate is steady no domain will be crawled more than once a second.

The process of sorting spreads the domains out in the list as much as possible. For the sake of example, imagine the domains are the numbers 1 to 5, and as paths and other elements of the URL make no difference, they will be ignored. If there are 20 domains:

```
1 1 1 1 1 1 1 2 2 2 2 2 3 3 3 3 4 4 4 5
```

The domains are processed in the order of their frequency, with the most frequent being first. In this example this is domain *1* with the frequency of 7. The total, 20 is then divided by 7 which is 2.85. Starting from the index 0 the domain 1 is added at every 3.3 positions in the new queue (which has the same number of positions as the diversity pruned temporary queue):

```
1 _ _ 1 _ _ 1 _ _ 1 _ 1 _ _ 1 _ _ 1 _ _
```

The same process is done for the next most frequent domain, starting from the *first available*

index. If an entry already exists at that index the index is incremented or decremented depending on where the remaining space is. The final result is:

```
1 2 3 1 4 2 1 3 5 1 2 1 3 4 1 2 4 1 3 2
```

DOWNLOAD FILE

The design of the Download File library called for the use of a third-party library called Curl (and possibly its C++ wrapper Curl++). It was required that the Download File library would need to be able to run on several threads simultaneously so that it could be effectively implemented into the Crawler. Curl uses a call back to process incoming data, and due to its basis in C this call back and a few would-be class level variables it uses needed to be static in C++. If two threads finish downloads at the same time, the call back called simultaneously from the two threads cannot share access to the static variables and it often causes a Segmentation Fault. A Segmentation Fault is an error that occurs when a program attempts to access a memory location that it is not allowed to.

The Download File library had to be made from the ground up in the end. The interaction with remote servers is carried out using Boost's ASIO (Asynchronous IO) library. When a page is to be downloaded the URL is disassembled to extract the host, port and path which are then fed into a class in the Download File library called the HTTP Client. The client starts an IO Service which connects to the host on the specified port and sends the request formed by the parameters passed to the client. Boost ASIO does not support timeouts internally so a separate thread runs to track the connection time and terminates the IO Service if a pre-set delay is reached. The Download File class that starts the HTTP Client controls all parameters such as timeout, HTTP Request Verb, Accept types and request data. The response is returned as a string.

MYSQL CLIENT

Basic versions of the tracking lists, before the cache and queue were implemented were built on MySQL tables. The MySQL client allows C++ applications to insert and retrieve data from MySQL database tables. Inserting data is done through MySQL statements as strings and retrieving data returns a vector of maps, where each element in the vector corresponds to a row and each map entry is a mapping from a column/field name to a value.

RDFA

The RDFa library performs three main functions, it can parse RDFa pages and return RDF, decide whether or not it would be worth parsing a HTML page and it can submit RDF data to the Sesame Triple Store. Parsing is achieved by executing a custom Python script which calls pyRDFa, the W3C parser. RDF is sent to Sesame using the Download File library using the POST verb, sending the data itself as the request body.

ROBOTS

The robots class was successfully implemented as described in the design chapter. A small enhancement was added to ensure it was as effective as possible. The User-Agent tests match the first section of the string, anything before a whitespace character, after it was converted to lowercase. This is because spiders often use a long User-Agent to identify themselves, with a URL linking to a crawler help page, but they respond to any rules listed for

the short or prefix version of their User-Agent. For example, Google's web crawler obeys rules listed for "GoogleBot" but supplies the User-Agent of "Googlebot (+http://www.googlebot.com/bot.html)" amongst others.

SERVICE

In Linux for an application to run as a service it must fork itself off, this means (in the case of a service) the section of the code after the fork command is executed in a different process and the original process terminates. The service class is designed to be inherited by an application whose main method can then either call Run or ForkedRun. Run starts the application normally, in a blocking manner whereas ForkedRun forks the main application as another process and terminates the original process.

TCP SERVER

The TCP Server implements Boost's ASIO (Asynchronous IO) library to create an IO service listening on a specified port and with a pointer to a Processor instance which will later be used to deal with requests. When a connection is received a new Session is started which immediately begins an asynchronous read. Every time a request is received it is processed using the aforementioned Processor and the result is returned followed by an asynchronous read again. This loop continues until a socket error occurs (which happens if the client disconnects) and then the Session is stopped and destroyed. The TCP Server is used in almost all of the main applications.

TCP CLIENT

The TCP Client implements Boost's ASIO (Asynchronous IO) library to create an IO service which connects to a host on a specific port. The Client is used to communicate with TCP Service instances and relies on the request-response system used. When the TCP Client's *write* method is called with a request string, the string is sent to the corresponding host then the client waits until a full response (ending in a new line) is received. Upon receipt of a response the data is returned to the calling method.

TCP SERVICE CLIENT

The TCP Service Client extends the basic TCP client, making it more suitable for connecting to TCP Server based services whilst minimizing errors. Its primary use is in the Crawler because it needs to maintain error free, uninterrupted connections to most of the services. Instead of taking a port and a host it takes a service and a host as connection parameters. The service identifier is used to lookup a host from the Setting Server (which tracks the servers on which the services are running, as discussed in section 5.2.3). The request process is similar to that of the TCP Client, sending a request string and waiting for a response, however if an error occurs the client will attempt to reconnect and then resend the request.

SETTING CLIENT

The Setting Client is another extension of the TCP Client which is aimed specifically at communicating with the Setting Server (described in section 5.2.3). It automatically connects to the host named "master" on the default setting server port. It cannot send custom request strings like the TCP Client but provides methods for retrieving and changing setting value pairs as well as requesting host lists for different services. The methods and their necessity

will become clearer with the explanation of the Setting Server and Node Discovery on section 5.2.3.

SERVICE ANNOUNCER

The Service Announcer is yet another extension of the TCP client which has one purpose, to announce the presence of a running service to the Spider system. The class automatically connects to the Setting Server on the host named “master” on the default setting server port. Upon creation the Announcer takes a service name which it sends to the setting server along with the host’s name, this process repeats every 10 seconds while the service is running. This is known as the service’s heartbeat.

LOG CLIENT

The Log Client is the last extension to the TCP client. The Log Client can take log messages and send them to the Log Server. It takes message priority level (Info, Warning, Error and Debug) and the message itself as parameters and converts them to the format the log server requires.

5.1.3: APPLICATIONS

SETTING SERVER AND NODE DISCOVERY

The setting server provides both a centralized location for programs to discover which servers are running services and a place to store and retrieve common settings. Because it keeps track of servers and services it means that services do not always have to run on the same server, however the setting server does to an extent as it is always accessed by the DNS name “master”. DNS is used to resolve server names to IPs so master can be changed to point to another server. When services “check-in” with the setting server they register their host name so when another server needs to connect it still uses DNS as the final resolution step. Every 20 seconds a check is done and services that have not sent a heartbeat since the last check are removed from the internal list of running services, which is why services send a heartbeat every 10 seconds. Settings and values are stored in a setting file provided by the common library so that they are persistent in the event of a failure.

CORE

The core service does not perform any specific functions; it just runs as a service with a service announcer to inform the setting server of which servers are online.

LOG SERVER

The log server takes logging data from TCP clients and archives it in a log file using a storage mechanism provided by the common library. Clients can also access all or chunks of the logs.

ROBOTS SERVER

The robots server, as described in the design chapter takes requests for robots data for a specific domain. It either retrieves it from the cache if it exists, or it downloads and processes the robots file then caches and sends the response.

POWER CACHE (DUPLICATE URLS)

The Power Cache provides a public, central interface to the Cache library. All crawlers can access this to check if a URL has already been crawled. Formatting checks are done upon

receipt of a URL to ensure that it is a valid URL to ensure no arbitrary data is put in the Cache which could potentially slow it down. A Mutex is used to ensure only one request is processed at a time. As the cache did not perform as well as expected (results in Chapter 6) each crawler also cached the last 10,000 URLs it had discovered in an attempt to minimize load on the main cache.

ITD SERVER (INSERT AND TO-DO/ URL QUEUE)

The ITD Server is the service which provides a centralized URL queue; though the design called for the use of a Trie variation this did not hold up under performance tests and a different, Paged Queue system was designed and used (discussed in section 5.2.2). A Mutex is used to ensure only one request is processed at a time.

CRAWLER

The Crawler application itself starts a specified number of threads all running the Crawler class on repeat. The Crawler class starts by requesting a page from the ITD Server, if none are available it waits for a second and tries again as the ITD Server may be processing a page. When a URL has been received to be crawled the domain is extracted and robots data is requested for it so that the robots parser can then check if the URL is allowed to be crawled. If the URL is allowed to be crawled it is downloaded and the content-type is checked, the crawler proceeds to the next step if it is a type html page. A regular expression is used to extract links, the first 100 links are checked against the power cache and if they are unique then they are sent to the ITD Server. Finally the page data is checked by the RDFa library and it if is deemed that it should be parsed for RDFa it is then run through W3C's RDFa parser pyRDFa and the results are uploaded to Sesame. To improve performance each crawler caches around 10,000 Power Cache and Robots requests and their results.

5.1.5: STORAGE

Of the many distributed fault-tolerant file systems 5 were installed and tested; these were MooseFS (MFS), Cloudstore (Kosomos), Lustre, XtreemFS and GlusterFS. If they could be installed successfully their feature set was analysed and if these features were suitable then performance was also tested. All installation tests were done on Ubuntu 9.10 under four nodes (servers) running on the VirtualBox virtual machine.

Cloudstore (Kosomos) was extremely difficult to install, it contained minimal help and proved very hard to compile due to dependency issues. Due to limited time constraints no serious attempts were made to struggle through the dependency mess so Cloudstore was discounted as an option.

Lustre, a High-Availability, striping file system now maintained by Sun (Oracle) was easy to install and configure, however on closer inspection it has no fault-tolerance. The data is striped across all nodes, but only one copy is ever made. File performance parameters had to be manually set so the number of stripes and sizes could not be easily controlled.

XtreemFS was equally easy to set up and configure, but once again its main aim was High-Availability. In this case there was no striping so redundancy was the mechanism that provided High-Availability but data cannot automatically span several machines. Furthermore it is designed to work over the internet with an encryption layer which adds overhead. Adding

files to XtreemFS was not as easy as copying them into a folder, for each file the system had to be told how many times to replicate it.

GlusterFS is very similar to XtreemFS in the way it works. It was more difficult to install on an existing operating system, however it does have its own operating system based on Fedora Core where it is preinstalled. It accesses drives at a low level setting up a form of network based RAID (either 1 OR 0, not both) which again means either redundancy or striping, but not both.

MooseFS (MFS) is a file system project which was very impressive and far surpassed expectations on features, stability and performance! Like Cloudstore, it claims to be based on Google's GFS but unlike Cloudstore it was very easy to install and configure. Access is very simple through a FUSE library, allowing a MFS cluster to be mounted as a folder in Linux. A few small tools allow file's and folder's replication counts to be set either individually or recursively and once set they could be inherited. With all these benefits MFS was used as a storage system.

5.2: TESTING

C++ is a language that does not have built in memory management and garbage collection. This means that it is more susceptible to memory leaks than languages like C# and Java. Valgrind contains tools which can be used to profile programs and detect problems including threading and memory errors. To attempt to ensure that the applications and libraries developed were free of such errors they were all tested with Valgrind and changes were made when errors were found. The libraries had test applications associated with them so that all aspects of the library could also be tested.

CHAPTER 6: RESULTS AND DISCUSSION

The Spider is made up of a number of components, so during testing and evaluation both the individual components were tested, and the system as a whole. All tests were carried out on the same set of hardware, as described in section 6.1.

6.1: HARDWARE AND INFRASTRUCTURE

A cluster of 24 computers was used to run the system. Each machine was running Ubuntu 9.10 with Python and GCC development tools. The Power Cache and ITD Server were run on the most powerful server, the setting server was run on the routing server and crawlers were run on all but the routing server. The routing server also hosted No Machine (remote desktop) sessions and provided DNS services to the cluster. The cluster specification and images are in Appendix A and Appendix B.

6.2: CRAWLER

6.2.1: URL QUEUE

The requirements define the need for a queue structure capable of queuing URLs, but more specifically the politeness requirements dictate that it should analyse the domains of the URLs at some point and re-arrange them so that none are crawled sequentially, and ideally each is crawled as infrequently as possible. Two important tests were carried out, one checked that everything entering the queue eventually left the queue, and the other tested the diversity of URLs leaving the queue. The automatic diversity tuning means the attempted diversity is based on the previous rate of requests, therefore in the tests the rate of requests is controlled so that the ideal diversity can be known.

In the tests the request rate was no less than 10 per second so the expected diversity was never higher than 0.1. The diversity of input URLs matched the expected output diversity because if the input is less diverse than the expected output the queue cannot compensate as it has nothing with which to pad it out, so the peak diversity is recorded (the least diverse output set, where a set is all the URLs output in one second). If the average output diversity was taken it would exactly match the input which is why it is not measured. The results are detailed in Table 2.

Note: a diversity of 0.1 (or 1/10) means out of 10 URLs none will have the same domain, and therefore a diversity 0.01 (or 1/100) means out of 100 URLs none will have the same domain.

	URLs In	URLs Out	Request Rate	Expected Diversity	Actual Peak Diversity
1	100	100	10/sec	0.1	0.1
2	1,000	1,000	50/sec	0.02	0.02
3	10,000	10,000	100/sec	0.01	0.01
4	100,000	100,000	1000/sec	0.001	0.001
5	1,000,000	1,000,000	1000/sec	0.001	0.001

TABLE 2: URL QUEUE TEST RESULTS

The Queue performed as expected. A potential bottleneck does exist; if lots of requests are made to the queue and the queue has no processed data and only temporary data then it

must process the temporary data causing the requests to hang until it is done. This often occurs at the beginning and end of a crawl.

6.2.2: Politeness

The politeness requirements are implemented in three parts of the system, the URL Queue, the URL Cache and in the Crawler. The URL Queue and Cache are discussed in sections 6.2.1 and 6.2.5 respectively. The Crawler checks whether a URL is allowed to be crawled by testing it against the domain's robots.txt rules (the Robots Exclusion Standard). There are four main areas to be tested in the Robots parser: the Disallow keyword, the Allow keyword, User-Agent checking and specific rule matching. Tests were carried out using the following robots data:

```
User-Agent: RDFaSbot
Allow: /bad/but/ok
Disallow: /be
Disallow: /bad/

User-Agent: *
Disallow: /secret/
Allow: /secondary/
```

The robots data was tested against different user agents and paths to see if the expected results matched up with the actual results. "Yes" means the parser deemed the URL to be crawl-able and "No" means it did not. The results are listed in Table 3.

	User-Agent	Path	Expected Result	Actual Result
1	RDFaSbot	/bad	No	No
2	RDFaSbot	/base	No	No
3	RDFaSbot	/bad/more	No	No
4	RDFaSbot	/bad/but	No	No
5	RDFaSbot	/bad/but/ok	Yes	Yes
6	RDFaSbot	/bad/but/ok/and	Yes	Yes
7	RDFaSbot	/bad/but/okeydokey	Yes	Yes
8	Googlebot	/test/	Yes	Yes
9	Googlebot	/secret/code	No	No
10	Googlebot	/secondary/more-specific	Yes	Yes
11	RDFaSbot	/secret/	Yes	Yes

TABLE 3: ROBOTS PARSER TEST RESULTS

Tests 1-7 checked User-Agent matching and basic disallow and allow rules. Tests 8-10 checked wildcard rules with test 10 also checking a more specific (or longest) rule overrode a previous rule. Test 11 checked that a matched User-Agent was not also tested against the wildcard rules. All the tests performed as expected so it can be assumed that the crawler obeys the Robots Exclusion Standard.

6.2.3: Downloading Pages

Downloading pages, when required, will need to be done as quickly as the internet connection and corresponding server will allow, so that the task does not become a limiting

factor. As the requirements specified it also needs to be able to timeout so that it does not indefinitely block the crawling thread if the server is too busy to respond or packets are lost. The requirements also state that the crawler must only download or accept files which contain text or html, and it should ignore other files. To test the Download File class 6 test pages were created to test specific scenarios. Tests were done over a 100Mbit LAN connection to a web server, the default timeout is 30 seconds. Table 4 shows the results.

	Test Scenario	Expected Result	Actual Result
1	Page with 30 Second Delay	Client Timeout	Client Timeout
2	Page with 29 Second Delay	Download Succeeded	Download Succeeded
3	10Mb HTML Page	Download at 12 Mb/s	Download at 11.5 Mb/s
4	PDF Document	Invalid content type	Invalid content type
5	Image	Invalid content type	Invalid content type
6	2Kb HTML Page	Download Succeeded	Download Succeeded

TABLE 4: DOWNLOAD FILE CLASS TEST RESULTS

Tests 1 and 2 checked the timeout threshold with test 1 taking as long as the timeout to load, with the small added delay of latency it just went over the 30 second mark and caused a timeout. Test 3 is a large page to measure the maximum I/O rate, with a 100Mbit connection the best speed achievable is 12 Mb/sec; the result of 11.5 Mb/sec is more than acceptable as packet overhead means full theoretical speed is rarely achieved. Tests 4 and 5 check content type restrictions are working and finally test 6 is an average page to be accepted and downloaded. Contrary to the requirements, the Download File class is not fully HTTP 1.1 compliant due to time constraints.

6.2.4: Extracting Links

So that the crawler can find other pages to crawl the requirements specify that it must be able to extract the URL from links on pages. The use of regular expressions means that even if a page is not fully compliant with a HTML standard the links should still be extractable unless the links themselves are largely malformed. Tests were done on the link extraction system to make sure that it can extract at most 100 links from a page with valid markup and to discover how it handles pages with invalid markup. Common markups were tested and the validity of a page's markup was determined by W3C's validator service. The results are listed in Table 5.

	Markup Version	Valid	Links on Page	Links found
1	XHTML 1.0 Transitional	Yes	1	1
2	XHTML 1.0 Transitional	Yes	100	100
3	XHTML 1.0 Transitional	Yes	200	100
4	XHTML 1.0 Transitional	No	100	100
5	HTML 4.01 Strict	Yes	1	1
6	HTML 4.01 Strict	Yes	100	100
7	HTML 4.01 Strict	Yes	200	100
8	HTML 4.01 Strict	No	100	100
9	XHTML + RDFa	Yes	1	1
10	XHTML + RDFa	Yes	100	100
11	XHTML + RDFa	Yes	200	100
12	XHTML + RDFa	No	100	100

TABLE 5: LINK EXTRACTION TEST RESULTS

Tests 3, 7 and 11 checked that the extraction mechanism obeyed the 100 links limit whereas tests 1, 2, 5, 6, 9 and 10 checked that the correct number of links could be extracted. Tests 4, 8 and 12 checked how pages with invalid markup behaved. On these pages all link markup was valid but other parts were not and links were well distributed including in the head tags and outside the html tags. This shows that the extraction method was not strict about markup validity. In theory links outside the body should be ignored. Accepting invalidly positioned links in markup may make the spider susceptible to spamming.

6.2.5: URL CACHE

Most web pages contain links, and every link on the page represents a request to the URL cache therefore if the URL cache cannot rapidly check for duplicates it will cause the whole system to slow down. Some large crawls did actually cause the system to slow down drastically, as described in section 6.4.1.

The basic Trie used for the URL cache had several potential performance enhancements available, comparisons were done to test which combination of enhancements performed best. Comparisons were also done between using the MFS distributed file system and storage on the local hard drives. Several tests were done for each configuration using different numbers of Trie entries (1000, 10000 and 100000); each entry was written and then checked. For each test the duration was measured in milliseconds. All tests were carried out on the machine on which the URL Cache was also running. The results are shown in Table 6 and Figure 13.

The table notation is as follows: HDD indicates the test was run on the internal hard drive, MFS indicates that the test was run on the distributed file system. C means the C FILE*class was used for disk I/O and C++ means the std::fstream class was used. The presence of BCM shows that recent Trie Block Entries were cached in memory and finally NFL means that data streams were not flushed to disk immediately after a write and stream flushing was left to the discretion of the operating system. Note: NFL is only worth using with BCM as reading un-cached data forces a flush which renders deliberately not flushing pointless.

	Number of Trie Entries		
Test	1,000	10,000	100,000
HDD C	260	3185	37105
HDD C BCM	175	2345	29912
HDD C BCM NFL	178	2352	30673
HDD C++	446	6114	75889
HDD C++ BCM	155	2149	28004
HDD C++ BCM NFL	158	2211	28140
MFS C	6038	57511	810152
MFS C BCM	2201	26887	395676
MFS C BCM NFL	3686	44690	583140
MFS C++	2718	39309	432759
MFS C++ BCM	710	8086	94279
MFS C++ BCM NFL	724	8517	93132
MFS Average	2680	30833	401523
HDD Average	229	3059	38287
Basic Average	2366	26530	338976
BCM Average	810	9867	136968
BCM NFL Average	1187	14443	183771

TABLE 6: TRIE PERFORMANCE TEST RESULTS

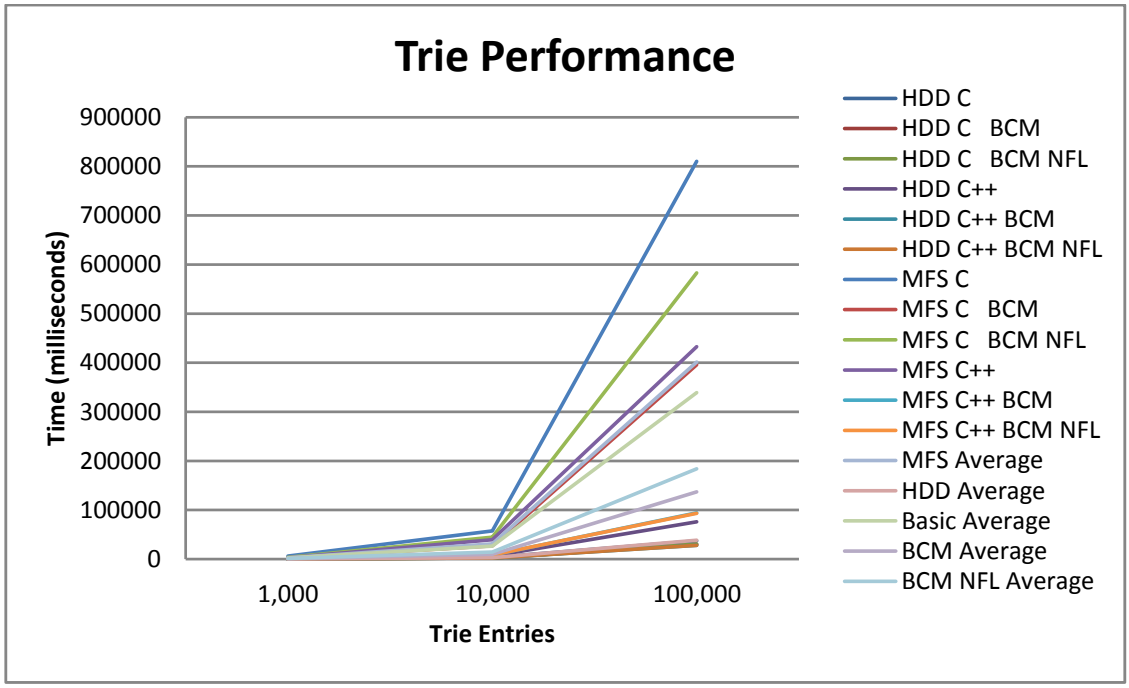


FIGURE 13: A GRAPH VISUALIZING THE TRIE PERFORMANCE TEST RESUTLS

The Trie implementation rapidly seeks to different chunks of the file and this type of file access was noticeably faster, by at least 10 times on the local drive. Using MFS is a vast performance bottleneck, however using the local drive does not provide the mass storage and redundancy options. It is also interesting to note that C FILE* works faster on the local hard drive than the C++ std::fstream but on MFS it is the other way round.

Surprisingly, forcing the data streams to flush after block writes actually performed better than leaving it up to the operating system. It was expected that these results would be the other way round. The Trie also slowed down exponentially as more entries were added. This is expected as URL characters being represented as nodes become more spread out in the underlying file if it has a common prefix that was added much earlier, which increases seek distance and time.

6.3: Indexing Accuracy

The indexing accuracy tests measure how many pages the crawler discovers from a known set and how many Triples it finds from a known amount. A program was made specifically for this test which generates webpages with content that includes triples. The program takes a number of levels deep to make the website, and the number of pages to create at each level. The number of triples on each page can also be specified. Four main sets of tests were done; each had a different number of triples per page, from 0 to 3. Then for each main set 5 sites were generated each with different numbers of pages. The results are in Table 7.

	Pages Generated	Triples Generated	Pages Found	Triples Found
1	2	0	2	0
2	8	0	8	0
3	40	0	40	0
4	341	0	341	0
5	3906	0	3906	0
6	2	2	2	2
7	7	7	8	8
8	40	40	40	40
9	341	341	341	341
10	3906	3906	3906	3906
11	2	8	2	8
12	7	14	8	16
13	40	80	40	80
14	341	682	341	682
15	3906	7812	3906	7812
16	2	6	2	6
17	7	21	8	24
18	40	120	40	120
19	341	1023	341	1023
20	3906	11718	3906	11718

TABLE 7: SPIDER ACCURACY TEST RESULTS

The Spider successfully managed to find all pages and triples on the test sites. It was given a progression regular expression to make sure that it did not crawl outside of the evaluation website being tested at the time. The depth racing issue (discussed later) did not occur because the initial crawl depth was set much higher than the actual overall depth of the site.

Every time a new site was tested the Sesame Triple Store's repository had to be emptied so that the number of triples on the next site could be counted. Unfortunately, after a few of the

tests Sesame was unable to fully empty all the triples which required the Tomcat Server to be stopped and the repository manually deleted.

6.4: Scalability

6.4.1: Scalability, speed and overall performance

A requirement of the Spider was that it should be scalable and efficient so that hopefully it can crawl as quickly as environmental conditions permit (such as internet connection speed).

Table 8 shows the results of several test crawls, all starting from the University of Sheffield's website (http://www.sheffield.ac.uk). Tests were done using different numbers of nodes (servers), different crawl depths (which roughly governed the size of the crawl) and the URL Caching method. The duration is in seconds and the rate is pages per second.

	Depth	Nodes	URL Cache	Duration	Triples	Pages	Rate	Data D'loaded
1	5	23	MFS Trie	2558	47158	193618	76	20GB
2	5	23	HDD Trie	2140	73061	259008	121	30GB
3	6	23	MFS Trie	STOPPED DUE TO SLOW SPEED				
4	6	23	HDD Trie	12214	301219	1832157	150	150GB

TABLE 8: SPIDER PERFORMANCE RESULTS

All crawls slow down towards the end because URLs are rapidly being requested but few are coming in so the *Insert/To Do Server* uses time to try and maintain diversity but struggles, and there are few if any new URLs to pad out common ones. In theory a larger crawl may not suffer from this problem because more sites should be discovered which may assist in maintaining the diversity of the URLs being sent to crawlers.

Crawl 1 ran noticeably slower than crawl 2 and this is simply down to using MFS as the storage system for the URL Cache. As demonstrated in section 6.2.5 the Trie structure used in the URL Cache performs poorly on MFS. Crawl number was stopped soon after it was started because it slowed down to a rate of 4 pages per second. This is because the URL Cache's Trie was taking a long time to process queries. Due to the crawl being deeper the URL Cache grew more rapidly it required disk seeks over greater distances and MFS does not cope well with this situation. As crawl number 4 shows, when using the Hard Drive for the URL Cache's Trie storage the crawl was fast.

It is also interesting to note that the number of pages crawled and triples found on crawls of the same depth differ, this is discussed in the next section.

6.4.2: Stopping and Depth Racing

In the Spider's implementation, depth was used as a stopping condition which produced a side-effect which should have been expected; the number of pages found in crawls with identical starting parameters could differ greatly. It is a form of race condition where the order in which the pages are crawled affects the number crawled in total. To illustrate a simple hierarchy will be used to represent a crawl of depth 2. Figure 14 shows the crawl finding 8 pages and Figure 15 shows the same crawl finding only 3 pages because they were crawled and discovered in a different order.

Each node in the diagram represents a page and the crawl always starts with the node at the top (node number 1). The number in the middle represents the depth at which the page was discovered and the number on the arrow is the order the discoveries were made. The number adjacent to each node is to identify them for easy reference in the explanations.

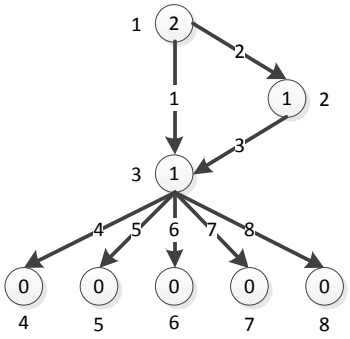


FIGURE 14: CRAWL PROGRESSION IN DEPTH RACING SCENARIO 1

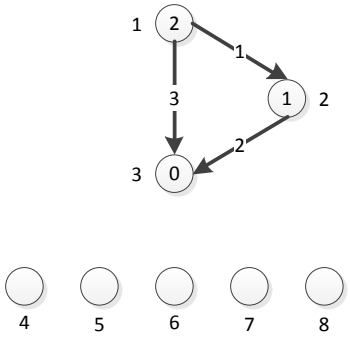


FIGURE 15: CRAWL PROGRESSION IN DEPTH RACING SCENARIO 2

In Figure 14 the first node discovered from the start (node 1) is node 3 which is given a depth of 1, then node 2 is discovered. Node 2 then finds node 3 but as it already has a depth of 1 it is not overwritten by 0 as the URL cache already knows of this page and won't add it to the URL Queue again. Finally node 3 discovers nodes 4-8 and as they are at depth 0 no further attempts are made to discover if they lead anywhere.

In Figure 15 the first node discovered from the start (node 1) is node 2 which is given a depth of 1. Node 2 then finds node 3 and gives it a depth of 0. Node 1 also finds node 3 afterwards but does not reassign it a depth of 1 as it already has a depth of 0 and the URL cache will prevent it from being re-queued with a different depth. This means that nodes 4-8 are never discovered.

The example here would never occur because of the small number of pages, but when the URL Queue becomes large and multiple multi-threaded crawlers are used then this race condition does occur. This is due to external delays affecting the order in which URLs finish being crawled and new URLs are sent to the queue which then alters the result of the diversity calculations which changes the order in which subsequent URLs are crawled. Any stopping condition may cause this, be it depth, total pages crawled or even until a specific page was found.

CHAPTER 7: CONCLUSIONS

7.1: OVERVIEW

Crawling the web is non-trivial and intricate task. There are scores of potential bottlenecks as the components of a scalable system are very tightly coupled. During development every time one fix was applied to a slower component the whole system sped up, however it often exposed another bottleneck or a coding error. Crawling also requires lots of care and responsibility as a crawler will potentially access hundreds of pages per second and if the order in which URLs are crawled in is not carefully controlled it could cause websites to be over-crawled and crash denying access to legitimate users.

The crawler's performance was slightly disappointing at times when using the MooseFS distributed file system as the storage method however this was easily combatted by using storage on local drives.

Extracting *only* directly specified RDFa elements can be problematic because some of the better RDFa parsers also export implicit RDFa such as bookmarks and style sheets. A totally compliant parser can be difficult and time consuming to create and until more customization is available a layer between exporting and storing to filter the data is needed.

7.2: PROGRESS

Though a basic working crawler had been created before the Survey and Analysis it was completely re-written in the last three months. The decision to rewrite it all was made because a lot of the code was not very generic or transferrable between internal applications which meant that creating most of the services was very time consuming. The newer version of the crawler uses language features to make the code more efficient and generic. The project progress is informally documented at http://www.rdfas.com/blog.

7.3: FUTURE IMPROVEMENTS

There are several elements of the Spider system which could have been implemented better, or implemented in the first place to improve accuracy and/or performance.

7.3.1: URL CACHE

The overall performance of the URL Cache (Trie) was disappointing, the high level of seeks meant that (on MFS at least) the performance was *very* poor. Further work and research needs to be done on storing some sort of index on a file system. It may be worth considering the technique used in the URL Queue where temporary data are processed and stored in chunks so major data shifting is committed in memory and not on disk.

7.3.2: DNS CACHING AND PREFETCHING

Every time a page was requested from a web server, the Download File class first had to resolve the domain name of the site to an IP address. This places a lot of load on local and upstream DNS servers. Many DNS servers will cache lookups that they cannot authoritatively respond to, which will help reduce load, however by design each Download File instance on

each thread will make DNS requests even if another thread in that process has already resolved the domain in question, caching DNS results at the crawler level will help minimize requests to the local DNS server. Furthermore the URL Queue knows what domains are going to be crawled in the near future and it could instruct crawlers to asynchronously look up in advance (pre-fetch) the domain names to minimize resolution delays when the time comes to crawl a page on that domain.

7.3.3: Distributed Locking

Locking is commonly used to ensure that resources are only accessed or modified by one process at a time. Distributed locking would mean that all the crawlers could secure shared access to resources like the cache or the queue without the need of a cache server or queue server which would then mean that these single purpose servers are no longer a potential point of failure. Removing specific servers in place of shared access to resources also reduces the overhead on a specific server as it would distribute the workload but may imply the need for shared resources such as a file system.

7.3.4: Triple filtering

Some Triples extracted by the parser pyRDFa were related to style elements of the page, most commonly the style sheet. These were not explicitly declared through namespace declarations but implied by the HTML tags that link to the style sheet. As all Triples were sent to the Triple Store it means the total count of Triples showed specifically written Triples as well as the implicit style Triples which gives a false representation of how many Triples have been created and found. A potential solution would be to parse the Triples after they have been extracted from a page, but before they are sent to the Triple Store and remove any that match certain patterns.

7.4: Further work

7.4.1: Tree storage

Trees can be a very efficient method for storing data, especially hierarchical data and most of the web is a massive interlinked hierarchy. Domains, folders and HTML pages all follow a hierarchical structure and thus spiders (amongst other things) can make great use of trees. Unfortunately trees perform poorly when stored on sequential storage with large (compared to memory) and variable seek delays. Disk caching does improve access to blocks of data which are near each other however when related data is sparsely separated seek times can make access slow. When trees are stored in memory seek time and access is extremely fast and not that variable meaning trees can perform very well.

As many data structures can become very large when populated it is often not possible to store it all in memory. Unfortunately there seems to be minimal published research on efficient algorithms for storing tree structures on disk based (or persistent) storage mechanisms. With solid state drives (SSDs) slowly becoming quite popular this research may not eventually be needed but they are currently still very expensive compared to traditional magnetic disk based hard drives.

7.4.2: Distributed file systems

In applications where lots of data needs to be processed and/or stored they are likely to require more space than is currently available on a single physical disk. Many solutions that are currently commercially available still have the risk of a single point of failure when being accessed by several clients and it is most often the host machine which connects the storage to the clients. Distributed file systems avoid this common point of failure as data is replicated amongst many nodes and in many cases master nodes, which would otherwise be a point of failure, can have secondary backups ready to take over in the event of a failure.

There are several implementations of distributed file systems, many of which are open source. Unfortunately they are often poorly documented and overly complicated to setup or restrictive when it comes to interacting with them. Work on more openly documented and flexible distributed file systems would be a positive step forwards.

7.4.3: Scalable Triple Stores

Though there are several Triple Stores available, only one (4Store) seems to be designed to cope with a high level of scalability and actually describes the ability to have multiple storage nodes in its documentation. Unfortunately it is suggested that these storage nodes are powerful, resource rich servers. Though Triple Stores have a query language which vaguely resembles SQL's query language it is often claimed that an SQL style database engine does not make an ideal backend for Triple Stores even though many are implemented on top of a common database engine like MySQL. Many powerful database engines have now emerged; given time and increased industry use more research and powerful solutions for Triple Stores may also emerge. For projects like this one to succeed a very reliable Triple Store would be useful as Sesame suffered a few problems (as noted in section 6.3).

7.4.4: Open spider research

While writing this paper it has become very clear that there is little in-depth published research about the detailed mechanisms involved in making a Spider. This is almost certainly because any successful attempts could potentially be used for commercial gain, however at this point that is questionable as the search engine market is very much saturated and dominated by a handful of companies. Techniques used to make large scale crawlers are very transferrable and constitute interesting and valuable areas of research, for example storing large volumes of data, efficient forward and reverse indexes and storing complex data structures on a file system. It would be desirable to see more research published on these and related subjects in the future and carrying on the work presented here would be a good start.

REFERENCES

AC & NC (2009) *Raid Levels*, 1 Jan, [Online]. AC & NC. http://www.raid.com/04_01_0_1.html [Accessed 21 Nov 2009].

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

Berners-Lee, T. (2005) *RFC3986: Uniform Resource Identifier (URI): Generic Syntax*, [Online].. http://www.ietf.org/rfc/rfc3986.txt [Accessed 14 Apr 2010].

Brin, S. and Page, L. (1998) *The Anatomy of a Large-Scale Hypertextual Web Search Engine*, Standford University.

Broadband.org (2009) *Fastest Broadband Speeds - Top 3 Speeds*, 1 November, [Online]. Broadband.org.
http://www.broadband.org/fastest_broadband.html [Accessed 25 November 2009].

Cafarella, M. and Cutting, D. (2004) Building Nutch: Open Source Search, 2, (2), Available: 1542-7730.

Cody, D. (2001) *Using Apache to stop bad Robots*, 22 Aug, [Online]. Evolt.org. http://evolt.org/article/Using_Apache_to_stop_bad_robots/18/15126/ [Accessed 25 Nov 2009].

Dean, J. (2008) *Jeff Dean on Google Infrastructure*, 11 Jul, [Online]. Perspectives. http://perspectives.mvdirona.com/CommentView,guid,dd99224c-5fe4-4b4b-80fe-0600e9633429.aspx [Accessed 10 Nov 2009].

Fielding, R., Irvine, U., Gettys, J., Compaq, Mogul, J., Frystyk, H., MIT, Masinter, Xerox, Leach, P., Microsoft and Berners-Lee, T. (1999) *RFC2616 Hypertext Transfer Protocol -- HTTP/1.1*, [Online]..
http://www.ietf.org/rfc/rfc2616.txt [Accessed 14 Apr 2010].

Fielding, R., Irvine, U., Gettys, J., Compaq, Mogul, J., Frystyk, H., MIT, Masinter, Xerox, Leach, P., Microsoft and Berners-Lee, T. (1999) *RFC2616 Hypertext Transfer Protocol -- HTTP/1.1*, [Online]..
http://www.ietf.org/rfc/rfc2616.txt [Accessed 10 April 2010].

Ghemawat, S., Gobioff, H. and Leung, S.-T. (2003) *The Google File System*, ACM.

Goodrich, M.T. and Ramassia, R. (2004) *Data Structures and Algorithms in Java*, in Goodrich, M.T. and Ramassia, R. *Data Structures and Algorithms in Java*, Wiley.

Google (2008) *We knew the web was big*, 25 Jul, [Online]. The Official Google Blog. http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html [Accessed 18 Nov 2009].

Internet Archive (2009) *PetaBox*, 20 Nov, [Online]. Internet Archive. http://www.archive.org/web/petabox.php [Accessed 20 Nov 2009].

Internet World Stats (2009) *Internet Usage Statistics*, 18 Nov, [Online]. Internet World Stats. http://www.internetworldstats.com/stats.htm [Accessed 18 Nov 2009].

iProspect (2006) *iProspect Search Engine Behaviour Study*, April, [Online]. iProspect. http://www.iprospect.com/premiumPDFs/WhitePaper_2006_SearchEngineUserBehavior.pdf [Accessed 15 April 2010].

Jack Rusher *Triple store*, [Online]. W3. http://www.w3.org/2001/sw/Europe/events/20031113-storage/positions/rusher.html [Accessed 26 Nov 2009].

Lardinois, F. (2009) *Google Patents Its Homepage*, 2 Sep, [Online]. ReadWriteWeb. http://www.readwriteweb.com/archives/google_patents_its_homepage.php [Accessed 20 Nov 2009].

Majestic-12 (2009) *Majestic SEO*, 18 Nov, [Online]. Majestic SEO. http://www.majesticseo.com/ [Accessed 18 Nov 2009].

Matthew Komorowski (2009) *A History of Storage Cost*, 01 July, [Online]. mkomo.com. http://www.mkomo.com/cost-per-gigabyte [Accessed 12 March 2010].

Mono Project (2010) *Mono: getting started*, [Online]. Mono. http://www.mono-project.com/Start [Accessed 10 April 2010].

Network Working Group (1998) *RFC2460*, 1 Dec, [Online]. IETF Tools. http://tools.ietf.org/html/rfc2460 [Accessed 20 Nov 2009].

Network Working Group (1999) *Hypertext Transfer Protocol -- HTTP/1.1*, 1 May, [Online]. IETF.
http://tools.ietf.org/html/rfc2616 [Accessed 22 Nov 2009].

Pages, T.W.R. (1994) *A Standard for Robot Exclusion*, 1 May, [Online]. The Web Robots Pages. http://www.robotstxt.org/orig.html [Accessed 25 Nov 2009].

RDFa (2010) *Consume*, 14 April, [Online]. RDFaWiki. http://rdfa.info/wiki/Consume [Accessed 17 April 2010].

W3C (2004) *RDF Primer*, 10 Feb, [Online]. W3C. http://www.w3.org/TR/rdf-primer/ [Accessed 20 Nov 2009].

W3C (2008) *RDFa Primer*, 14 Oct, [Online]. W3C. http://www.w3.org/TR/xhtml-rdfa-primer/ [Accessed 19 Nov 2009].

W3C (2010) *RDFa Distiller and Parser*, 9 March, [Online]. W3C. http://www.w3.org/2007/08/pyRdfa/ [Accessed 16 April 2010].

Wikipedia (2009) *Meta Element*, 14 Nov, [Online]. Wikipedia. http://en.wikipedia.org/wiki/Meta_element#The_robots_attribute [Accessed 25 Nov 2009].

Wikipedia (2009) *Robots exclusion standard*, 12 Nov, [Online]. Wikipedia. http://en.wikipedia.org/wiki/Robots_Exclusion_Standard [Accessed 23 Nov 2009].

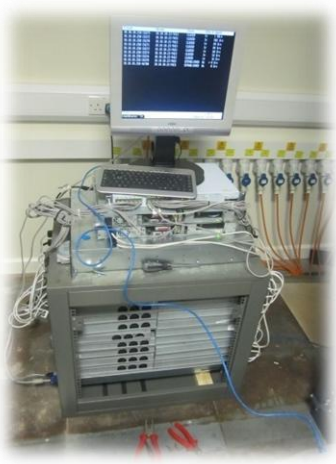
Wikipedia (2010) *Hypertext Transfer Protocol*, 14 April, [Online]. Wikipedia. http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol [Accessed 15 April 2010].

Wikipedia (2010) *URL Normalization*, 1 April, [Online]. Wikipedia. http://en.wikipedia.org/wiki/URL_normalization [Accessed 1 April 2010].

Wikipedia (2010) *Web Crawlers*, 19 April, [Online]. Wikipedia. http://en.wikipedia.org/wiki/Web_crawler [Accessed 20 April 2010].

APPENDICES

APPENDIX A: RDFAS CLUSTER DEPLOYED IN THE DCS SERVER ROOM


APPENDIX B: CLUSTER SPECIFICATIONS

Node	CPU	RAM	Hard Disk	Roles
1	2.4 GHz P4	1 GB	80 GB	Crawler, MFS Node
2	2.4 GHz P4	256 MB	80 GB	Crawler, MFS Node
3	2.4 GHz P4	256 MB	80 GB	Crawler, MFS Node
4	2.4 GHz P4	256 MB	80 GB	Crawler, MFS Node
5	2.4 GHz P4	512 MB	80 GB	Crawler, MFS Node
6	2.4 GHz P4	256 MB	6 GB	Crawler, MFS Node
7	2.4 GHz P4	256 MB	80 GB	Crawler, MFS Node
8	2.4 GHz P4	512 MB	80 GB	Crawler, MFS Node
9	2.4 GHz P4	512 MB	80 GB	Crawler, MFS Node
10	2.4 GHz P4	512 MB	80 GB	Crawler, MFS Node
11	2.4 GHz P4	512 MB	80 GB	Crawler, MFS Node
12	2.4 GHz P4	512 MB	80 GB	Crawler, MFS Node
13	2.4 GHz P4	256 MB	80 GB	Crawler, MFS Node
14	2.4 GHz P4	512 MB	10 GB	Crawler, MFS Node
15	2.4 GHz P4	256 MB	80 GB	Crawler, MFS Node
16	2.4 GHz P4	512 MB	80 GB	Crawler, MFS Node
17	2.4 GHz P4	512 MB	80 GB	Crawler, MFS Node
18	2.4 GHz P4	512 MB	80 GB	Crawler, MFS Node
19	2.4 GHz P4	768 GB	80 GB	Crawler, MFS Node
20	2.4 GHz P4	1 GB	80 GB	Crawler, MFS Node
21	2.4 GHz P4	1 GB	80 GB	Robot Server, MySQL, Crawler, MFS Node
22	2.4 GHz P4	256 MB	80 GB	Crawler, MFS Node
23	2 x 2 GHz Xeon	2 GB	20 GB	ITD Server, Power Cache, Sesame, Crawler, MFS Master
"router"	1GHz P3	1 GB	20 GB	Router, Setting Server, Log Server, DNS
Total:	57.8 GHz	13.75 GB	1.6 TB	